Supercomputing in Plain English

Stupid Compiler Tricks

Henry Neeman

Director

OU Supercomputing Center for Education & Research

October 15 2004

Outline

- Dependency Analysis
  - What is Dependency Analysis?
  - Control Dependencies
  - Data Dependencies
- Stupid Compiler Tricks
  - Tricks the Compiler Plays
  - Tricks You Play With the Compiler
  - Profiling

Dependency Analysis

What Is Dependency Analysis?

Dependency analysis describes of how different parts of a program affect one another, and how various parts require other parts in order to operate correctly.

A ***control dependency*** governs how different sequences of instructions affect each other.

A ***data dependency*** governs how different pieces of data affect each other.

Much of this discussion is from references [1] and [5].

Control Dependencies

Every program has a well-defined ***flow of control*** that moves from instruction to instruction to instruction.

This flow can be affected by several kinds of operations:

- Loops
- Branches (if, select case/switch)
- Function/subroutine calls
- I/O (typically implemented as calls)

Dependencies affect **parallelization**!

Branch Dependency

```
y = 7
IF (x /= 0) THEN
    y = 1.0 / x
END IF
```

The value of **y** depends on what the condition **(x /= 0)** evaluates to:

- If the condition evaluates to **.TRUE.**, then **y** is set to **1.0/x**.
- Otherwise, **y** remains **7**.

Loop Carried Dependency

```
DO i = 2, length
  a(i) = a(i-1) + b(i)
END DO
```

Here, each iteration of the loop depends on the previous iteration. That is, the iteration **`i=3`** depends on iteration **`i=2`**, iteration **`i=4`** depends on **`i=3`**, iteration **`i=5`** depends on **`i=4`**, etc.

This is sometimes called a ***loop carried dependency***.

There is no way to execute iteration **`i`** until after iteration **`i-1`** has completed, so this loop can’t be parallelized.

Why Do We Care?

Loops are the favorite control structures of High Performance Computing, because compilers know how to ***optimize*** their performance using instruction-level parallelism: superscalar, pipelining and vectorization can give excellent speedup.

Loop carried dependencies affect whether a loop can be parallelized, and how much.

Loop or Branch Dependency?

Is this a **loop carried dependency** or a **branch dependency**?

```
DO i = 1, length
  IF (x(i) /= 0) THEN
    y(i) = 1.0 / x(i)
  END IF
END DO
```

Call Dependency Example

```
x = 5
y = myfunction(7)
z = 22
```

The flow of the program is interrupted by the **call** to **`myfunction`**, which takes the execution to somewhere else in the program.

It’s similar to a branch dependency.

I/O Dependency

```
X = a + b
PRINT *, x
Y = c + d
```

Typically, I/O is implemented by hidden subroutine calls, so we can think of this as equivalent to a call dependency.

Reductions Aren’t Dependencies

```
array_sum = 0
DO i = 1, length
  array_sum = array_sum + array(i)
END DO
```

Other kinds of reductions: product, **.AND.**, **.OR.**, minimum, maximum, index of minimum, index of maximum, number of occurrences of a particular value, etc.

Reductions are so common that hardware and compilers are optimized to handle them.

Also, they aren’t really dependencies, because the order in which the individual operations are performed doesn’t matter.

Data Dependencies

"A data dependence occurs when an instruction is dependent on data from a previous instruction and therefore cannot be moved before the earlier instruction [or executed in parallel]." [6]

```
a = x + y + cos(z);
b = a * c;
```

The value of **b** depends on the value of **a**, so these two statements **must** be executed in order.

Output Dependencies

```
x = a / b;
y = x + 2;
x = d – e;
```

Notice that **x** is assigned **two different values**, but only one of them is retained after these statements are done executing. In this context, the final value of **x** is the "output."

Again, we are forced to execute in order.

Why Does Order Matter?

- Dependencies can affect whether we can execute a particular part of the program in **parallel**.
- If we cannot execute that part of the program in parallel, then it'll be **SLOW**.

