# Chapter 5: Optimization Limits, Code Profiling

**Topics:**

- Limits on Optimization
- Code Profiling
  - Identifying performance bottlenecks

# Announcements

Buffer Lab grading this week

Recitation ex #4 released next Monday, due in a week

- Will release solutions before midterm

Midterm #2 likely Tuesday Nov 11

Performance Lab due Mon Nov 17 by 8 am

Essential that you read the textbook in detail & do the practice problems

- Read Chapter 5, all sections
- Read Chapter 6, all sections

# Recap…

## Improving software performance

- How does combine4 achieves a CPE of 2 ?
  - Need low level assembly
  - How is it mapped to 2 x86 integer functional units?

combine4 combine3 combine2

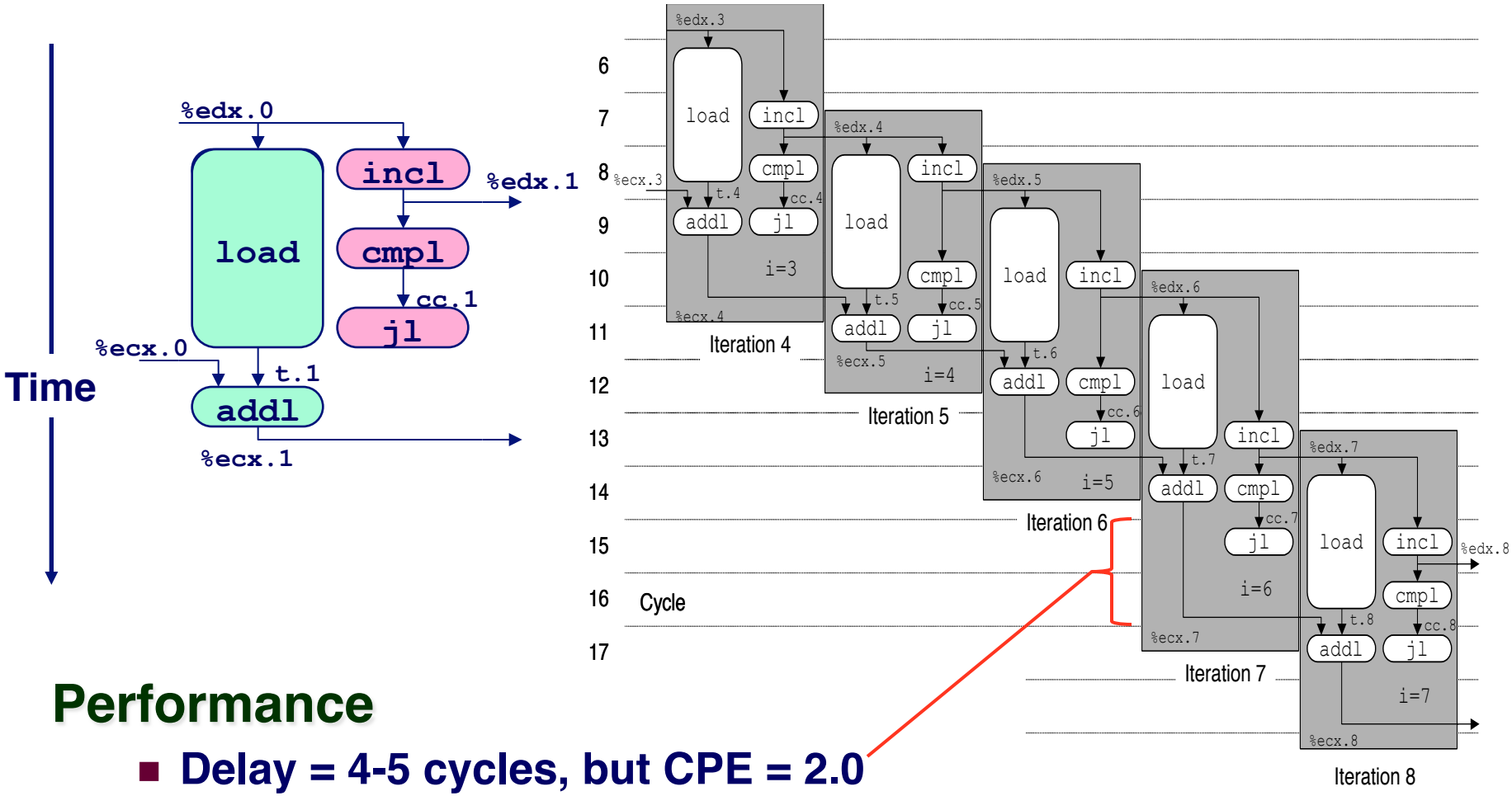
Machine-independent optimizations

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

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

```
load (%eax,%edx.0,4) ➔ t.1
iaddl t.1, %ecx.0    ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.1    ➔ cc.1
jl-taken cc.1
```

