Supercomputing in Plain English

Stupid Compiler Tricks

Henry Neeman, Director

Director, OU Supercomputing Center for Education & Research (OSCER)
Assistant Vice President, Information Technology – Research Strategy Advisor
Associate Professor, College of Engineering
Adjunct Associate Professor, School of Computer Science
University of Oklahoma

Tuesday February 10 2015

This is an experiment!

It's the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the toll free phone bridge to fall back on.

PLEASE MUTE YOURSELF

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

At OU, we will turn off the sound on all conferencing technologies.

That way, we won't have problems with echo cancellation.

Of course, that means we cannot hear questions.

So for questions, you'll need to send e-mail.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE REGISTER

If you haven’t already registered, please do so.

You can find the registration link on the SiPE webpage:

http://www.oscer.ou.edu/education/

Our ability to continue providing Supercomputing in Plain English depends on being able to show strong participation.

We use our headcounts, institution counts and state counts (since 2001, over 2000 served, from every US state except RI and VT, plus 17 other countries, on every continent except Australia and Antarctica) to improve grant proposals.

Download the Slides Beforehand

Before the start of the session, please download the slides from the Supercomputing in Plain English website:

http://www.oscer.ou.edu/education/

That way, if anything goes wrong, you can still follow along with just audio.

PLEASE MUTE YOURSELF.

H.323 (Polycom etc) #1

If you want to use H.323 videoconferencing – for example, Polycom – then:

- If you AREN'T registered with the OneNet gatekeeper (which is probably the case), then:
  - Dial **`164.58.250.51`**
  - Bring up the virtual keypad.
    On some H.323 devices, you can bring up the virtual keypad by typing:
    `#`
    (You may want to try without first, then with; some devices won't work with the #, but give cryptic error messages about it.)
  - When asked for the conference ID, or if there's no response, enter:
    `0409`
  - On most but not all H.323 devices, you indicate the end of the ID with:
    `#`

H.323 (Polycom etc) #2

If you want to use H.323 videoconferencing – for example, Polycom – then:

- If you ARE already registered with the OneNet gatekeeper (most institutions aren’t), dial:

  `2500409`

Many thanks to James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman of OneNet for providing this.

PLEASE MUTE YOURSELF.

Wowza #1

You can watch from a Windows, MacOS or Linux laptop using Wowza from the following URL:

http://jwplayer.onenet.net/stream6/sipe.html

Wowza behaves a lot like YouTube, except live.

Many thanks to James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman of OneNet for providing this.

PLEASE MUTE YOURSELF.

Wowza #2

Wowza has been tested on multiple browsers on each of:

- Windows (7 and 8): IE, Firefox, Chrome, Opera, Safari
- MacOS X: Safari, Firefox
- Linux: Firefox, Opera

PLEASE MUTE YOURSELF.

Toll Free Phone Bridge

IF ALL ELSE FAILS, you can use our toll free phone bridge:

800-832-0736

* 623 2874 #

Please mute yourself and use the phone to listen.

Don’t worry, we’ll call out slide numbers as we go.

Please use the phone bridge **ONLY** if you cannot connect any other way: the phone bridge can handle only 100 simultaneous connections, and we have over 500 participants.

Many thanks to OU CIO Loretta Early for providing the toll free phone bridge.

PLEASE MUTE YOURSELF.

Please Mute Yourself

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

(For Wowza, you don't need to do that, because the information only goes from us to you, not from you to us.)

At OU, we will turn off the sound on all conferencing technologies.

That way, we won't have problems with echo cancellation.

Of course, that means we cannot hear questions.

So for questions, you'll need to send e-mail.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

Questions via E-mail Only

Ask questions by sending e-mail to:

sipe2015@gmail.com

All questions will be read out loud and then answered out loud.

PLEASE MUTE YOURSELF.

Onsite: Talent Release Form

If you're attending onsite, you **MUST** do one of the following:

- complete and sign the Talent Release Form,

OR

- sit behind the cameras (where you can't be seen) and don't talk at all.

If you aren't onsite, then **PLEASE MUTE YOURSELF.**

TENTATIVE Schedule

Tue Jan 20: Overview: What the Heck is Supercomputing?

Tue Feb 3: The Tyranny of the Storage Hierarchy

Tue Feb 3: Instruction Level Parallelism

Tue Feb 10: Stupid Compiler Tricks

Tue Feb 17: Shared Memory Multithreading

Tue Feb 24: Distributed Multiprocessing

Tue March 3: Applications and Types of Parallelism

Tue March 10: Multicore Madness

Tue March 17: **NO SESSION** (OU's Spring Break)

Tue March 24: **NO SESSION** (Henry has a huge grant proposal due)

Tue March 31: High Throughput Computing

Tue Apr 7: GPGPU: Number Crunching in Your Graphics Card

Tue Apr 14: Grab Bag: Scientific Libraries, I/O Libraries, Visualization

Thanks for helping!