Loop Dependency Example

```
if ((dst == src1) && (dst == src2)) {
  for (index = 1; index < length; index++) {
    dst[index] = dst[index-1] + dst[index];
  }
}
else if (dst == src1) {
  for (index = 1; index < length; index++) {
    dst[index] = dst[index-1] + src2[index];
  }
}
else if (dst == src2) {
  for (index = 1; index < length; index++) {
    dst[index] = src1[index-1] + dst[index];
  }
}
else if (src1 == src2) {
  for (index = 1; index < length; index++) {
    dst[index = src1[index-1] + src1[index];
  }
}
else {
  for (index = 1; index < length; index++) {
    dst[index] = src1[index-1] + src2[index];
  }
}
```

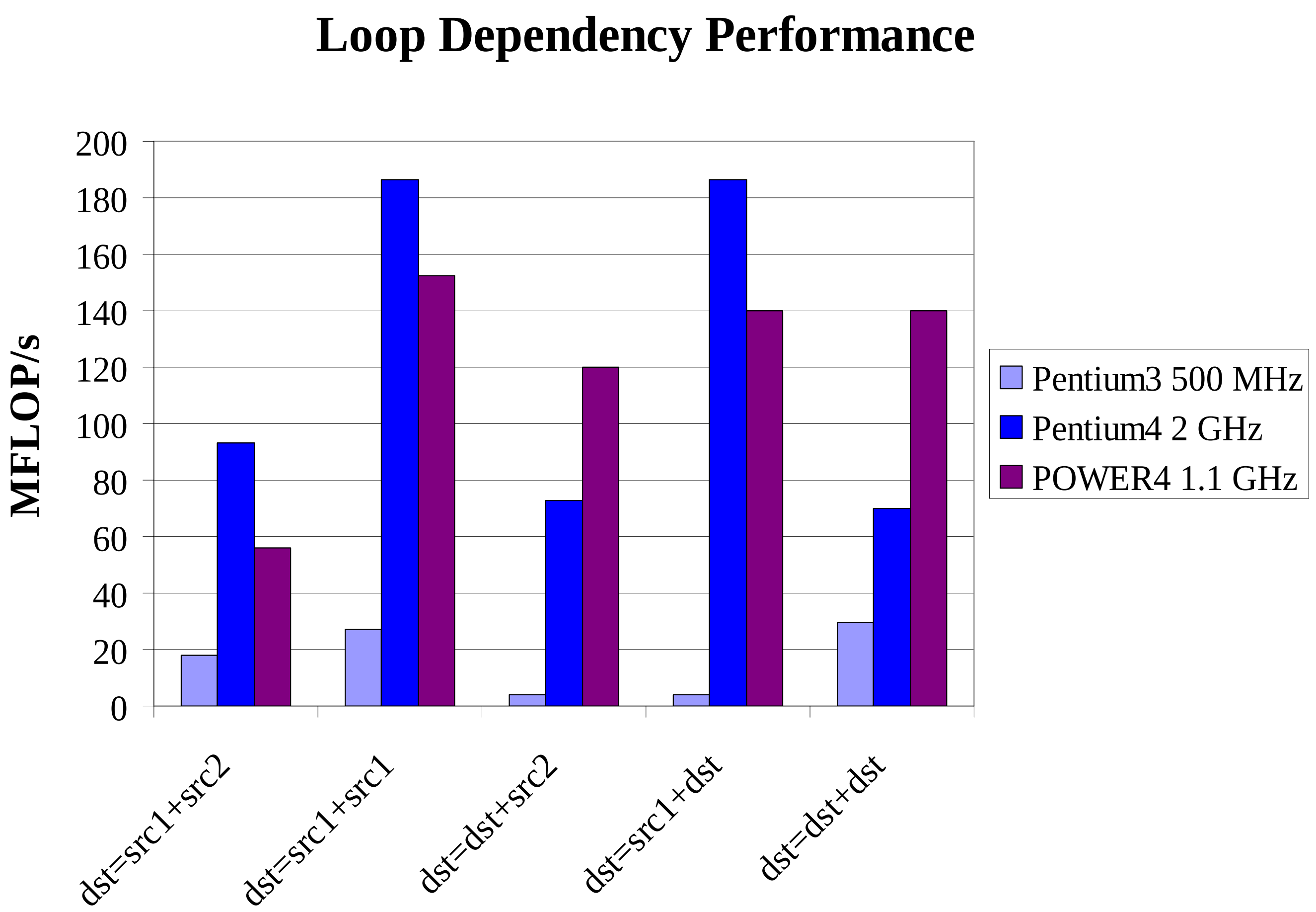
Loop Dep Example (cont’d)

```
if ((dst == src1) && (dst == src2)) {
  for (index = 1; index < length; index++) {
    dst[index] = dst[index-1] + dst[index];
  }
}
else if (dst == src1) {
  for (index = 1; index < length; index++) {
    dst[index] = dst[index-1] + src2[index];
  }
}
else if (dst == src2) {
  for (index = 1; index < length; index++) {
    dst[index] = src1[index-1] + dst[index];
  }
}
else if (src1 == src2) {
  for (index = 1; index < length; index++) {
    dst[index = src1[index-1] + src1[index];
  }
}
else {
  for (index = 1; index < length; index++) {
    dst[index] = src1[index-1] + src2[index];
  }
}
```

