Optimizing Program Performance

Reading: B&O Chapter 5

Preliminaries → Overview, Big-O, etc.

Compilers → What they are good at and what they are not

Example w/ some basic methods → code motion, etc.

Example w/ HW dependent methods → unrolling, parallelization

Dealing w/ flow control → data dependence, conditional moves

Program Development Paradigm

HISTORIC

Artifact	Problem Description	→	Algorithm	→	Program	→	Executable
Who does transformation		*programmer*		*programmer*		*compiler*	
Model	**generally ad hoc**		**Abstract, e.g., “RAM” or “PRAM”**		**Abstract, e.g., “RAM” or “PRAM”**		**Target Machine**

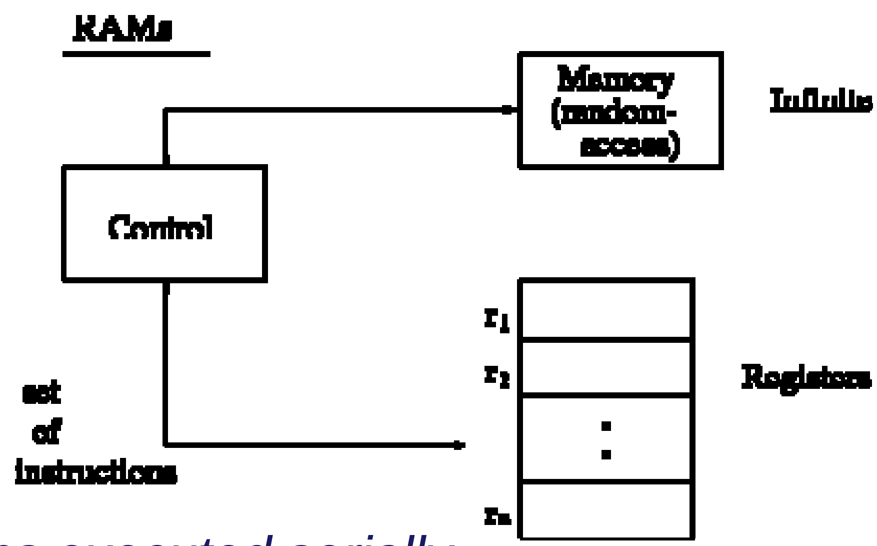
w/ complex architectures

Artifact	Problem Description	→	Algorithm	→	Program	→	Executable
Who does transformation		*programmer*		*programmer*		*compiler*	
Model	**Formal methods BTSOTC**		**Also Target Machine**		**Also Target Machine**		**Families of Target Machines**

How to Select Algorithm – Theory

Best Algorithm → Algorithm that has best ***asymptotic complexity*** when running on a ***RAM***

– Here, **RAM = *Random Access Machine*** NOT *Random Access Memory*



Two questions:

1. What is a **RAM**?
2. What is **Asymptotic Complexity**?

A **RAM** is an abstract computer that resembles most computers pre-1970:

-- *single thread of control with instructions executed serially*

-- *single memory image with uniform access time*

Note: this model was HUGELY influential and incredibly useful!! Most algorithm complexities (as you learned in EC330) are based on this or similar models.

Asymptotic Complexity *a.k.a. Big-O notation*

For functions f(n) and g(n), we say that "f(n) is Big-O of g(n)" if

> **There exists a constant $c > 0$ such that there exists a constant n_0 that for all $n \geq n_0$, $0 \leq f(n) \leq c * g(n)$**

We write f(n) = O(g(n)), but the "=" is not the usual meaning.

The intention is to ignore multiplicative constants and low-order additive terms to allow us to say

> **$3*n^2 + 14*n - 7 = O(n^2)$**
>
> **$0.0007 * n^3$ is *not* $O(n^2)$**

Technically, $\mathbf{10^{1,000,000} \times n}$ is **O(n)**, but programs with that running time are very slow. This is known as "hiding" the constants within the Big-O. Knowing when/how to "hide the constants" and when to take them seriously is a critical part of selecting algorithms for high performance code. In general, the more complex the architecture, i.e., the further away from the RAM model, the more important the constants. In this course we care about the constants!

Optimizing Program Performance -- Intro

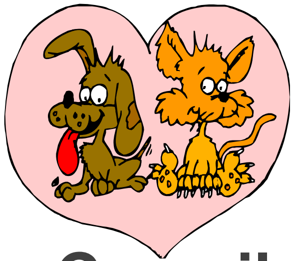
<u>Primary objective</u> in writing a program → **make sure it runs correctly!!**

<u>Other objectives</u> → **readable, understandable, debugable, testable, maintainable, extendable, secure, …**

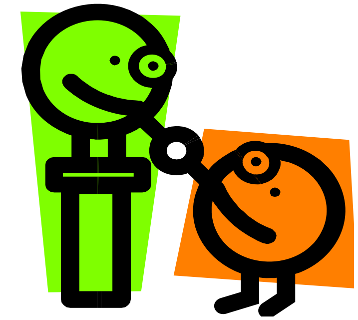
Making the program run <u>fast</u> is also often critical, but we can never lose sight of the "other objectives." High performance programming thus involves trade-offs.

<u>Much involved in HPP:</u>

1. Select appropriate algorithms (e.g., quick sort versus insertion sort)
   - *As you will learn, the hidden constants are often critical and can lead to selecting algorithms that are not asymptotically optimal*
2. Write code so that it is optimizable *by the compiler*
   - *That's right, the compiler. Compilers are brilliant but are far from perfect and often behave unpredictably.*



Your friend the compiler



Compilers have several inherent limitations:

1. Compilers (in general) cannot recognize, create, or select algorithms
2. Compilers insist on generating safe code for any conceivable situation, whether or not that situation will actually occur. Thus compilers often do not apply all of the optimizations they could (or *you* would).
3. Compilers apply MANY optimizations. These often interfere with each other in unpredictable ways.

Therefore, your job involves:

1. getting the algorithm right
2. helping out the compiler, especially, by not introducing "optimization blockers" that prevent it from doing its work
3. writing code so that the appropriate optimizations are applied, perhaps by you, in the code itself
4. experimenting with variations in algorithm, code, and compiler settings – that is, iterating

Performance Realities

- ***There's more to performance than asymptotic complexity***

- **Constant factors matter too!**
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Optimizing Compilers

- **Provide efficient mapping of program to machine**
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- **Don’t (usually) improve asymptotic efficiency**
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- **Have difficulty overcoming “optimization blockers”**
  - potential memory aliasing
  - potential procedure side-effects

Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
  - 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**

Generally Useful Optimizations

- **Optimizations that you or the compiler should do regardless of processor / compiler**

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

```
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];
```

Compiler-Generated 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;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
        *rowp++ = b[j];
```

Where are the FP operations?

Register Convention

Inputs
%rcx ← n
%rdx ← i
%rdi ← a
%rsi ← b

In code
%rax ← n, then t
%rdx ← rowp
%r8d,%r8 ← j

```
set_row:
        testq   %rcx, %rcx              # Test n
        jle     .L4                     # If 0, goto done
        movq    %rcx, %rax              # rax = n
        imulq   %rdx, %rax              # rax *= i
        leaq    (%rdi,%rax,8), %rdx     # rowp = A + n*i*8
        movl    $0, %r8d                # j = 0
.L3:                                   # loop:
        movq    (%rsi,%r8,8), %rax      # t = b[j]
        movq    %rax, (%rdx)            # *rowp = t
        addq    $1, %r8                 # j++
        addq    $8, %rdx                # rowp++
        cmpq    %r8, %rcx               # Compare n:j
        jg      .L3                     # If >, goto loop
.L4:                                   # done:
        rep ; ret
```

Reduction in Strength

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

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

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    – On Intel Nehalem, integer multiply requires 3 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;
}
```

