15-213

Code Optimization
April 6, 2000

Topics

- Machine-Independent Optimizations
  - Code motion
  - Reduction in strength
  - Common subexpression sharing
- Machine-Dependent Optimizations
  - Pointer code
  - Unrolling
  - Enabling instruction level parallelism
- Advice

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

Optimizing Compilers

Provide efficient mapping of program to machine

- register allocation
- code selection and ordering
- 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 a Fundamental Constraint:

- must not cause any change in program behavior under *any possible condition*
  - often prevents it from making optimizations that would only affect behavior under seemingly bizarre, 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
  - e.g., using an "`int`" in C for what could be an enumerated type

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

Machine-Independent Optimizations

- **Optimizations you 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

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

→

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


Machine-Independent Opts. (Cont.)

Reductions 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 Pentium II or III, integer multiply only requires 4 CPU cycles**

```
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;
}
```

- **Keep data in registers rather than memory**
  - Compilers have trouble making this optimization

Machine-Independent Opts. (Cont.)

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;
```

```
int 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

Important Tools

Measurement

- **Accurately compute time taken by code**
  - Most modern machines have built in cycle counters
- **Profile procedure calling frequencies**
  - Unix tool gprof
- **Custom-built tools**
  - E.g., L4 cache simulator

Observation

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

Optimization Example

```
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;
  }
}
```

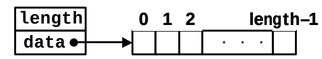
Procedure

- Compute sum of all elements of integer vector
- Store result at destination location
- Vector data structure and operations defined via abstract data type

Pentium II/III Performance: Clock Cycles / Element

- 40.3 (Compiled -g)   28.6 (Compiled -O2)

Vector ADT



Procedures

vec_ptr new_vec(int len)
- Create vector of specified length

int get_vec_element(vec_ptr v, int index, int *dest)
- Retrieve vector element, store at *dest
- Return 0 if out of bounds, 1 if successful

int *get_vec_start(vec_ptr v)
- Return pointer to start of vector data

- Similar to array implementations in Pascal, ML, Java
  - E.g., always do bounds checking

Understanding Loop

```
void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v))
      goto done;
  loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
      goto loop
  done:
}
```

1 iteration

Inefficiency

- Procedure vec_length called every iteration
- Even though result always the same

Move `vec_length` Call Out of Loop

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int len = vec_length(v);
  *dest = 0;
  for (i = 0; i < len; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

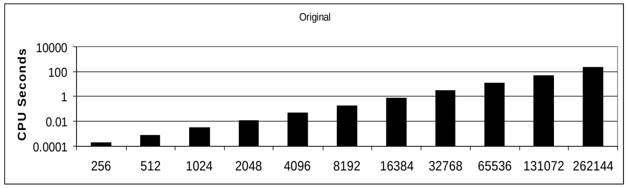
Optimization

- Move call to vec_length out of inner loop
  - Value does not change from one iteration to next
  - Code motion
- CPE: 20.2 (Compiled -O2)
  - **vec_length** requires only constant time, but significant overhead

Code Motion Example #2

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');
}
```



CPU time quadruples every time double string length

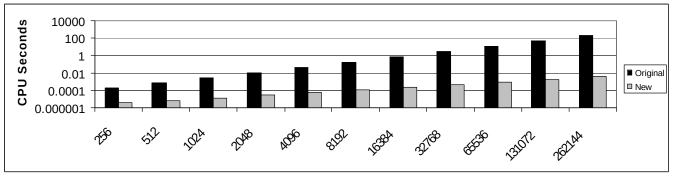
Convert Loop To Goto Form

```
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**
- `strlen` **linear in length of string**
  - Must scan string until finds '\0'
- **Overall performance is quadratic**

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');
}
```



CPU time doubles every time double string length

Optimization Blocker: Procedure Calls

Why couldn't the compiler move `vec_len` ***or*** `strlen` ***out of the inner loop?***

- **Procedure May Have Side Effects**
  - i.e, 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`

Why doesn't compiler look at code for `vec_len` ***or*** `strlen`***?***

- **Linker may overload with different version**
  - Unless declared static
- **Interprocedural optimization is not used extensively due to cost**

Warning:

- **Compiler treats procedure call as a black box**
- **Weak optimizations in and around them**

Reduction in Strength

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int len = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

Optimization

- **Avoid procedure call to retrieve each vector element**
  - Get pointer to start of array before loop
  - Within loop just do pointer reference
  - Not as clean in terms of data abstraction
- **CPE: 6.76 (Compiled -O2)**
  - Procedure calls are expensive!
  - Bounds checking is expensive

Eliminate Unneeded Memory References

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

Optimization

- **Don't need to store in destination until end**
- **Local variable `sum` held in register**
- **Avoids 1 memory read, 1 memory write per cycle**
- **CPE: 3.06 (Compiled -O2)**
  - Memory references are expensive!

Optimization Blocker: Memory Aliasing

Aliasing

- **Two different memory references specify single location**

Example

- **`v: [3, 2, 17]`**
- **`combine3(v, get_vec_start(v)+2) --> ?`**
- **`combine4(v, get_vec_start(v)+2) --> ?`**

Observations

- **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

Machine-Independent Opt. Summary

Code Motion

- *compilers are not very good at this, especially with procedure calls*

Reduction in Strength

- **Shift, add instead of multiply or divide**
  - *compilers are (generally) good at this*
  - *Exact trade-offs machine-dependent*
- **Keep data in registers rather than memory**
  - *compilers are not good at this, since concerned with aliasing*

Share Common Subexpressions

- *compilers have limited algebraic reasoning capabilities*

Machine-Dependent Optimizations

Pointer Code

- A bad idea with a good compiler
- But may be more efficient than array references if you have a not-so-great compiler

Loop Unrolling

- Combine bodies of several iterations
- Optimize across iteration boundaries
- Amortize loop overhead
- Improve code scheduling

Enabling Instruction Level Parallelism

- Making it possible to execute multiple instructions concurrently

Warning:

- Benefits depend heavily on particular machine
- Best if performed by compiler
  - But sometimes you are stuck with a mediocre compiler

Pointer Code