The various versions of the loop either:

- **do have loop carried dependencies**, or
- **don’t have loop carried dependencies**.

Loop Dependency Performance



Stupid Compiler Tricks

Stupid Compiler Tricks

- Tricks Compilers Play
  - Scalar Optimizations
  - Loop Optimizations
  - Inlining
- Tricks You Can Play with Compilers
  - Profiling
  - Hardware counters

Compiler Design

The people who design compilers have a lot of experience working with the languages commonly used in High Performance Computing:

- Fortran: 45ish years
- C: 30ish years
- C++: 15ish years, plus C experience

So, they’ve come up with clever ways to make programs run faster.

Tricks Compilers Play

Scalar Optimizations

- Copy Propagation
- Constant Folding
- Dead Code Removal
- Strength Reduction
- Common Subexpression Elimination
- Variable Renaming
- Loop Optimizations

Not every compiler does all of these, so it sometimes can be worth doing these by hand.

Much of this discussion is from [2] and [5].

Copy Propagation

Before

```
x = y
z = 1 + x
```

Has data dependency

Compile

After

```
x = y
z = 1 + y
```

No data dependency

Constant Folding

Before

```
add = 100
aug = 200
sum = add + aug
```

After

```
sum = 300
```

Notice that **`sum`** is actually the sum of two constants, so the compiler can precalculate it, eliminating the addition that otherwise would be performed at runtime.

Dead Code Removal

Before

```
var = 5
PRINT *, var
STOP
PRINT *, var * 2
```

After

```
var = 5
PRINT *, var
STOP
```

Since the last statement never executes, the compiler can eliminate it.

Strength Reduction

Before	After
`x = y ** 2.0`	`x = y * y`
`a = c /  2.0`	`a = c * 0.5`

Raising one value to the power of another, or dividing, is more expensive than multiplying. If the compiler can tell that the power is a small integer, or that the denominator is a constant, it'll use multiplication instead.

Common Subexpression Elimination

Before

```
d = c * (a / b)
e = (a / b) * 2.0
```

After

```
adivb = a / b
d = c * adivb
e = adivb * 2.0
```

The subexpression `(a / b)` occurs in both assignment statements, so there’s no point in calculating it twice.

This is typically only worth doing if the common subexpression is expensive to calculate.

Variable Renaming

Before	After
`x = y * z`	`x0 = y * z`
`q = r + x * 2`	`q = r + x0 * 2`
`x = a + b`	`x = a + b`

The original code has an **output dependency**, while the new code **doesn’t** – but the final value of `x` is still correct.

Loop Optimizations

- Hoisting Loop Invariant Code
- Unswitching
- Iteration Peeling
- Index Set Splitting
- Loop Interchange
- Unrolling
- Loop Fusion
- Loop Fission

Not every compiler does all of these, so it sometimes can be worth doing some of these by hand.

Much of this discussion is from [3] and [5].

Hoisting Loop Invariant Code

Code that doesn't change inside the loop is called **loop invariant**. It doesn't need to be calculated over and over.