# Combine4 Sum over 2 Integer Units




## Performance

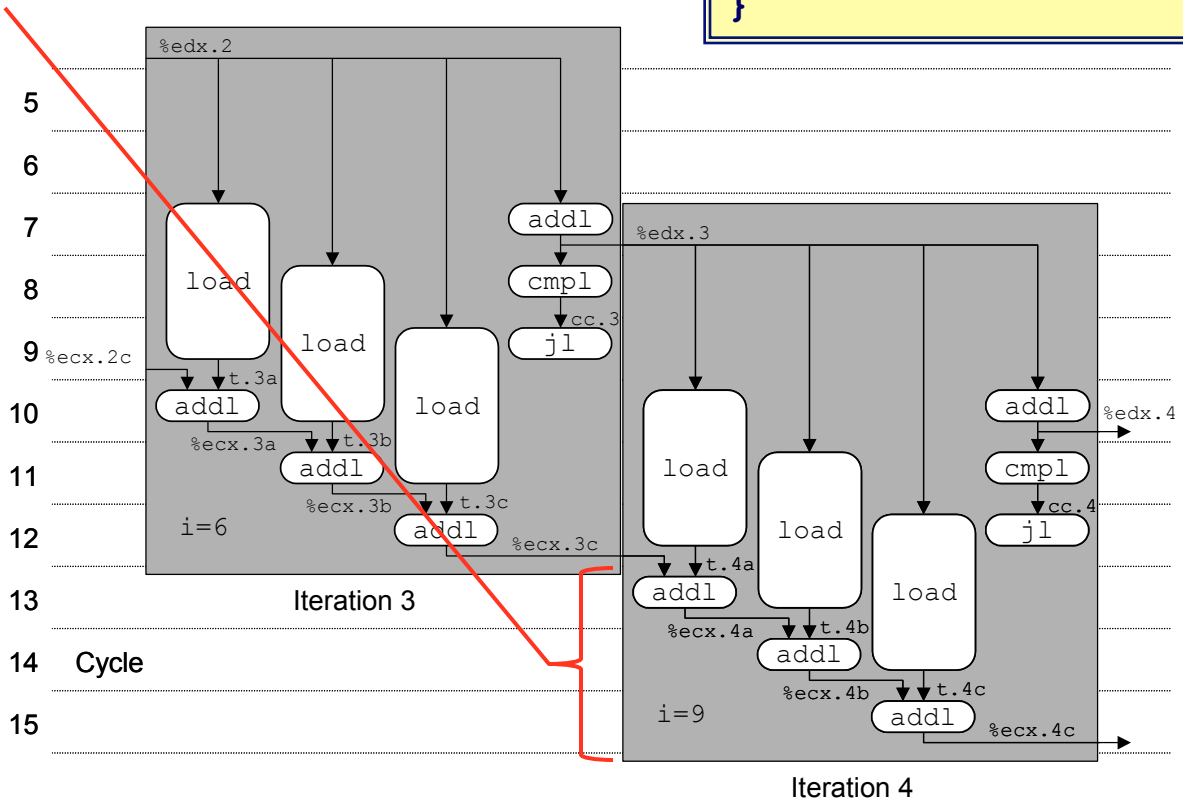
- Delay = 4-5 cycles, but CPE = 2.0

# Recap…

## Improving software performance

- Can we achieve a CPE below 2?
- combine5 unrolls loop and gets CPE ≈ 1, machine-dependent

```
combine5...{
  ...
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
           + data[i+2];
  }
  ...
}
```



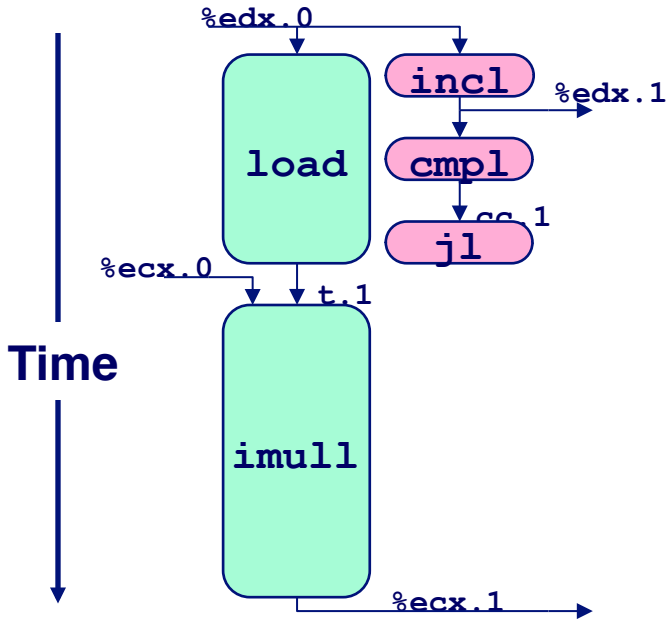
# Recap…

- combine6 uses *both* loop unrolling *and* parallel independent accumulators
- Doesn't help much for addition, but it does help a lot for multiplication

**Loop unrolling with k=3, and Parallel accumulators**

```
combine6...{
  ...
  for (i = 0; i < limit; i+=3) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+1];
    acc2 = acc2 + data[i+2];
  }
  sum = acc0 + acc1 + acc2;
  ...
}
```