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;
```

```
long 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;
```

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

1 multiplication: i*n

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Optimization Blocker #1: Memory Aliasing

<u>Memory Aliasing</u> ⇔ two pointers may designate the same memory location

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

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

Do twiddle1 and twiddle2 have the same behavior?

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

What about this code?

Memory Matters

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L53:
        addsd   (%rcx), %xmm0           # FP add
        addq    $8, %rcx
        decq    %rax
        movsd   %xmm0, (%rsi,%r8,8)     # FP store
        jne     .L53
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double B[3] = {0,0,0};

sum_rows1(A, B, 3);
```

Value of **B**:

```
init:   [0, 0, 0]
i = 0: [3, 0, 0]
i = 1: [3, 28, 0]
i = 2: [3, 28, 224]
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Value of **B**:

```
init:   [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 27, 16]
i = 2: [3, 27, 224]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Removing Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
         double val = 0;
         for (j = 0; j < n; j++)
             val += a[i*n + j];
         b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L66:
        addsd    (%rcx), %xmm0    # FP Add
        addq     $8, %rcx
        decq     %rax
        jne      .L66
```

- No need to store intermediate results

Optimization Blocker: Memory Aliasing

- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Optimization Blocker #2: Procedure Calls

<u>Memory Aliasing</u> ⇔ two pointers may designate the same memory location

```
int f();

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

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

Do func1 and func2 have the same behavior?

```
int counter = 0;

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

What about this code?

Example

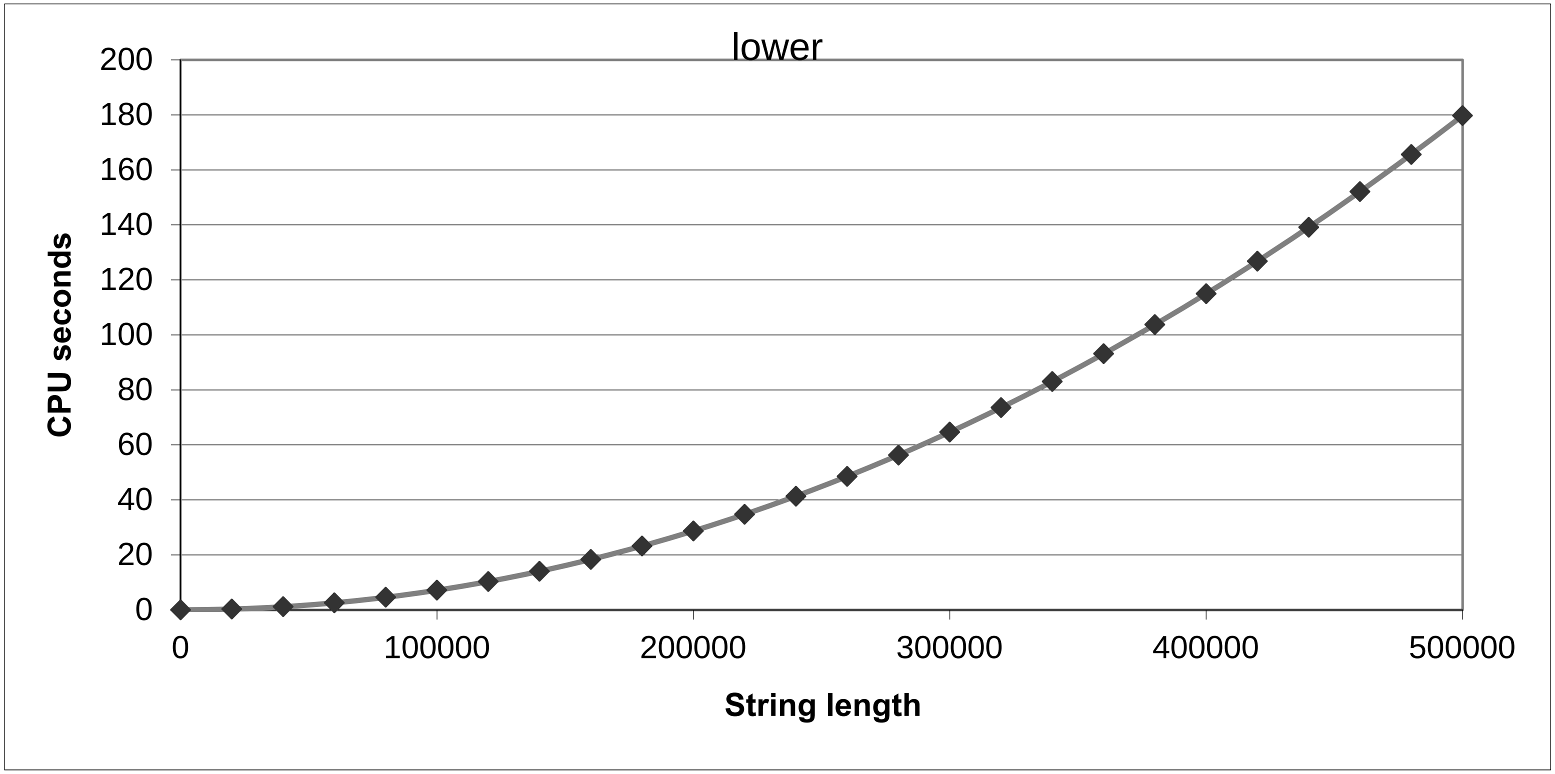
- **Procedure to Convert String to Lower Case**

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

  - Extracted from 213 lab submissions, Fall, 1998

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
   int i = 0;
   if (i >= strlen(s))
     goto done;
 loop:
   if (s[i] >= 'A' && s[i] <= 'Z')
       s[i] -= ('A' - 'a');
   i++;
   if (i < strlen(s))
     goto loop;
 done:
}
```

- `strlen` executed every iteration

Calling Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

- **Strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null character.
- **Overall performance, string of length N**
  - N calls to strlen
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