Before

```
DO i = 1, n
  a(i) = b(i) + c * d
  e = g(n)
END DO
```

After

```
temp = c * d
DO i = 1, n
  a(i) = b(i) + temp
END DO
e = g(n)
```

Unswitching

```
DO i = 1, n
  DO j = 2, n
    IF (t(i) > 0) THEN
      a(i,j) = a(i,j) * t(i) + b(j)
    ELSE
      a(i,j) = 0.0
    END IF
  END DO
END DO
```

The condition is j-independent.

Before

```
DO i = 1, n
  IF (t(i) > 0) THEN
    DO j = 2, n
      a(i,j) = a(i,j) * t(i) + b(j)
    END DO
  ELSE
    DO j = 2, n
      a(i,j) = 0.0
    END DO
  END IF
END DO
```

So, it can migrate outside the j loop.

After

Iteration Peeling

Before

```
DO i = 1, n
  IF ((i == 1) .OR. (i == n)) THEN
    x(i) = y(i)
  ELSE
    x(i) = y(i + 1) + y(i – 1)
  END IF
END DO
```

We can eliminate the IF by ***peeling*** the weird iterations.

After

```
x(1) = y(1)
DO i = 2, n - 1
  x(i) = y(i + 1) + y(i – 1)
END DO
x(n) = y(n)
```

Index Set Splitting

```
DO i = 1, n
  a(i) = b(i) + c(i)
  IF (i > 10) THEN
    d(i) = a(i) + b(i – 10)
  END IF
END DO
```

Before

```
DO i = 1, 10
  a(i) = b(i) + c(i)
END DO
DO i = 11, n
  a(i) = b(i) + c(i)
  d(i) = a(i) + b(i – 10)
END DO
```

After

Note that this is a generalization of **peeling**.

Loop Interchange

Before

```
DO i = 1, ni
  DO j = 1, nj
    a(i,j) = b(i,j)
  END DO
END DO
```

After

```
DO j = 1, nj
  DO i = 1, ni
    a(i,j) = b(i,j)
  END DO
END DO
```

Array elements **`a(i,j)`** and **`a(i+1,j)`** are near each other in memory, while **`a(i,j+1)`** may be far, so it makes sense to make the **`i`** loop be the inner loop. (This is reversed in C, C++ and Java.)

Unrolling

Before

```
DO i = 1, n
  a(i) = a(i)+b(i)
END DO
```

After

```
DO i = 1, n, 4
  a(i)   = a(i)  +b(i)
  a(i+1) = a(i+1)+b(i+1)
  a(i+2) = a(i+2)+b(i+2)
  a(i+3) = a(i+3)+b(i+3)
END DO
```

You generally **shouldn’t** unroll by hand.

Why Do Compilers Unroll?

We saw last time that a loop with a lot of operations gets better performance (up to some point), especially if there are lots of arithmetic operations but few main memory loads and stores.

Unrolling creates multiple operations that typically load from the same, or adjacent, cache lines.

So, an unrolled loop has more operations without increasing the memory accesses by much.

Also, unrolling decreases the number of comparisons on the loop counter variable, and the number of branches to the top of the loop.

Loop Fusion

```
DO i = 1, n
  a(i) = b(i) + 1
END DO
DO i = 1, n
  c(i) = a(i) / 2
END DO
DO i = 1, n
  d(i) = 1 / c(i)
END DO
```

Before

```
DO i = 1, n
  a(i) = b(i) + 1
  c(i) = a(i) / 2
  d(i) = 1 / c(i)
END DO
```

After

As with unrolling, this has fewer branches. It also has fewer total memory references.

Loop Fission

```
DO i = 1, n
  a(i) = b(i) + 1
  c(i) = a(i) / 2
  d(i) = 1 / c(i)
END DO !! i = 1, n
```

Before

```
DO i = 1, n
  a(i) = b(i) + 1
END DO !! i = 1, n
DO i = 1, n
  c(i) = a(i) / 2
END DO !! i = 1, n
DO i = 1, n
  d(i) = 1 / c(i)
END DO !! i = 1, n
```

After

Fission reduces the cache load and the number of operations per iteration.

To Fuse or to Fizz?

The question of when to perform fusion versus when to perform fission, like many many optimization questions, is highly dependent on the application, the platform and a lot of other issues that get very, very complicated.

Compilers don’t always make the right choices.

That’s why it’s important to examine the actual behavior of the executable.