- OU IT
  - OSCER operations staff (Brandon George, Dave Akin, Brett Zimmerman, Josh Alexander, Patrick Calhoun)
  - Horst Severini, OSCER Associate Director for Remote & Heterogeneous Computing
  - Debi Gentis, OSCER Coordinator
  - Jim Summers
  - The OU IT network team
- James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman, OneNet
- Kay Avila, U Iowa
- Stephen Harrell, Purdue U

This is an experiment!

It’s the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the toll free phone bridge to fall back on.

PLEASE MUTE YOURSELF.

Coming in 2015!

Linux Clusters Institute workshop May 18-22 2015 @ OU

http://www.linuxclustersinstitute.org/workshops/

Great Plains Network Annual Meeting, May 27-29, Kansas City

Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency May 31 - June 6 2015

XSEDE2015, July 26-30, St. Louis MO

https://conferences.xsede.org/xsede15

IEEE Cluster 2015, Sep 23-27, Chicago IL

http://www.mcs.anl.gov/ieeecluster2015/

OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2015, **Sep 22-23 2015** @ OU

SC13, Nov 15-20 2015, Austin TX

http://sc15.supercomputing.org/

PLEASE MUTE YOURSELF.

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 [6].

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 (F90)

```
y = 7
IF (x <= 2) THEN
    y = 3
END IF
z = y + 1
```

Note that `(x <= 2)` means "x less than or equal to two."

The value of `y` depends on what the condition `(x <= 2)` evaluates to:

- If the condition `(x <= 2)` evaluates to `.TRUE.`, then `y` is set to `3`, so `z` is assigned `4`.
- Otherwise, `y` remains `7`, so `z` is assigned `8`.

https://en.wikipedia.org/wiki/Dependence_analysis

Branch Dependency (C)

```
y = 7;
if (x <= 2) {
    y = 3;
}
z = y + 1
```

Note that `(x <= 2)` means “x less than or equal to two.”

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

- If the condition `(x <= 2)` evaluates to `true`, then `y` is set to **3**, so `z` is assigned **4**.
- Otherwise, `y` remains **7**, so `z` is assigned **8**.

https://en.wikipedia.org/wiki/Dependence_analysis

Loop Carried Dependency (F90)

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

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

Loop Carried Dependency (C)

```
for (i = 1; i < length; i++) {
  a[i] = a[i-1] + b[i];
}
```

Here, each iteration of the loop depends on the previous: iteration `i=3` depends on iteration `i=2`, iteration `i=4` depends on iteration `i=3`, iteration `i=5` depends on iteration `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? (F)

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
```

Loop or Branch Dependency? (C)

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

```
for (i = 0; i < length; i++) {
  if (x[i] != 0) {
    y[i] = 1.0 / x[i];
  }
}
```

Call Dependency Example (F90)

```
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.

Call Dependency Example (C)

```
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 (F90)

```
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.

I/O Dependency (C)

```
x = a + b;
printf("%f", 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
```

A ***reduction*** is an operation that converts an array to a scalar.

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.

Reductions Aren’t Dependencies

```
array_sum = 0;
for (i = 0; i < length; i++) {
  array_sum = array_sum + array[i];
}
```

A ***reduction*** is an operation that converts an array to a scalar.

Other kinds of reductions: product, **&&**, **||**, 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 (F90)

“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].” [7]

```
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.

Data Dependencies (C)

"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]." [7]

```
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 (F90)

```
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.

Output Dependencies (C)