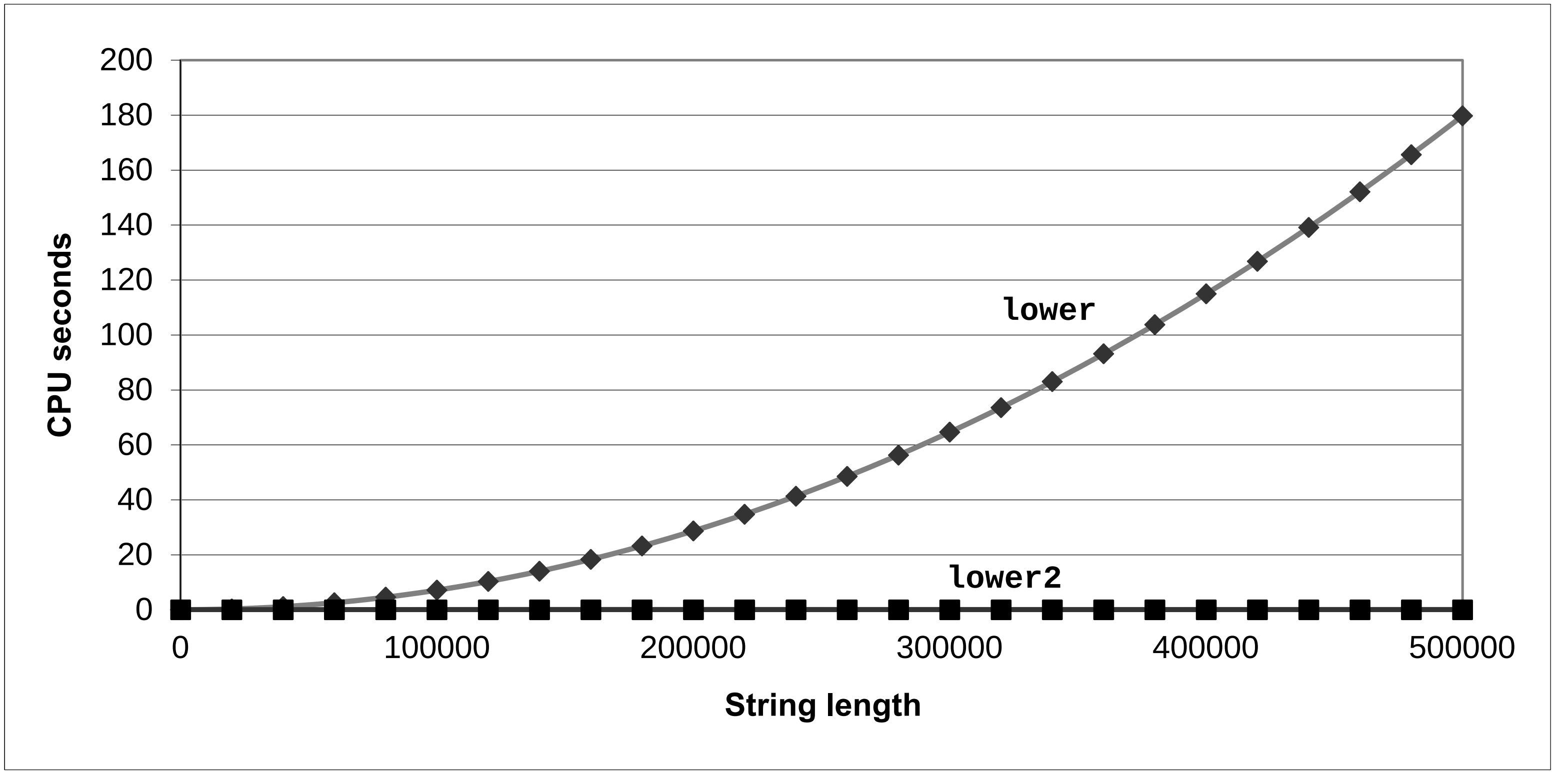
Improving Performance

```
void lower(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');
}
```

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



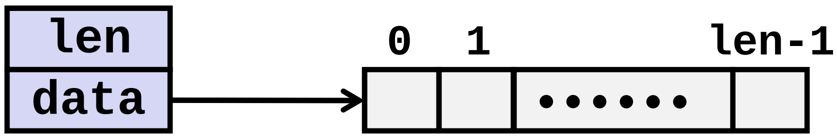
Optimization Blocker: Procedure Calls

- ***Why couldn't compiler move* `strlen` *out of 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`
- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations near them
- **Remedies:**
  - Use of `inline` functions
    - GCC does this with –O2
    - See web aside ASM:OPT
  - Do your own code motion

```
int lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    lencnt += length;
    return length;
}
```

Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    int len;
    double *data;
} vec;
```




```
/* retrieve vector element and store at val */
double get_vec_element(*vec, idx, double *val)
{
     if (idx < 0 || idx >= v->len)
          return 0;
     *val = v->data[idx];
     return 1;
}
```

Benchmark Computation

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

Compute sum or product of vector elements

- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `float`
  - `double`
- **Operations**
  - Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`

Benchmark Performance

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

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	29.0	29.2	27.4	27.9
Combine1 –O1	12.0	12.0	12.0	13.0

Basic Optimizations

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

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

```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

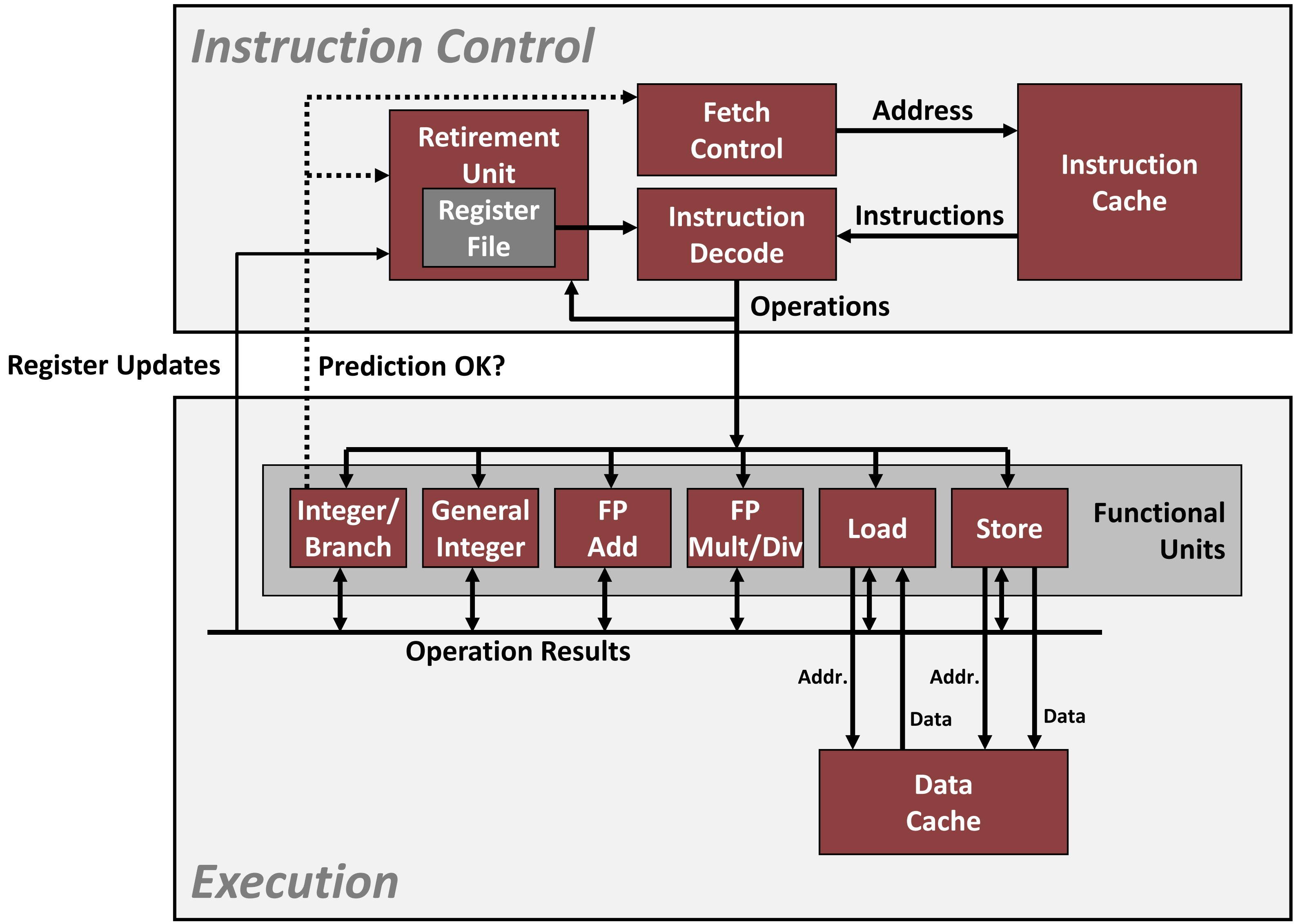
Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	12.0	12.0	12.0	13.0
Combine4	2.0	3.0	3.0	5.0