Inlining

Before

```
DO i = 1, n
  a(i) = func(i)
END DO
…
REAL FUNCTION func (x)
  …
  func = x * 3
END FUNCTION func
```

After

```
DO i = 1, n
  a(i) = i * 3
END DO
```

When a function or subroutine is ***inlined***, its contents are transferred directly into the calling routine, eliminating the overhead of making the call.

Tricks You Can Play with Compilers

The Joy of Compiler Options

Every compiler has a different set of options that you can set.

Among these are options that control single processor optimization: superscalar, pipelining, vectorization, scalar optimizations, loop optimizations, inlining and so on.

Example Compile Lines

- IBM Regatta

```
xlf90 –O –qmaxmem=-1 –qarch=auto
  –qtune=auto –qcache=auto –qhot
```

- Intel

```
ifort –O –tpp7 -xW
```

- Portland Group f90

```
pgf90 -fastsse –Mdalign –Mvect=assoc
```

- NAG f95

```
f95 –O4 –Ounsafe –ieee=nonstd
```

- SGI Origin2000

```
f90 –Ofast –ipa
```

- Sun UltraSPARC

```
f90 –fast
```

- CrayJ90

```
f90 –O 3,aggress,pattern,recurrence
```

What Does the Compiler Do?

Example: NAG f95 compiler

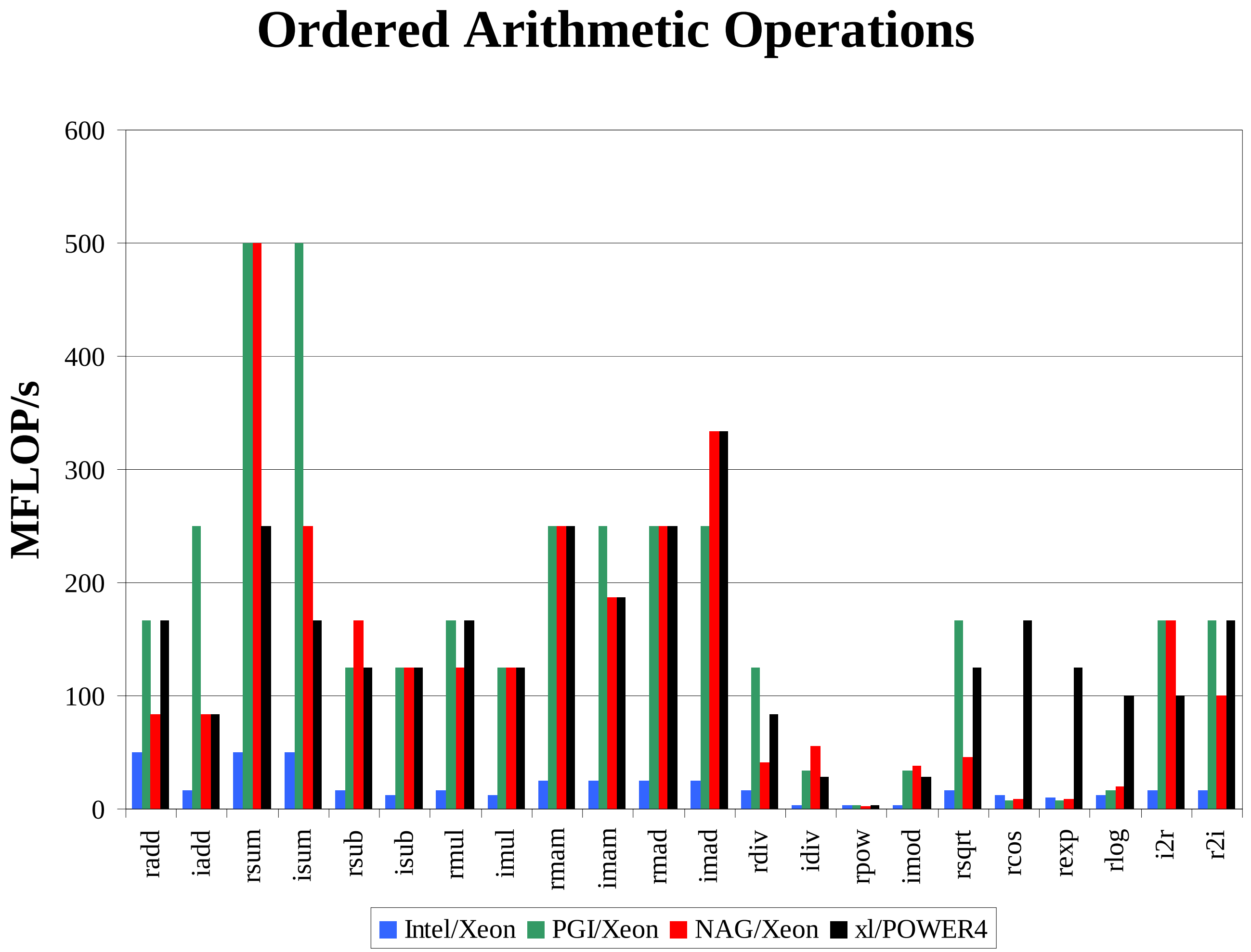
```
f95 –O<level> source.f90
```

Possible levels are `–O0, -O1, -O2, -O3, -O4`:

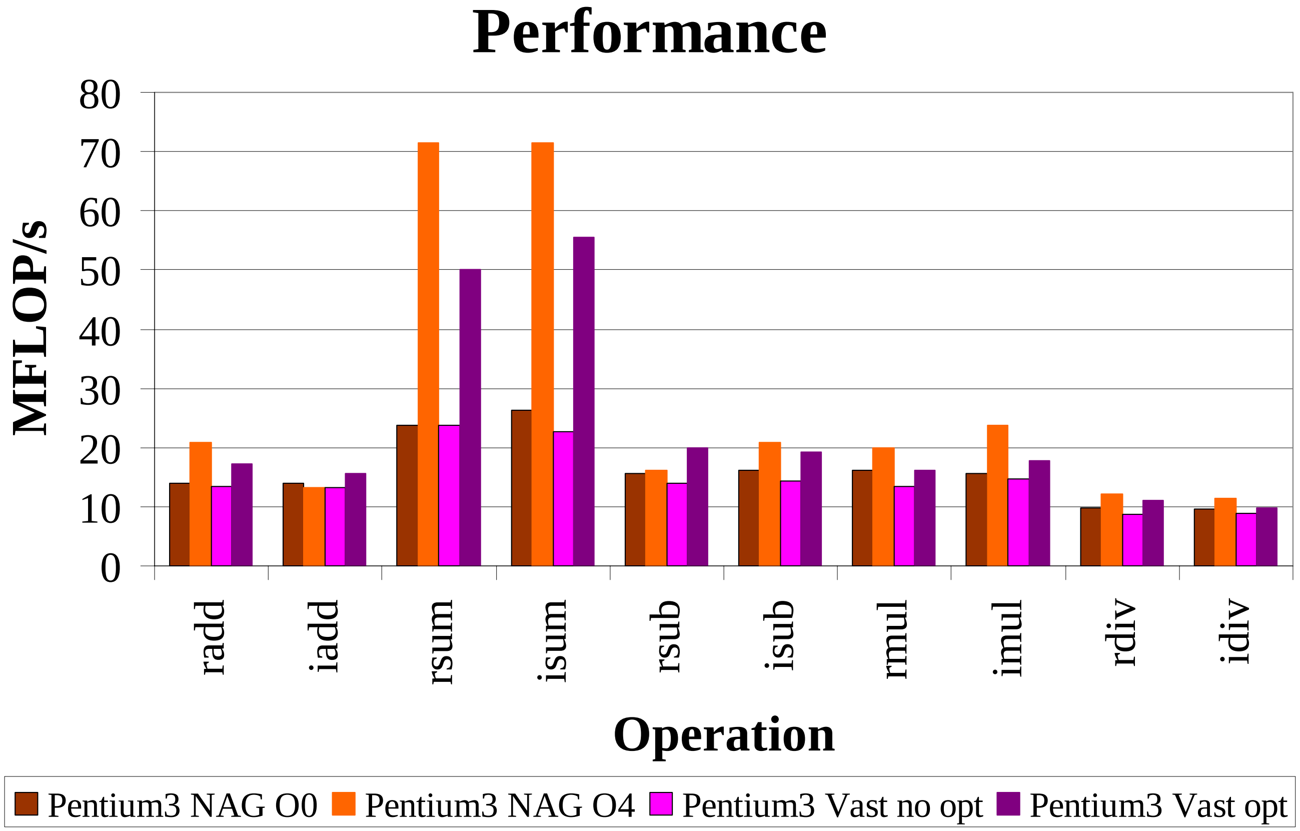
```
-O0    No optimisation. …
-O1    Minimal quick optimisation.
-O2    Normal optimisation.
-O3    Further optimisation.
-O4    Maximal optimisation.
```
[4]