**Before Optimization, CPE = 4**



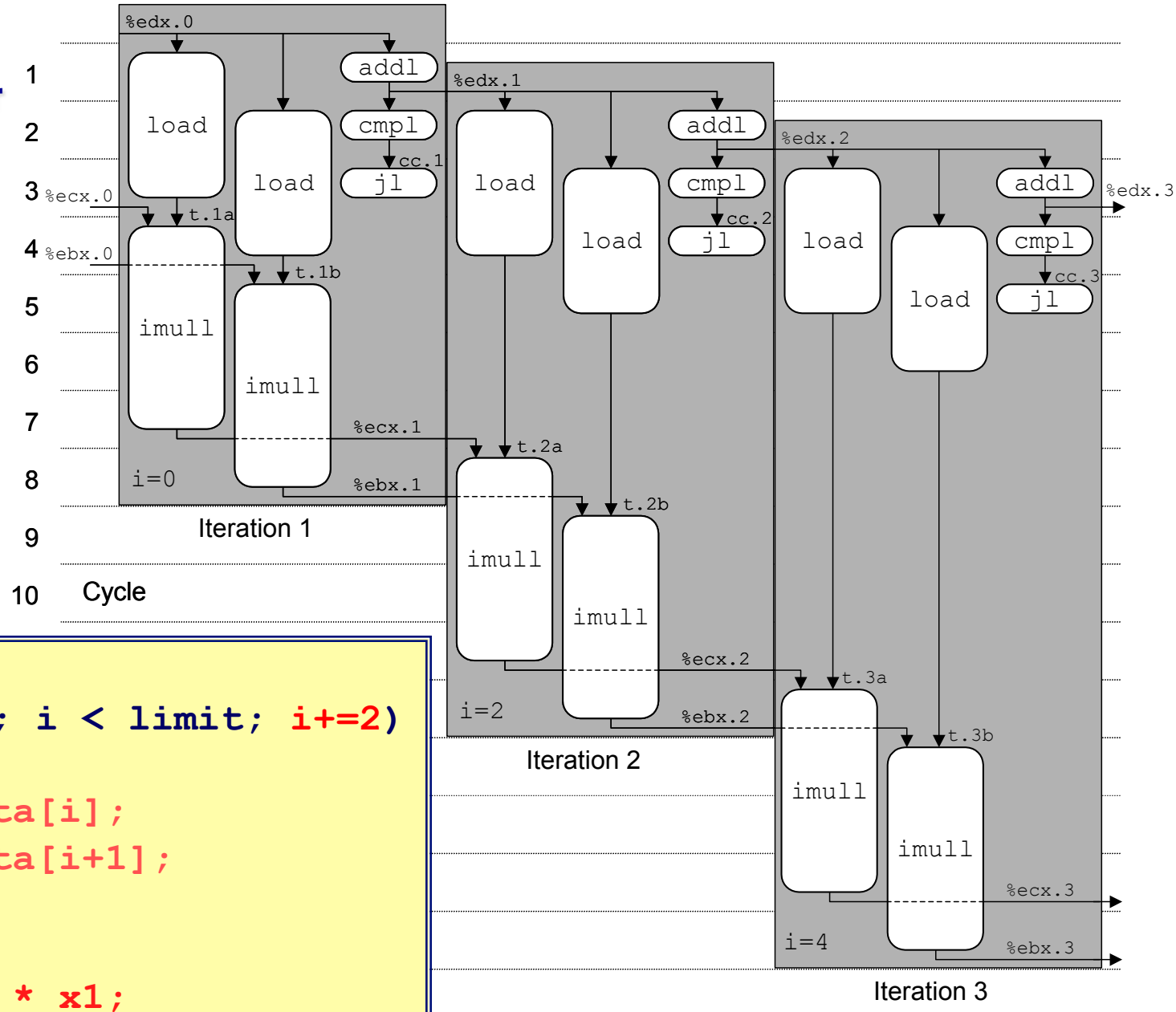

**Before Optimization:**

```
  ...
  for (i = 0; i < limit; i++) {
    x *= data[i];
  }
  ...
  *dest = x;
```

# Loop Unrolling + Parallel Products

**Can keep 4-cycle multiplier busy by pipelining two multiplications**

**After optimization: CPE = 2.0 !**




```
  ...
  for (i = 0; i < limit; i+=2)
{
    x0 *= data[i];
    x1 *= data[i+1];
  }
  ...
  *dest = x0 * x1;
```

# Summary: Results for Pentium III

| Method | Integer | | Floating Point | |
|---|---|---|---|---|
| | + | * | + | * |
| Abstract -g | 42.06 | 41.86 | 41.44 | 160.00 |
| Abstract -O2 | 31.25 | 33.25 | 31.25 | 143.00 |
| Move vec_length | 20.66 | 21.25 | 21.15 | 135.00 |
| Move function call | 6.00 | 9.00 | 8.00 | 117.00 |
| Accum. in temp | 2.00 | 4.00 | 3.00 | 5.00 |
| Unroll 4 | 1.50 | 4.00 | 3.00 | 5.00 |
| Unroll 16 | 1.06 | 4.00 | 3.00 | 5.00 |
| 4 X 2 | 1.50 | 2.00 | *1.50* | 2.50 |
| *8 X 4* | *1.25* | *1.25* | 1.50 | *2.00* |
| 8 X 8 | 1.88 | 1.88 | 1.75 | 2.00 |
| *Worst : Best* | *39.7* | *33.5* | *27.6* | *80.0* |

- Biggest gain doing basic optimizations
- But, last little bit helps

# Limitations of Parallel Execution

We saw in the previous slide that parallelizing to 8x8 actually resulted in worse performance

Need Lots of Registers

- e.g. to hold 8 parallel sums/products
- Only 6 usable integer registers
  - Also needed for pointers, loop conditions
- 8 FP registers
- When not enough registers, must spill temporaries onto stack
  - Wipes out any performance gains
- Not helped by renaming
  - Cannot reference more operands than instruction set allows
  - Major drawback of IA32 instruction set

# Register Spilling Example

## Example

- 8 X 8 integer product, so 8 parallel accumulators for partial products
- 7 local variables share 1 register
- See that we are storing locals on stack
  - e.g., at `-8(%ebp)`
- All this memory manipulation essentially negates any performance gain due to parallelism

```
.L165:
     imull (%eax),%ecx
     movl -4(%ebp),%edi
     imull 4(%eax),%edi
     movl %edi,-4(%ebp)

     movl -8(%ebp),%edi
     imull 8(%eax),%edi
     movl %edi,-8(%ebp)

     movl -12(%ebp),%edi
     imull 12(%eax),%edi
     movl %edi,-12(%ebp)

     …
     addl $32,%eax
     addl $8,%edx
     cmpl -32(%ebp),%edx
     jl .L165
```

# What About Branches?

## Challenge

- Instruction Control Unit must work well ahead of Exec. 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
```


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

# 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   0xfffffff8(%ebp),%esp
8048a2f:  movl   %ecx,(%eax)
```