- Eliminates sources of overhead in loop

Modern CPU Design



Superscalar Processor

- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- **Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have**

- **Most CPUs since about 1998 are superscalar.**
- **Intel: since Pentium Pro**

Nehalem CPU

- **Multiple instructions can execute in parallel**
  - 1 load, with address computation
  - 1 store, with address computation
  - 2 simple integer (one may be branch)
  - 1 complex integer (multiply/divide)
  - 1 FP Multiply
  - 1 FP Add

- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**11--21**	**11--21**
Single/Double FP Multiply	3/5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**10--23**	**10--23**

x86-64 Compilation of Combine4

```
void combine4(vec_ptr v, data_t *dest){
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t * d[i];
  *dest = t;
}
```

- **Inner Loop**

(Case: Integer Multiply)

```
.L519:                          # Loop:
  imull  (%rax,%rdx,4), %ecx    # t = t * d[i]
  addq   $1, %rdx               # i++
  cmpq   %rdx, %rbp             # Compare length:i
  jg     .L519                  # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Latency Bound	1.0	3.0	3.0	5.0

Determine dependencies and critical path

```
for (i = 0; i < length; i++)
    t = t * d[i];
```

Figure 5.13

Graphical representation of inner-loop code for `combine4`. Instructions are dynamically translated into one or two operations, each of which receives values from other operations or from registers and produces values for other operations and for registers. We show the target of the final instruction as the label `loop`. It jumps to the first instruction shown.

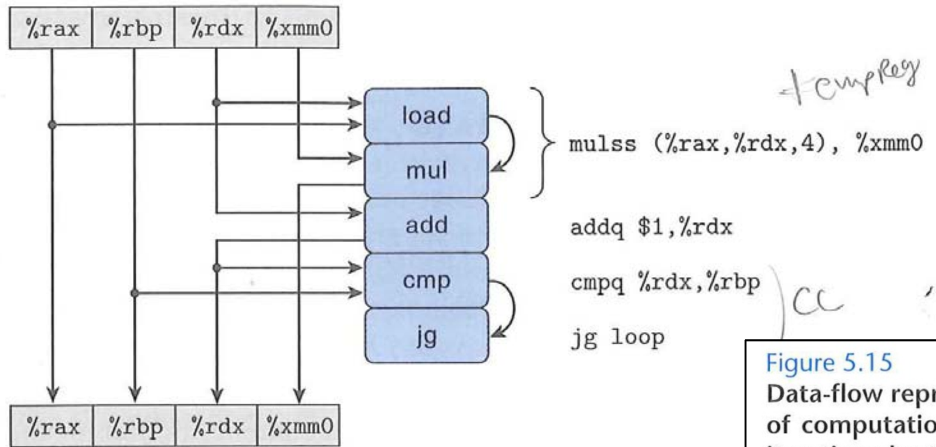


Figure 5.14

Abstracting `combine4` **operations as data-flow graph.** (a) We rearrange the operators of Figure 5.13 to more clearly show the data dependencies, and then (b) show only those operations that use values from one iteration to produce new values for the next.

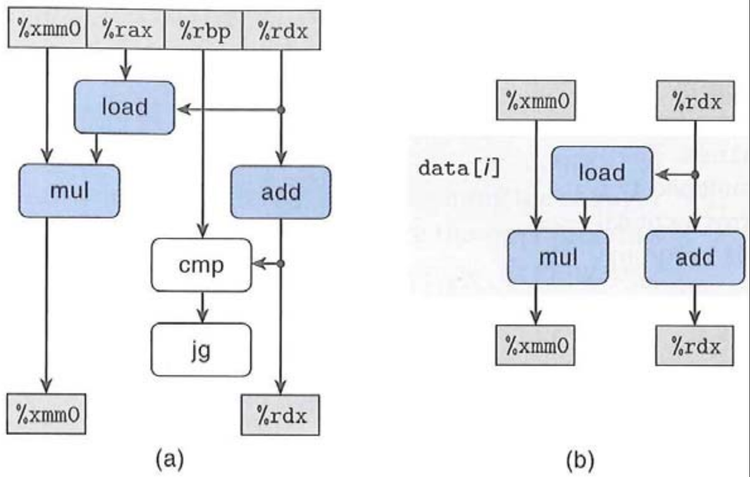
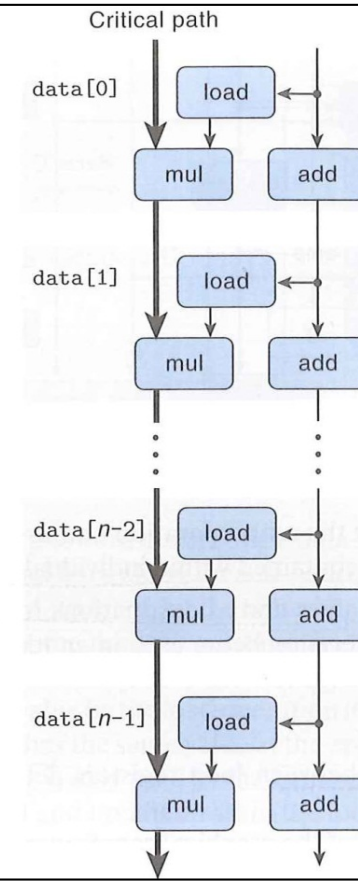


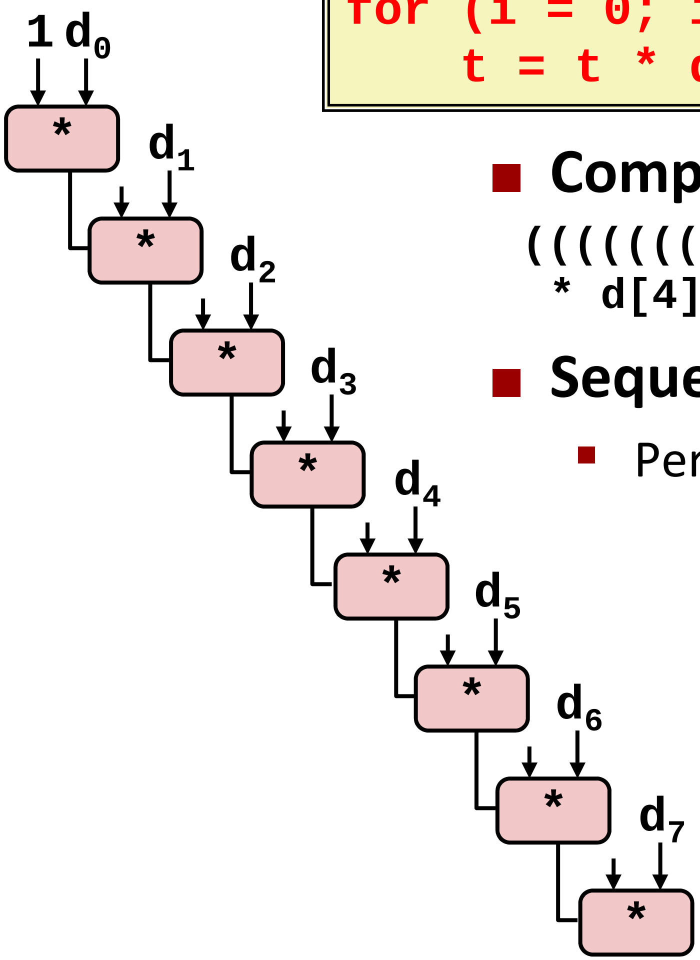
Figure 5.15

Data-flow representation of computation by ***n*** **iterations by the inner loop of** `combine4`. The sequence of multiplication operations forms a critical path that limits program performance.



Combine4 = Serial Computation (OP = *)

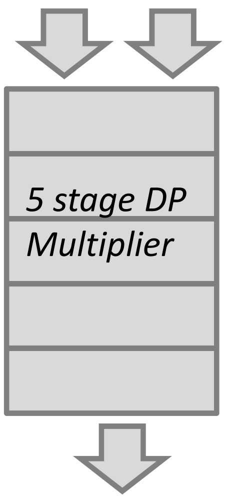
```
t = 1;
for (i = 0; i < 8; i++)
    t = t * d[i];
```



- **Computation (length=8)**

  ((((((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])

- **Sequential dependence**
  - Performance: determined by latency of OP



Loop Unrolling

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- **Perform 2x more useful work per iteration**