```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int *dend = data+length;
  int sum = 0;
  while (data != dend) {
    sum += *data;
    data++;
  }
  *dest = sum;
}
```

Optimization

- Use pointers rather than array references
- GCC generates code with 1 less instruction in inner loop
- CPE: 2.06 (Compiled -O2)
  - Less work to do on each iteration

Warning: **Good compilers do a better job optimizing array code!!!**

Pointer vs. Array Code Inner Loops

```
L23:
  addl (%eax),%ecx
  addl $4,%eax
  incl %edx      # i++
  cmpl %esi,%edx # i < n?
  jl L23
```

```
L28:
  addl (%eax),%edx
  addl $4,%eax
  cmpl %ecx,%eax # data == dend?
  jne L28
```

Array Code

- GCC does partial conversion to pointer code
- Still keeps variable i
  - To test loop condition

Pointer Code

- Loop condition based on pointer value

Performance

- Array Code: 5 instructions in 3 clock cycles
- Pointer Code: 4 instructions in 2 clock cycles

Pentium II/III CPU Design

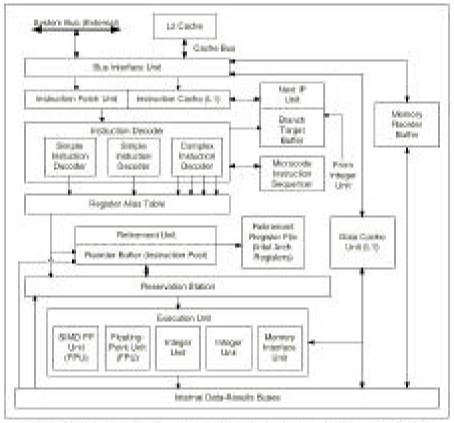


Figure 2-2. Functional Block Diagram of the P6 Family Processor Microarchitecture

Intel Architecture Software Developer's Manual Vol. 1: Basic Architecture

CPU Capabilities

Multiple Instructions Can Execute in Parallel

- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP

Some Instructions Take > 1 Cycle, But Can Be Pipelined

Instruction	Latency	Cycles/Issue
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

Loop Unrolling

```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int *dend = data+length-7;
  int sum = 0;
  while (data < dend) {
    sum += data[0]; sum += data[1];
    sum += data[2]; sum += data[3];
    sum += data[4]; sum += data[5];
    sum += data[6]; sum += data[7];
    data += 8;
  }
  dend += 7;
  while (data < dend) {
    sum += *data; data ++;
  }
  *dest = sum;
}
```

Optimization

- Combine multiple iterations into single loop body
- Amortizes loop overhead across multiple iterations
- CPE = 1.43
  - Only small savings in this case
- Finish extras at end

Loop Unrolling Assembly

```
L33:
  addl (%eax),%edx     # data[0]
  addl -20(%ecx),%edx  # data[1]
  addl -16(%ecx),%edx  # data[2]
  addl -12(%ecx),%edx  # data[3]
  addl -8(%ecx),%edx   # data[4]
  addl -4(%ecx),%edx   # data[5]
  addl (%ecx),%edx     # data[6]
  addl (%ebx),%edx     # data[7]
  addl $32,%ecx
  addl $32,%ebx
  addl $32,%eax
  cmpl %esi,%eax
  jb L33
```

Strange "Optimization"

%eax = data

%ebx = %eax+28

%ecx = %eax+24

- Wasteful to maintain 3 pointers when 1 would suffice

General Forms of Combining