Execute

# 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*

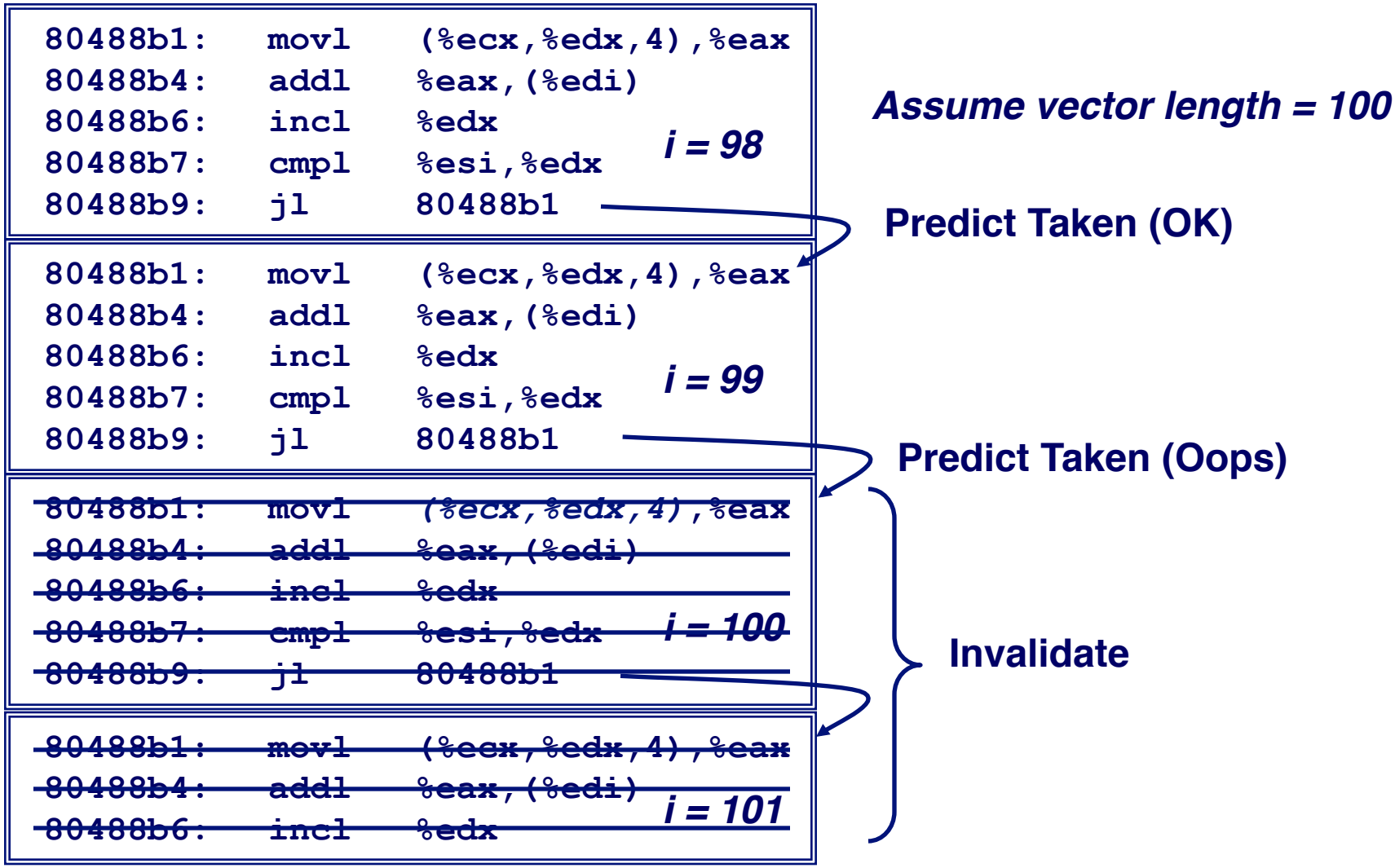
***Assume vector length = 100***

**Predict Taken (OK)**

**Predict Taken (Oops)**

**Read invalid location**

# Branch Misprediction Invalidation



# Branch Misprediction Recovery

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

*i = 98*

*Assume vector length = 100*

Predict Taken (OK)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
80488bb:   leal   0xfffffe8(%ebp),%esp
80488be:   popl   %ebx
80488bf:   popl   %esi
80488c0:   popl   %edi
```

*i = 99*

Definitely not taken

## Performance Cost

- Misprediction on Pentium III wastes ~14 clock cycles
- That's a lot of time on a high performance processor

# Avoiding Branches

## On Modern Processor, Branches Very Expensive

- Unless prediction can be reliable
- When possible, best to avoid altogether

## Example

- Compute maximum of two values
  - 14 cycles when prediction correct
  - 29 cycles when incorrect

```
int max(int x, int y)
{
  return (x < y) ? y : x;
}
```

```
  movl 12(%ebp),%edx  # Get y
  movl 8(%ebp),%eax   # rval=x
  cmpl %edx,%eax      # rval:y
  jge L11             # skip when >=
  movl %edx,%eax      # rval=y
L11:
```

# Conditional Move

- Added with P6 microarchitecture (PentiumPro onward)
- `cmov`*`XX`*`l %edx, %eax`
  - If condition *`xx`* holds, copy `%edx` to `%eax`
  - Doesn't involve any branching – much more predictable
  - Handled as operation within Execution Unit

```
movl 8(%ebp),%edx    # Get x
movl 12(%ebp),%eax   # rval=y
cmpl %edx, %eax      # rval:x
cmovll %edx,%eax     # If <, rval=x
```

- Current version of GCC won't use this instruction
  - Thinks it's compiling for a 386
  - Can turn on use of conditional move by gcc by compiling with optimization flags -O1, -O2, -O3, …
- Performance
  - 14 cycles on all data

# Conditional Move

- **Writing code that is conditional-move-friendly**
  - **For gcc, use conditional operations to compute outcome(s)**
  - **Then assign the result variable(s)**
  - **Example:**