```
for (i = 0; i < limit; i+=2)
    x = (x * d[i]) * d[i+1];
```

Figure 5.18
Graphical representation of inner-loop code for combine5. Each iteration has two mulss instructions, each of which is translated into a load and a mul operation.

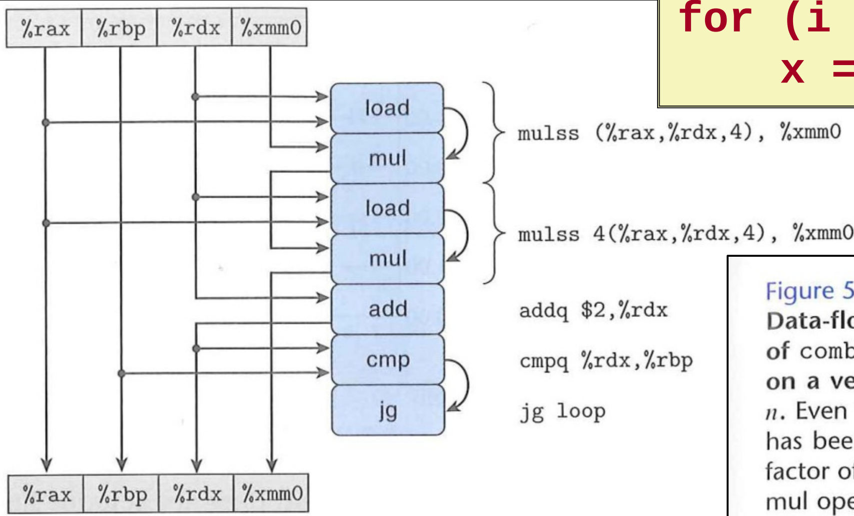


Figure 5.19
Abstracting combine5 **operations as data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.18 to show the data dependencies between successive iterations (a). We see that each iteration must perform two multiplications in sequence (b).

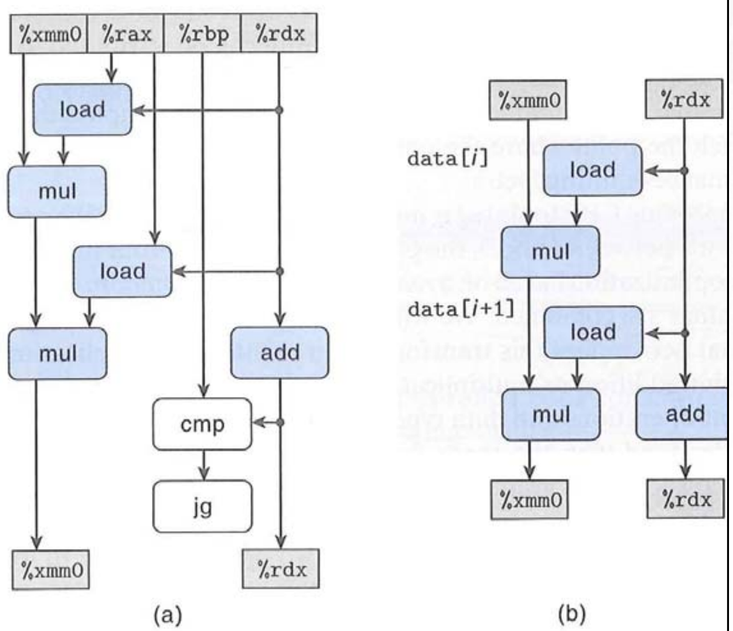
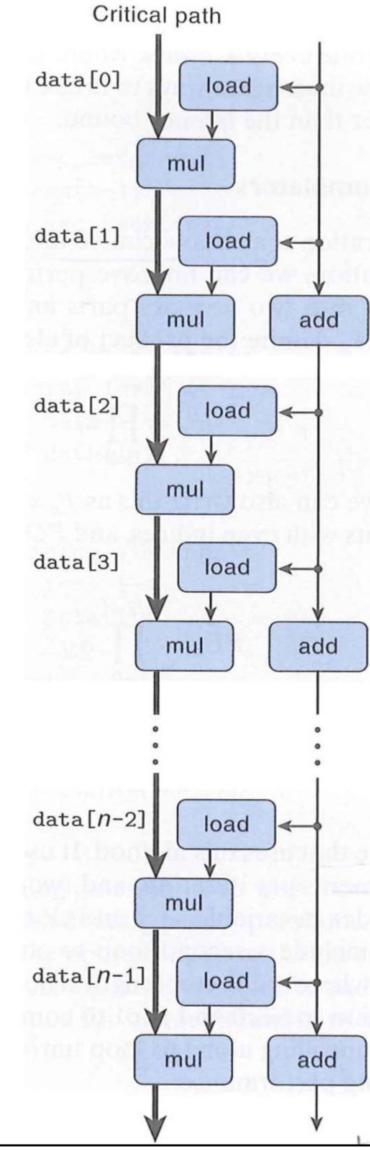


Figure 5.20
Data-flow representation of combine5 **operating on a vector of length** n. Even though the loop has been unrolled by a factor of 2, there are still n mul operations along the critical path.



Effect of Loop Unrolling

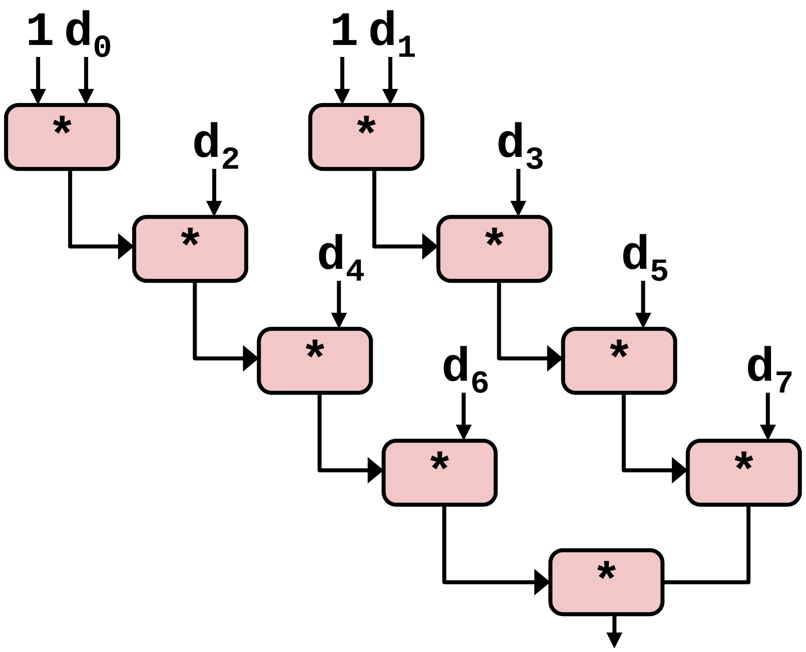
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Latency Bound	1.0	3.0	3.0	5.0

- **Helps integer multiply**
  - below latency bound
  - Compiler does clever optimization
- **Others don't improve. *Why?***
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

New Idea: Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent "streams" of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

Loop Unrolling with Separate Accumulators

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- **Different form of reassociation (see ahead)**