The man page is pretty cryptic.

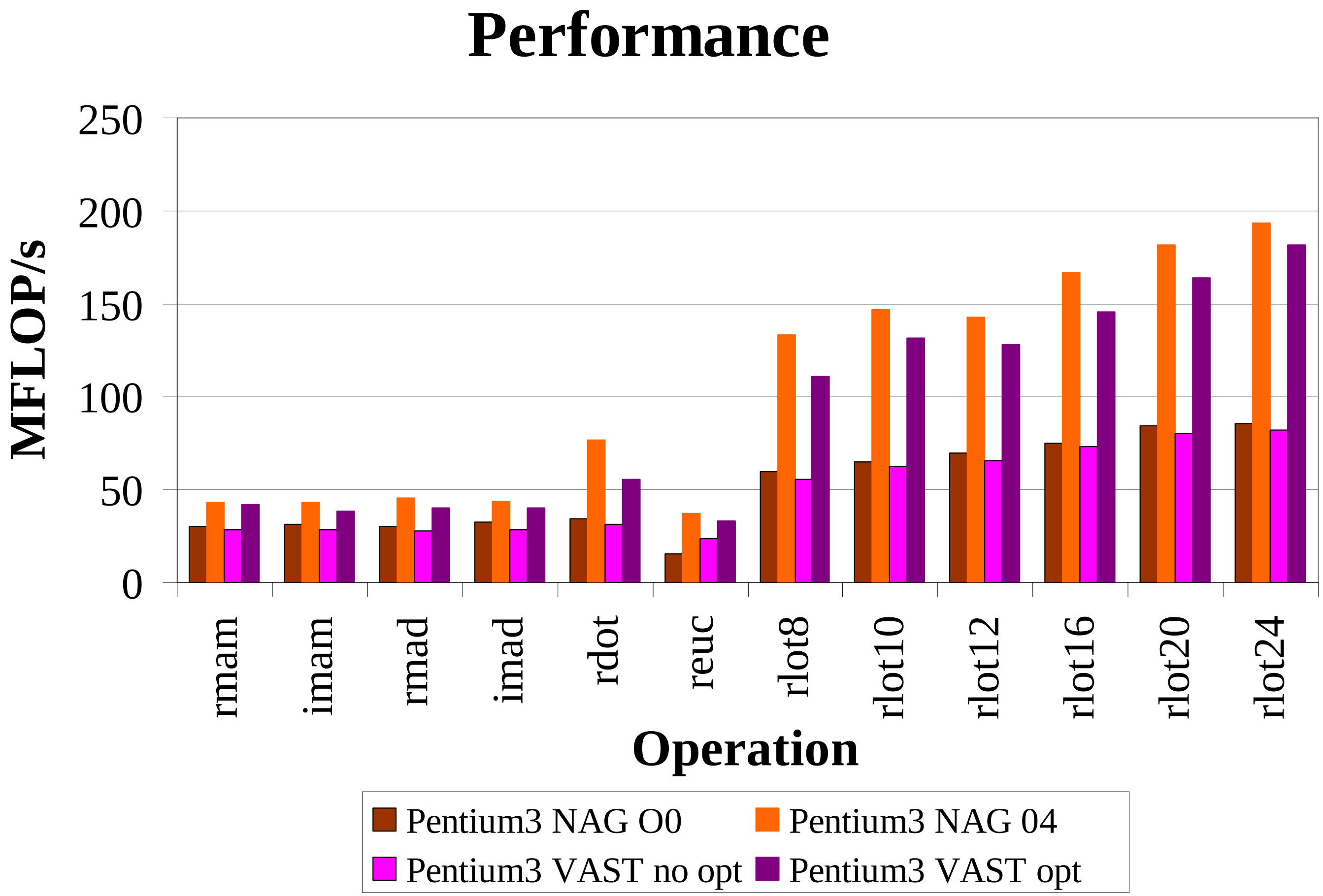
Arithmetic Operation Speeds



Optimization Performance



More Optimized Performance



Profiling

Profiling

Profiling means collecting data about how a program executes.