```
1 /* Rearrange two vectors so that for each i, b[i] >=
  a[i] */
2 void minmax2(int a[], int b[], int n) {
3   int i;
4   for (i = 0; i < n; i++) {
5     int min = a[i] < b[i] ? a[i] : b[i];
6     int max = a[i] < b[i] ? b[i] : a[i];
7     a[i] = min;
8     b[i] = max;
9   }
10}
```

# Limitations on Optimization

1. # Functional units of each type

   Don't have unlimited resources

2. How deeply each operation can be pipelined

3. # registers => Register spilling

   If you over-parallelize, there are not enough data registers to hold all the accumulators, so they have to be stored in memory

4. Branch prediction

   must refill pipeline after a misprediction

5. Data dependencies in your graph

   a) slow down pipeline

   b) memory dependencies (next)

6. Amdahl's Law (next)

# Memory Load/Store Bottlenecks (5.12)

x86 supports one Load unit and one Store unit to interact with memory

Load and Store functional units can initiate one load or store each clock cycle

- Latency is 3-4 clock cycles, so can pipeline 3-4 load or store instructions on each unit

Usually, can pipeline loads and stores, except

- A load can depend on a previous load, e.g. ls = ls->next
  - i.e. the second load cannot begin until the first load has finished
- Loads can depend on previous stores to the same memory address, forming a *write/read dependency*
  - i.e. if you write then read from the same memory location, must wait for write to complete before reading

# A Load Can Depend on a Prior Load

**Example (64 bit):**

```
# 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
```

# A Load Can Depend on a Prior Store

**Example:**

```
# what if %ecx = %ebx?
movl %eax,(%ecx)          # Store
movl (%ebx),%eax          # Load
addl $1,%eax
subl $1,%edx
jne loop
```

# if %ecx = %ebx, then you're reading from the same memory location that was just written – the read must wait for the write to complete

CPU creates a store buffer of addresses of stores. A load must check the store buffer to see if there is a dependency before loading

# Amdahl's Law – Know Your Limits!

**The speedup in your program via parallelization is fundamentally limited by the slowest sequential portion of your program**

T = time to execute a program

P = % of program that can be sped up a factor S times

$$\text{New time } T' = P*\frac{T}{S} + (1-P)*T$$

$$\text{Total speedup} = T/T' = \frac{1}{1-P + \frac{P}{S}}$$

As S-> ∞, $T/T' \rightarrow \frac{1}{1-P}$

Ex. P=70%, S=100, Total speedup = 1/(.3 +.007) = 3.25!

# Machine-Dependent Opt. Summary

## Loop Unrolling

- Some compilers do this automatically