```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x0 = x0 OP d[i];
    x1 = x1 OP d[i+1];
}
```

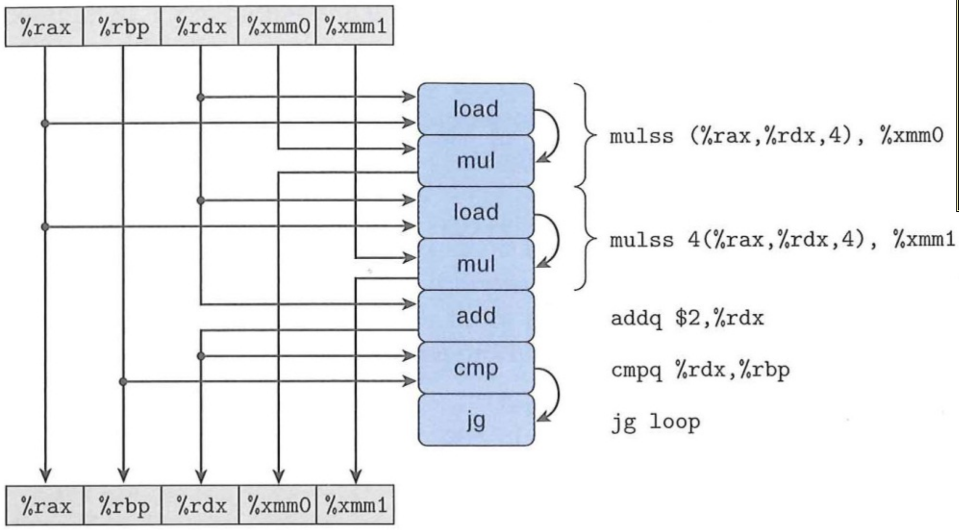


Figure 5.23 **Graphical representation of inner-loop code for combine6.** Each iteration has two mulss instructions, each of which is translated into a load and a mul operation.

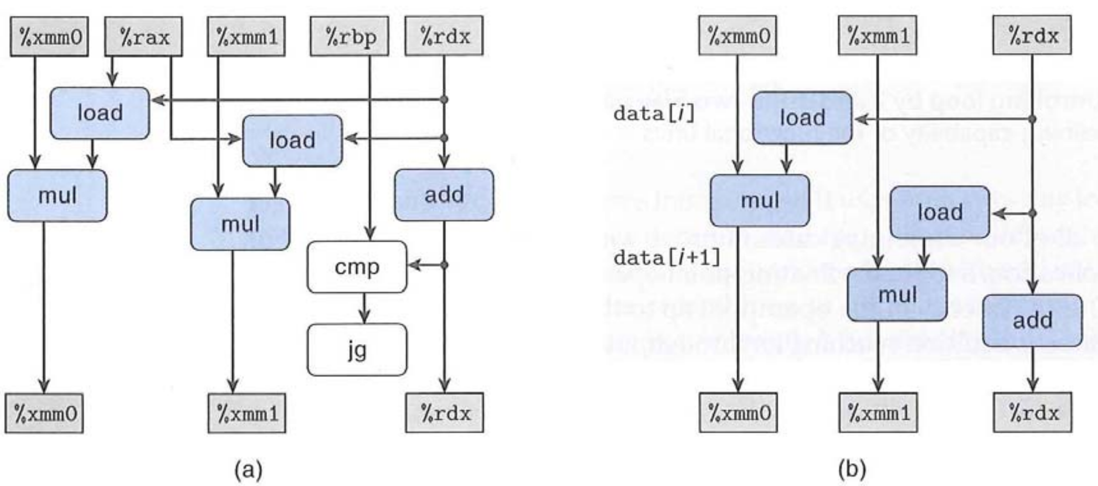
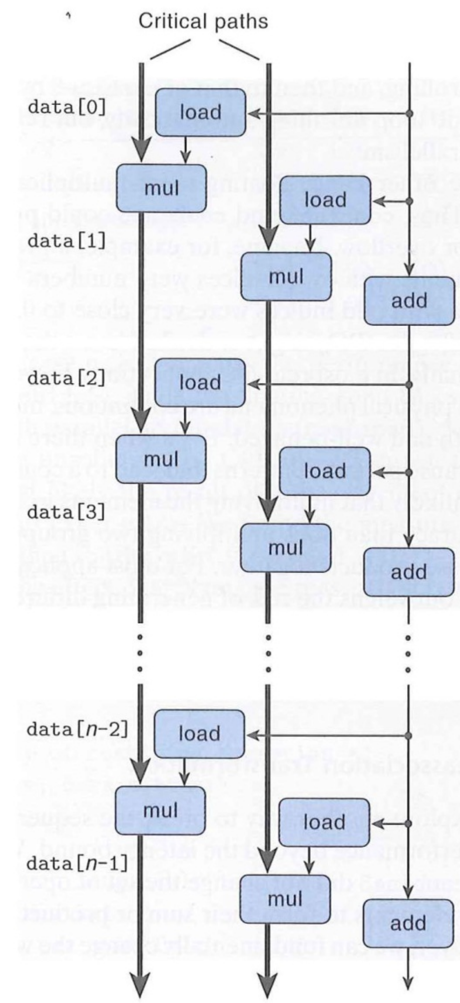


Figure 5.24 **Abstracting combine6 operations as data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.23 to show the data dependencies between successive iterations (a). We see that there is no dependency between the two mul operations (b).

Figure 5.25
Data-flow representation of combine6 operating on a vector of length n. We now have two critical paths, each containing $n/2$ operations.



Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x Parallel 2x	1.5	1.5	1.5	2.5
Latency Bound	1.0	3.0	3.0	5.0
Throughput Bound	1.0	1.0	1.0	1.0

- **2x speedup (over unroll2) for Int *, FP +, FP ***
  - Breaks sequential dependency in a "cleaner," more obvious way

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

Loop Unrolling with Reassociation

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare with previous