```
void abstract_combine(vec_ptr v, data_t *dest)
{
  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;
  }
}
```

Data Types

- Use different declarations for data_t
- Int
- Float
- Double

Operations

- Use different definitions of OP and IDENT
- + / 0
- * / 1

Optimization Results for Combining

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	40.26	43.52	41.70	146.61
Abstract -O2	28.61	31.12	29.38	139.41
Move vec_length	20.22	20.32	20.48	133.23
data access	6.76	9.06	8.06	110.66
Accum. in temp	3.06	4.09	3.20	5.20
Pointer	2.06	4.06	3.06	5.20
Unroll 8	1.43	4.06	3.06	5.19
Worst : Best	*28.15*	*10.72*	*13.63*	*28.25*

- Double & Single precision FP give identical timings
- Up against latency limits

Integer	Add: 1	Multiply: 4
FP	Add: 3	Multiply: 5

Particular data used had lots of overflow conditions, causing fp store to run very slowly

Parallel Loop Unrolling

```
void combine7(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int *dend = data+length-7;
  int sum1 = 0, sum2 = 0;
  while (data < dend) {
    sum1 += data[0]; sum2 += data[1];
    sum1 += data[2]; sum2 += data[3];
    sum1 += data[4]; sum2 += data[5];
    sum1 += data[6]; sum2 += data[7];
    data += 8;
  }
  dend += 7;
  while (data < dend) {
    sum1 += *data;  data ++;
  }
  *dest = sum1+sum2;
}
```

Optimization

- Accumulate in two different sums
  - Can be performed simultaneously
- Combine at end
- Exploits property that integer addition & multiplication are associative & commutative
- FP addition & multiplication not associative, but transformation usually acceptable

Parallel Loop Unrolling Assembly

```
L43:
  addl (%eax),%ebx       # data[0], sum1
  addl -20(%edx),%ecx    # data[1], sum2
  addl -16(%edx),%ebx    # data[2], sum1
  addl -12(%edx),%ecx    # data[3], sum2
  addl -8(%edx),%ebx     # data[4], sum1
  addl -4(%edx),%ecx     # data[5], sum2
  addl (%edx),%ebx       # data[6], sum1
  addl (%esi),%ecx       # data[7], sum2
  addl $32,%edx
  addl $32,%esi
  addl $32,%eax
  cmpl %edi,%eax
  jb L43
```

Registers

%eax = data   %esi = %eax+28   %edx = %eax+24

%ebx = sum1   %ecx = sum2

- Wasteful to maintain 3 pointers when 1 would suffice

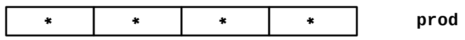
Optimization Results for Combining

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	40.26	43.52	41.70	146.61
Abstract -O2	28.61	31.12	29.38	139.41
Move vec_length	20.22	20.32	20.48	133.23
data access	6.76	9.06	8.06	110.66
Accum. in temp	3.06	4.09	3.20	5.20
Pointer	2.06	4.06	3.06	5.20
Unroll 8	1.43	4.06	3.06	5.19
8 X 2	1.31	2.06	*1.56*	2.70
8 X 4	1.71	1.96	1.93	*2.20*
8 X 8	2.32	2.35	2.13	2.44
9 X 3	*1.19*	*1.42*	*1.45*	2.44
Worst : Best	*33.83*	*30.65*	*28.76*	*66.64*

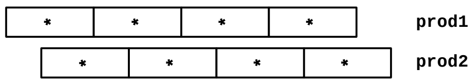
Parallel/Pipelined Operation

FP Multiply Computation

- **5 cycle latency, 2 cycles / issue**
- **Accumulate single product**



  - Effective computation time: 5 cycles / operation
- **Accumulate two products**

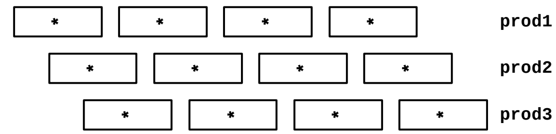


  - Effective computation time: 2.5 cycles / operation

Parallel/Pipelined Operation (Cont.)

FP Multiply Computation

- **Accumulate 3 products**
  - Effective computation time: 2 cycles / operation
  - Limited by issue rate



- **Accumulate > 3 products**
  - Can't go beyond 2 cycles / operation

Limitations of Parallel Execution

Need Lots of Registers

- **To hold sums/products**
- **Only 6 useable 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

Example

- **X 4 integer multiply**
- **4 local variables must share 2 registers**

```
L53: imull (%eax),%ecx
     imull -20(%edx),%ebx
     movl -36(%ebp),%edi
     imull -16(%edx),%edi
     movl -20(%ebp),%esi
     imull -12(%edx),%esi
     imull (%edx),%edi
     imull -8(%edx),%ecx
     movl %edi,-36(%ebp)
     movl -8(%ebp),%edi
     imull (%edi),%esi
     imull -4(%edx),%ebx
     addl $32,%eax
     addl $32,%edx
     addl $32,%edi
     movl %edi,-8(%ebp)
     movl %esi,-20(%ebp)
     cmpl -4(%ebp),%eax
     jb L53
```


Machine-Dependent Opt. Summary

Pointer Code

- **Look carefully at generated code to see whether helpful**

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 particularly bad
- **Do only for performance critical parts of code**

Role of Programmer

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

Don't: Smash Code into Oblivion

- **Hard to read, maintain, & assure correctness**

Do:

- **Select best 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

Focus on Inner Loops

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