- Generally not as clever as what can achieve by hand

## Exposing Instruction-Level Parallelism

- Very machine dependent

## Warning:

- Benefits depend heavily on particular machine
- Best if performed by compiler
  - But GCC on IA32/Linux is not very good
- Do only for performance-critical parts of code

# Important Profiling Tools

## Find the bottleneck…

## Measurement

- Accurately compute time taken by code
  - Most modern machines have built in cycle counters
  - Using them to get reliable measurements is tricky
- Profile procedure calling frequencies
  - Unix tool gprof
- Analyze memory performance, e.g. leaks
  - valgrind

## Observation

- Generating assembly code
  - Lets you see what optimizations compiler can make
  - Understand capabilities/limitations of particular compiler

# Code Profiling Example

## Task

- Count word frequencies in text document
- Produce sorted list of words from most frequent to least

## Steps

1. Convert strings to lowercase
2. Apply hash function
3. Each hash bucket is a linked list.
   Search list,
   - If not found, insert in list
   - If found, increment counter for word
4. Sort results

## Data Set

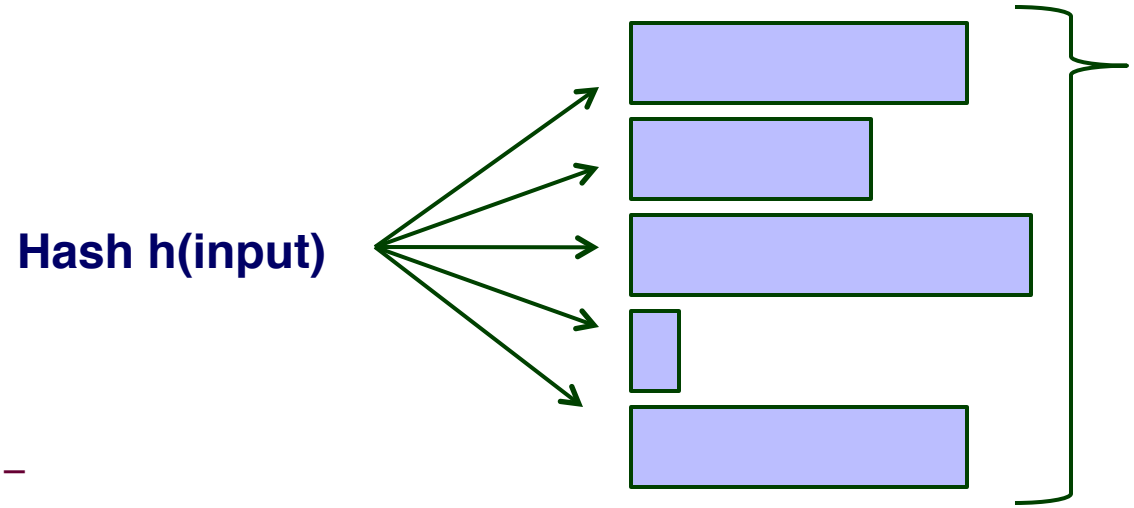
- Collected works of Shakespeare
- 946,596 total words, 26,596 unique
- Initial implementation: 9.2 seconds

**Shakespeare's most frequent words**

| | |
|---|---|
| 29,801 | the |
| 27,529 | and |
| 21,029 | I |
| 20,957 | to |
| 18,514 | of |
| 15,370 | a |
| 14010 | you |
| 12,936 | my |
| 11,722 | in |
| 11,519 | that |

# Hash Function

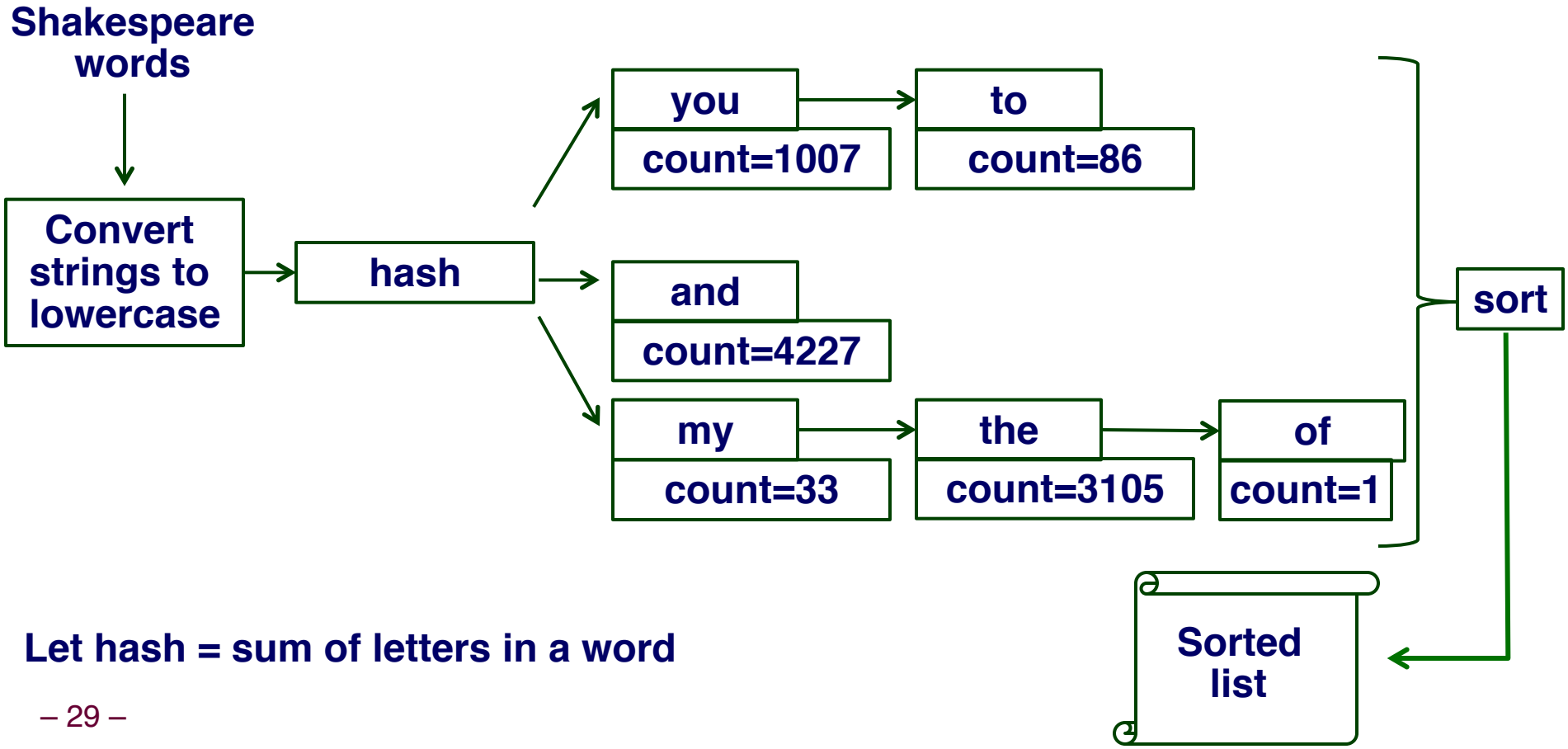
- Takes a set of input values and maps/groups them into N subsets or buckets
- Typically used to quickly classify/separate inputs into one of N possible categories
- Example 1: if set of input values = all English words, then h(word) could classify a word by its first letter, creating 26 buckets, e.g. bucket 'a' contains all words starting with 'a' …
- Example 2: if set of input values = all integers, then h(num) = num mod 4 would classify the input into one of 4 buckets



# Code Profiling Example (cont.)

## Task

- Count word frequencies in text document
- Produce sorted list of words from most frequent to least




Let hash = sum of letters in a word

# Code Profiling

## Augment Executable Program with Timing Functions

- Computes (approximate) amount of time spent in each function
- Time computation method
  - Periodically (~ every 10ms) interrupt program
  - Determine what function is currently executing
  - Increment its timer by interval (e.g., 10ms)
- Also maintains counter for each function indicating number of times called

## Using