The two major kinds of profiling are:

- Subroutine profiling
- Hardware timing

Subroutine Profiling

Subroutine profiling means finding out how much time is spent in each routine.

The 90-10 Rule: Typically, a program spends 90% of its runtime in 10% of the code.

Subroutine profiling tells you what parts of the program to spend time optimizing and what parts you can ignore.

Specifically, at regular intervals (e.g., every millisecond), the program takes note of what instruction it's currently on.

Profiling Example

On the IBM Regatta:

```
xlf90 –O –g -pg …
```

The **–g -pg** options tell the compiler to set the executable up to collect profiling information.

Running the executable generates a file named **gmon.out**, which contains the profiling information.

Profiling Example (cont’d)

When the run has completed, a file named **`gmon.out`** has been generated.

Then:

```
gprof executable
```

produces a list of all of the routines and how much time was spent in each.

Profiling Result

```
%    cumulative   self              self     total
time    seconds   seconds    calls  ms/call  ms/call  name
27.6      52.72     52.72   480000     0.11     0.11  longwave_ [5]
24.3      99.06     46.35      897    51.67    51.67  mpdata3_ [8]
 7.9     114.19     15.13      300    50.43    50.43  turb_ [9]
 7.2     127.94     13.75      299    45.98    45.98  turb_scalar_ [10]
 4.7     136.91      8.96      300    29.88    29.88  advect2_z_ [12]
 4.1     144.79      7.88      300    26.27    31.52  cloud_ [11]
 3.9     152.22      7.43      300    24.77   212.36  radiation_ [3]
 2.3     156.65      4.43      897     4.94    56.61  smlr_ [7]
 2.2     160.77      4.12      300    13.73    24.39  tke_full_ [13]
 1.7     163.97      3.20      300    10.66    10.66  shear_prod_ [15]
 1.5     166.79      2.82      300     9.40     9.40  rhs_ [16]
 1.4     169.53      2.74      300     9.13     9.13  advect2_xy_ [17]
 1.3     172.00      2.47      300     8.23    15.33  poisson_ [14]
 1.2     174.27      2.27   480000     0.00     0.12  long_wave_ [4]
 1.0     176.13      1.86      299     6.22   177.45  advect_scalar_ [6]
 0.9     177.94      1.81      300     6.04     6.04  buoy_ [19]
...
```

Hardware Timing

In addition to learning about which routines dominate in your program, you might also want to know how the hardware behaves; e.g., you might want to know how often you get a cache miss.

Many supercomputer CPUs have special hardware that measures such events, called ***event counters***.

Hardware Timing Example

On SGI Origin2000:

```
perfex –x –a executable
```

This command produces a list of hardware counts.

Hardware Timing Results

```
Cycles.......................      1350795704000
Decoded instructions...........    1847206417136
Decoded loads...................... 448877703072
Decoded stores...................    76766538224
Grad floating point instructions... 575482548960
Primary data cache misses.........   36090853008
Secondary data cache misses...        5537223904
. . .
```

This hardware counter profile shows that only 1% of memory accesses resulted in L2 cache misses, which is good, but that it only got 0.42 FLOPs per cycle, out of a peak of 2 FLOPs per cycle, which is bad.

Next Time

Part V:

Shared Memory Multithreading

References

[1] Kevin Dowd and Charles Severance, *High Performance Computing*, 2nd ed. O'Reilly, 1998, p. 173-191.

[2] Ibid, p. 91-99.

[3] Ibid, p. 146-157.

[4] NAG f95 man page.

[5] Michael Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Co., 1996.

[6] Kevin R. Wadleigh and Isom L. Crawford, *Software Optimization for High Performance Computing*, Prentice Hall PTR, 2000, pp. 14-15.