```
x = (x OP d[i]) OP d[i+1];
```

- **Can this change the result of the computation?**
- **Yes, for FP. *Why?***

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

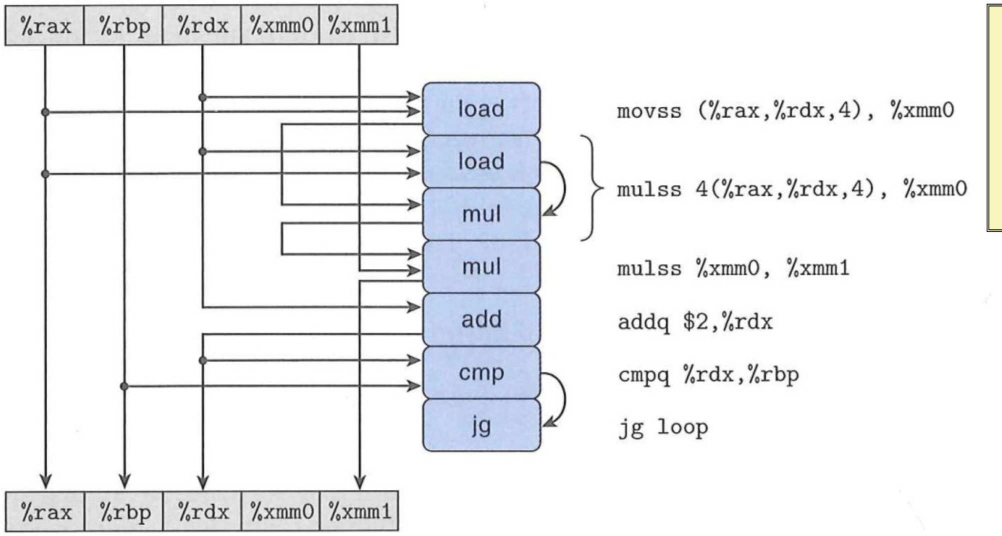


Figure 5.28 **Graphical representation of inner-loop code for** `combine7`. Each iteration gets decoded into similar operations as for `combine5` or `combine6`, but with different data dependencies.

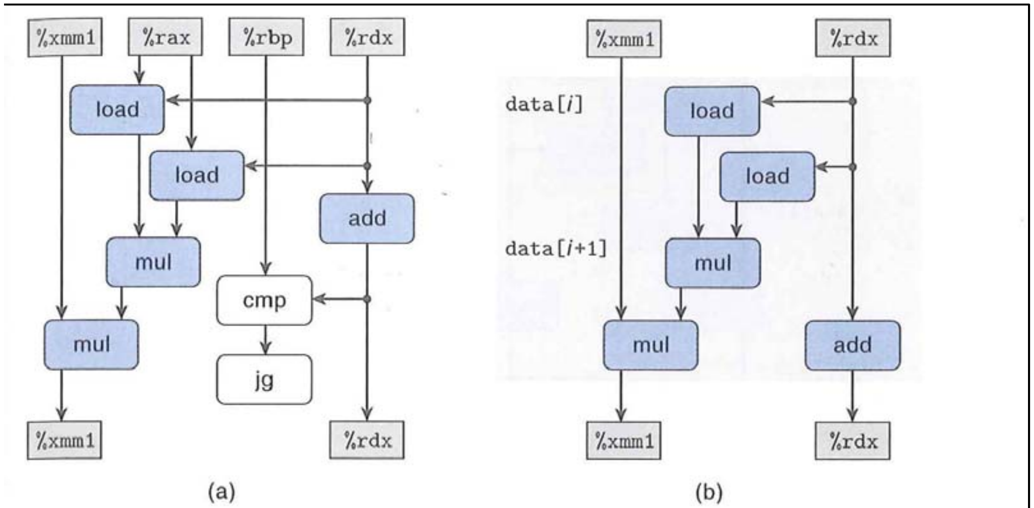
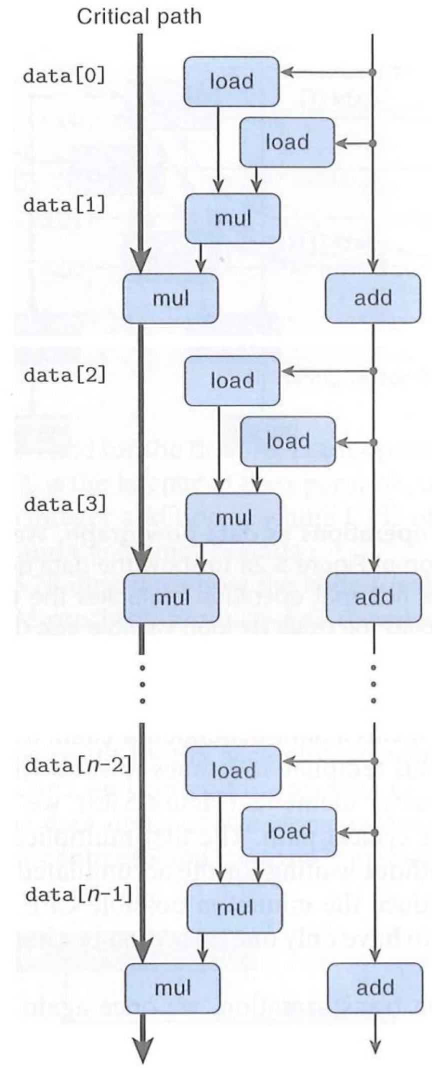


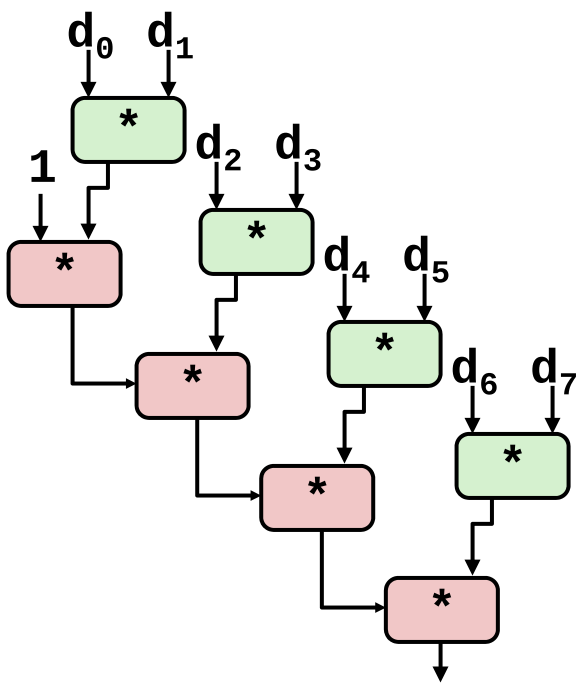
Figure 5.29 **Abstracting** `combine7` **operations as data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.28 to show the data dependencies between successive iterations (a). The first mul operation multiplies the two vector elements, while the second one multiplies the result by loop variable acc (b).

Figure 5.30
Data-flow representation of `combine7` **operating on a vector of length** n.
We have a single critical path, but it contains only $n/2$ operations.



Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



- **What changed:**
  - *Ops in the next iteration can be started early (no dependency)*

- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    CPE = D/2
  - Measured CPE slightly worse for FP mult

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x Parallel 2x	1.5	1.5	1.5	2.5
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Latency Bound	1.0	3.0	3.0	5.0
Throughput Bound	1.0	1.0	1.0	1.0

- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

Unrolling & Accumulating

- **Idea**
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K
- **Limitations**
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

Unrolling & Accumulating: Double *

- **Case**
  - Intel Nehelam
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 1.00

Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.00	5.00	5.00	5.00	5.00	5.00		
2		2.50		2.50		2.50		
3			1.67					
4				1.25		1.25		
6					1.00			1.19
8						1.02		
10							1.01	
12								1.00

Unrolling & Accumulating: Int +

- **Case**
  - Intel Nehelam
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.00	2.00	1.00	1.01	1.02	1.03		
2		1.50		1.26		1.03		
3			1.00					
4				1.00		1.24		
6					1.00			1.02
8						1.03		
10							1.01	
12								1.09

Accumulators

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Optimum	1.00	1.00	1.00	1.00
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	1.00	1.00	1.00	1.00

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code

Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Optimum	1.00	1.00	1.00	1.00
Vector Optimum	0.25	0.53	0.53	0.57
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	1.00	1.00	1.00	1.00
Vec Throughput Bound	0.25	0.50	0.50	0.50

- **Make use of SSE Instructions**
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

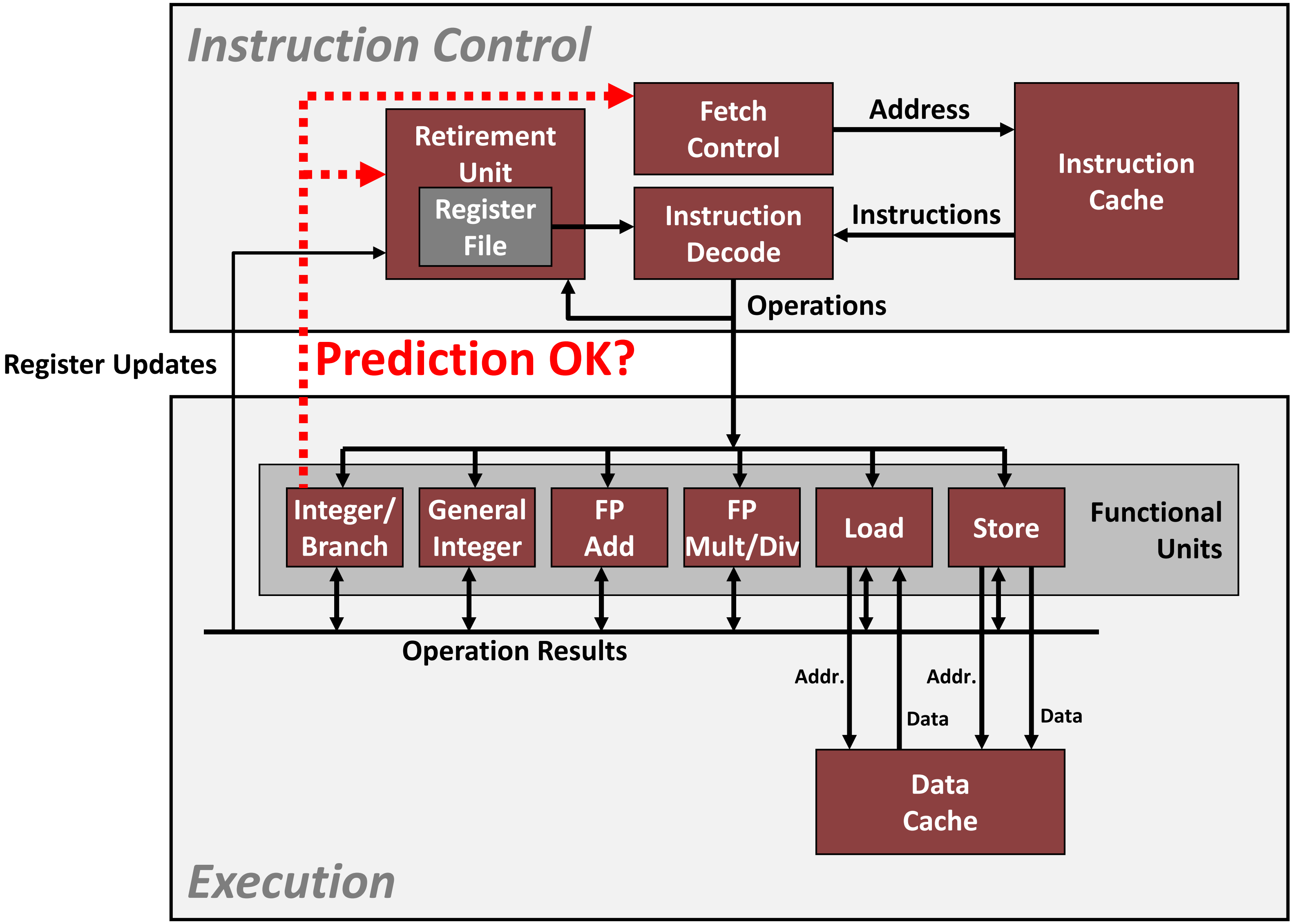
```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
80489fe:  movl   %esi,%esi
8048a00:  imull  (%eax,%edx,4),%ecx
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
80489fe:  movl   %esi,%esi
8048a00:  imull  (%eax,%edx,4),%ecx
```

Branch Not-Taken

Branch Taken