```
gcc -O2 -pg prog.c -o prog
```

```
./prog
```

- Executes in normal fashion, but also generates file `gmon.out`

```
gprof prog
```

- Generates profile information based on `gmon.out`

# Profiling Results

| % time | cumulative seconds | self seconds | calls | self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 86.60 | 8.21 | 8.21 | 1 | 8210.00 | 8210.00 | sort_words |
| 5.80 | 8.76 | 0.55 | 946596 | 0.00 | 0.00 | lower1 |
| 4.75 | 9.21 | 0.45 | 946596 | 0.00 | 0.00 | find_ele_rec |
| 1.27 | 9.33 | 0.12 | 946596 | 0.00 | 0.00 | h_add |

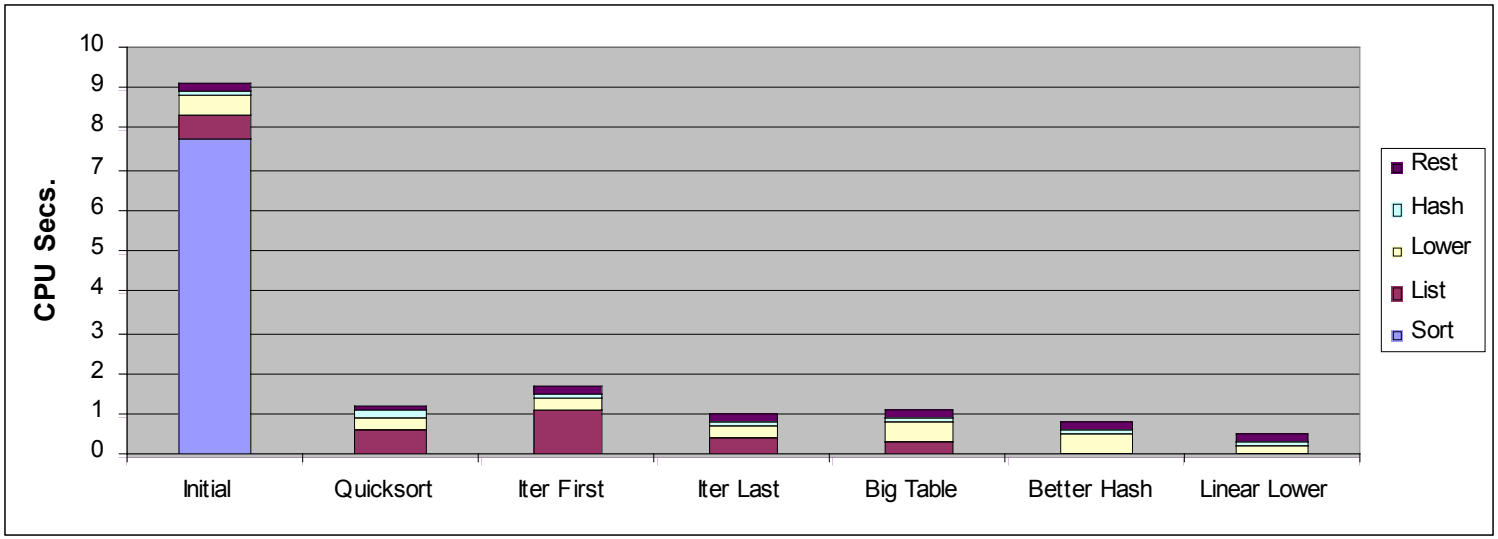
## Call Statistics

- Number of calls and cumulative time for each function

## Performance Limiter

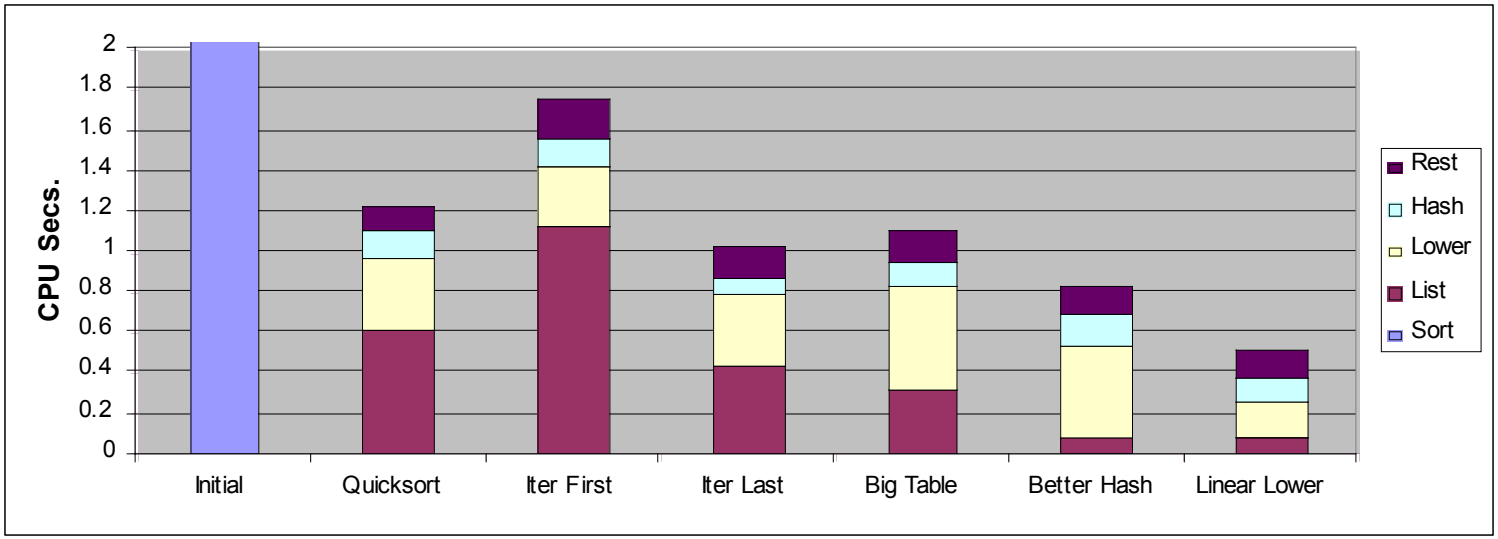
- Using inefficient sorting algorithm
- Single call uses 87% of CPU time

# Code Optimizations




- First step: Use more efficient sorting function
- Library function `qsort`

# Further Optimizations




- Iter first: Use iterative function to insert elements into linked list
  - Causes code to slow down
- Iter last: Iterative function, places new entry at end of list
  - Tend to place most common words at front of list
- Big table: Increase number of hash buckets
- Better hash: spreads hash results more uniformly across buckets, thus reducing length of linked lists
- Linear lower: Move `strlen` out of loop

# Profiling Observations

## Benefits

- Helps identify performance bottlenecks
- Especially useful when have complex system with many components

## Limitations

- Only shows performance for data tested
- e.g., linear lower did not show big gain, since words are short
  - Quadratic inefficiency could remain lurking in code, i.e. it could be revealed when input data is mostly longer word strings
- Timing mechanism fairly crude
  - Only works for programs that run for > 3 seconds

# Role of Programmer

*How should I write my programs, given that I have a good, optimizing compiler?*

## Don't: Completely Sacrifice Modularity for Performance

- Hard to read, maintain, & assure correctness

## Do: Balance Modularity and Performance

- Select best algorithm, e.g. best sorting algorithm
- Write code that's readable & maintainable
  - Procedures, recursion, without built-in constant limits
  - Even though these factors can slow down code
- Eliminate optimization blockers
  - Allows compiler to do its job
- Use profilers to identify bottlenecks

## Focus on Inner Loops

- Do detailed optimizations where code will be executed repeatedly
- Will get most performance gain here

# Supplementary Slides

# Great Reality #4

- ***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

# Translation Example

## Version of Combine4

- Integer data, multiply operation

```
.L24:                              # Loop:
  imull (%eax,%edx,4),%ecx         # t *= data[i]
  incl %edx                        # i++
  cmpl %esi,%edx                   # i:length
  jl .L24                          # if < goto Loop
```

## Translation of First Iteration

```
.L24:

  imull (%eax,%edx,4),%ecx
  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```
load (%eax,%edx.0,4)  ➔ t.1
imull t.1, %ecx.0     ➔ %ecx.1
incl %edx.0           ➔ %edx.1
cmpl %esi, %edx.1     ➔ cc.1
jl-taken cc.1
```

# Translation Example #1

```
imull (%eax,%edx,4),%ecx
```

```
load (%eax,%edx.0,4) ➔ t.1
imull t.1, %ecx.0    ➔ %ecx.1
```

- Split into two operations
  - `load` reads from memory to generate temporary result `t.1`
  - Multiply operation just operates on registers
- Operands
  - Registers `%eax` does not change in loop. Values will be retrieved from register file during decoding
  - Register `%ecx` changes on every iteration. Uniquely identify different versions as `%ecx.0`, `%ecx.1`, `%ecx.2`, …
    - » Register *renaming*
    - » Values passed directly from producer to consumers

# Translation Example #2

```
incl %edx
```

```
incl %edx.0          ➔ %edx.1
```

- Register `%edx` changes on each iteration. Rename as `%edx.0`, `%edx.1`, `%edx.2`, ...

# Translation Example #3

```
cmpl %esi,%edx
```

```
cmpl %esi, %edx.1  ➔ cc.1
```

- Condition codes are treated similar to registers
- Assign tag to define connection between producer and consumer

# Translation Example #4

```
jl .L24
```

```
jl-taken cc.1
```

- Instruction control unit determines destination of jump
- Predicts whether will be taken and target
- Starts fetching instruction at predicted destination
- Execution unit simply checks whether or not prediction was OK
- If not, it signals instruction control
  - Instruction control then “invalidates” any operations generated from misfetched instructions
  - Begins fetching and decoding instructions at correct target

# Results for Alpha Processor

| Method | Integer | | Floating Point | |
|---|---|---|---|---|
| | + | * | + | * |
| Abstract -g | 40.14 | 47.14 | 52.07 | 53.71 |
| Abstract -O2 | 25.08 | 36.05 | 37.37 | 32.02 |
| Move vec_length | 19.19 | 32.18 | 28.73 | 32.73 |
| data access | 6.26 | 12.52 | 13.26 | 13.01 |
| Accum. in temp | 1.76 | 9.01 | 8.08 | 8.01 |
| Unroll 4 | 1.51 | 9.01 | 6.32 | 6.32 |
| Unroll 16 | 1.25 | 9.01 | 6.33 | 6.22 |
| 4 X 2 | 1.19 | 4.69 | 4.44 | 4.45 |
| 8 X 4 | 1.15 | *4.12* | *2.34* | *2.01* |
| 8 X 8 | *1.11* | 4.24 | 2.36 | 2.08 |
| *Worst : Best* | *36.2* | *11.4* | *22.3* | *26.7* |

- Overall trends very similar to those for Pentium III.
- Even though very different architecture and compiler

# Results for Pentium 4

| Method | Integer | | Floating Point | |
|---|---|---|---|---|
| | + | * | + | * |
| Abstract -g | 35.25 | 35.34 | 35.85 | 38.00 |
| Abstract -O2 | 26.52 | 30.26 | 31.55 | 32.00 |
| Move vec_length | 18.00 | 25.71 | 23.36 | 24.25 |
| data access | 3.39 | 31.56 | 27.50 | 28.35 |
| Accum. in temp | 2.00 | 14.00 | 5.00 | 7.00 |
| Unroll 4 | 1.01 | 14.00 | 5.00 | 7.00 |
| Unroll 16 | 1.00 | 14.00 | 5.00 | 7.00 |
| 4 X 2 | 1.02 | 7.00 | 2.63 | 3.50 |
| 8 X 4 | 1.01 | *3.98* | *1.82* | *2.00* |
| 8 X 8 | *1.63* | 4.50 | 2.42 | 2.31 |
| *Worst : Best* | *35.2* | *8.9* | *19.7* | *19.0* |

- Higher latencies (int * = 14, fp + = 5.0, fp * = 7.0)
  - Clock runs at 2.0 GHz
  - Not an improvement over 1.0 GHz P3 for integer *
- Avoids FP multiplication anomaly

# Avoiding Branches with Bit Tricks

- In style of Lab #1
- Use masking rather than conditionals

```
int bmax(int x, int y)
{
  int mask = -(x>y);
  return (mask & x) | (~mask & y);
}
```

- Compiler still uses conditional
  - 16 cycles when predict correctly
  - 32 cycles when mispredict

```
  xorl %edx,%edx      # mask = 0
  movl 8(%ebp),%eax
  movl 12(%ebp),%ecx
  cmpl %ecx,%eax
  jle L13             # skip if x<=y
  movl $-1,%edx       # mask = -1
L13:
```

# Avoiding Branches with Bit Tricks

- Force compiler to generate desired code

```
int bvmax(int x, int y)
{
    volatile int t = (x>y);
    int mask = -t;
    return (mask & x) |
           (~mask & y);
}
```

```
  movl 8(%ebp),%ecx   # Get x
  movl 12(%ebp),%edx  # Get y
  cmpl %edx,%ecx      # x:y
  setg %al            # (x>y)
  movzbl %al,%eax     # Zero extend
  movl %eax,-4(%ebp)  # Save as t
  movl -4(%ebp),%eax  # Retrieve t
```

- `volatile` declaration forces value to be written to memory
  - Compiler must therefore generate code to compute `t`
  - Simplest way is `setg`/`movzbl` combination
- Not very elegant!
  - A hack to get control over compiler
- 22 clock cycles on all data
  - Better than misprediction