```
8048a25:  cmpl   %edi,%edx
8048a27:  jl     8048a20
8048a29:  movl   0xc(%ebp),%eax
8048a2c:  leal   0xffffffe8(%ebp),%esp
8048a2f:  movl   %ecx,(%eax)
```

Branch Prediction

- **Idea**
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don’t actually modify register or memory data

```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
. . .
```

Predict Taken

```
8048a25:  cmpl   %edi,%edx
8048a27:  jl     8048a20
8048a29:  movl   0xc(%ebp),%eax
8048a2c:  leal   0xffffffe8(%ebp),%esp
8048a2f:  movl   %ecx,(%eax)
```

Begin Execution

Branch Prediction Through Loop

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 98

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 99

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 100

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 101

Assume
vector length = 100

Predict Taken (OK)

Predict Taken
(Oops)

Read
invalid
location

Executed

Fetched

Branch Misprediction Invalidation

Assume

vector length = 100

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 98

Predict Taken (OK)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 99

Predict Taken (Oops)

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~
~~80488b7: cmpl %esi,%edx~~
~~80488b9: jl 80488b1~~

i = 100

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~

i = 101

Invalidate

Branch Misprediction Recovery

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx          i = 99
80488b9:   jl     80488b1
80488bb:   leal   0xffffffe8(%ebp),%esp
80488be:   popl   %ebx
80488bf:   popl   %esi
80488c0:   popl   %edi
```

Definitely not taken

- **Performance Cost**
  - Multiple clock cycles on modern processor
    - 44 clock cycles on the Intel Core i7
    - potentially → hundreds of instructions
  - Can be a major performance limiter

Effect of Branch Prediction

- **Loops**
  - Typically, only miss when hit loop end
- **Checking code**
  - Reliably predicts that error won't occur

```
void combine4b(vec_ptr v,
               data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++) {
      if (i >= 0 && i < v->len) {
          acc = acc OP v->data[i];
      }
    }
    *dest = acc;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Combine4b	4.0	4.0	4.0	5.0

Write Code Suitable for Conditional Moves

Conditional move instructions execute whether or not the condition is met. If not, then write is inhibited.

```
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
int cmovdiff(int x, int y) {
    int tval = y-x;
    int rval = x-y;
    int test = x<y;
    /* line below requires
       single instruction */
    if (test) rval = tval;
    return rval;
}
```

x in %edi

y in %esi

```
cmovdiff:
  movl  %edi, %edx
  subl  %esi, %edx   # tval = x-y
  movl  %esi, %eax
  subl  %edi, %eax   # result = y-x
  cmpl  %esi, %edi   # Compare x:y
  cmovg %edx, %eax   # If >, result = tval
  ret
```

Write Code Suitable for Conditional Moves

Compare two versions of the following code that rearranges two vectors so that for each i, b[i] >= a[i]. Which is faster? Actually data dependent!!

```
void minmax1(int a[], int b[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] > b[i] {
            int t = a[i];
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

```
void minmax2(int a[], int b[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        int min = a[i] < b[i] ? a[i] : b[i];
        int max = a[i] < b[i] ? b[i] : a[i];
        a[i] = min;
        b[i] = max;
    }
}
```

What about memory?

Last week we talked about memory hierarchy. But even in programs where we have successfully dealt with memory, that is, most references are to L1 cache, memory accesses can still be the limiting factor.

Read Access

Example 1: multiple memory references are required for each computation

- See Programming Assignment 2

Example 2: reads/writes follow one another in RAW (true) dependence

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

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

Figure 5.31 **Linked list functions.** Thes

```
  len in %eax, ls in %rdi
.L11:                              loop:
  addl    $1, %eax                 Increment len
  movq    (%rdi), %rdi             ls = ls->next
  testq   %rdi, %rdi               Test ls
  jne     .L11                     If nonnull, goto loop
```

CPE = ???

Getting High Performance

- **Good compiler and flags**
- **Don't do anything stupid**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered last week)