```
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];
  }
}
```

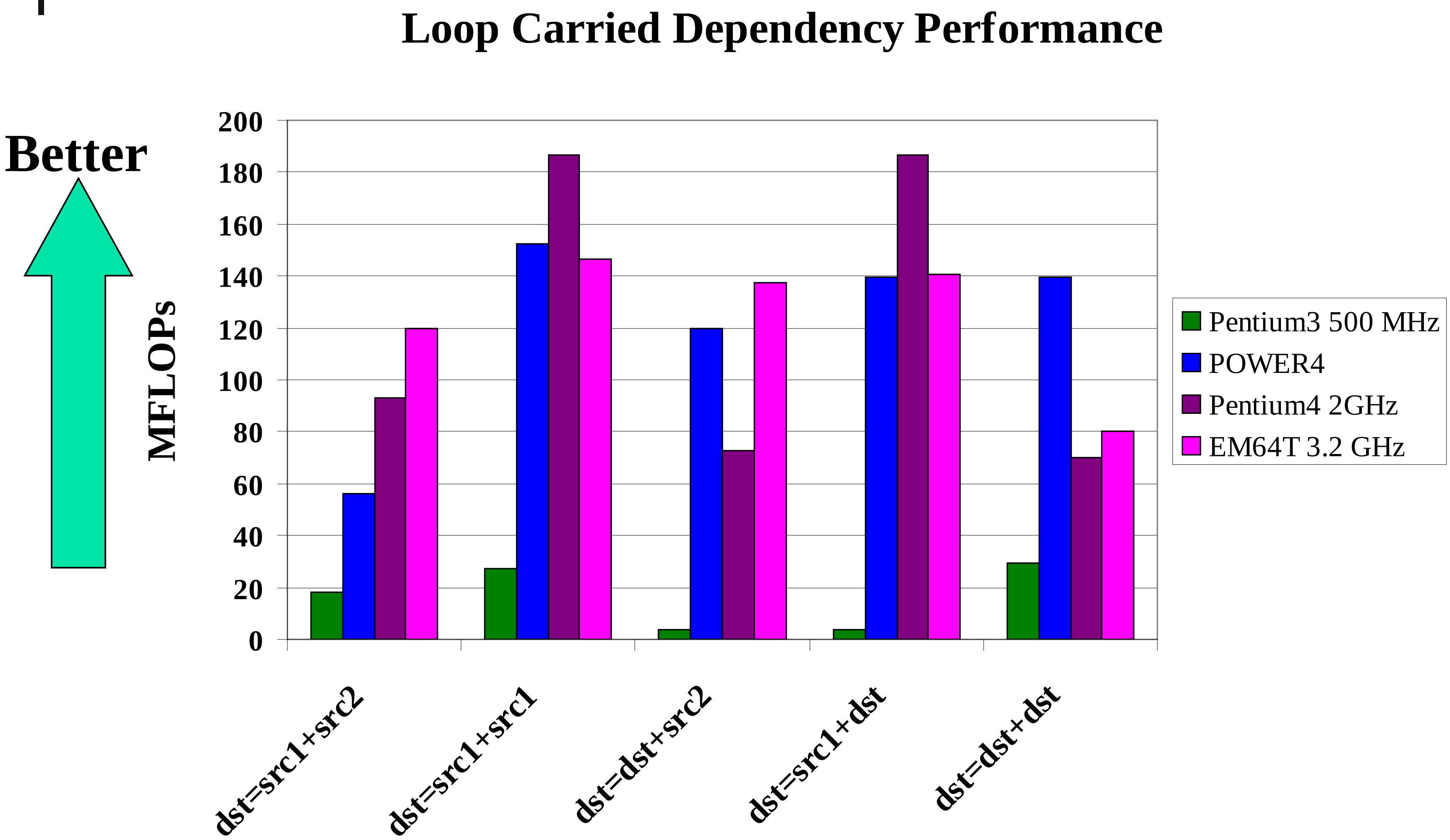
Loop Dep Example (cont'd)

```
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: 50+ years
- C: 40+ years
- C++: almost 30 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.

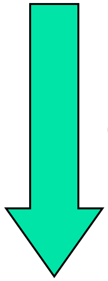
Much of this discussion is from [2] and [6].

Copy Propagation (F90)

Before

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

Has data dependency



Compile

After

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

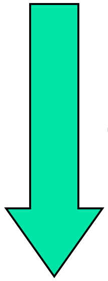
No data dependency

Copy Propagation (C)

Before

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

Has data dependency



Compile

After

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

No data dependency

Constant Folding (F90)

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.

Constant Folding (C)

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 (F90)

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.

Dead Code Removal (C)

Before

```
var = 5;
printf("%d", var);
exit(-1);
printf("%d", var * 2);
```

After

```
var = 5;
printf("%d", var);
exit(-1);
```

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

Strength Reduction (F90)

Before

```
x = y ** 2.0
a = c /  2.0
```

After

```
x = y * y
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.

Note: In Fortran, “`y ** 2.0`” means “y to the power 2.”

Strength Reduction (C)

Before

```
x = pow(y, 2.0);
a = c /  2.0;
```

After

```
x = y * y;
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.

Note: In C, "`pow(y, 2.0)`" means "y to the power 2."

Common Subexpression Elimination (F90)

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.

Common Subexpression Elimination (C)

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 (F90)

Before

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

After

```
x0 = y * z
q = r + x0 * 2
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.

Variable Renaming (C)

Before

```
x = y * z;
q = r + x * 2;
x = a + b;
```

After

```
x0 = y * z;
q = r + x0 * 2;
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

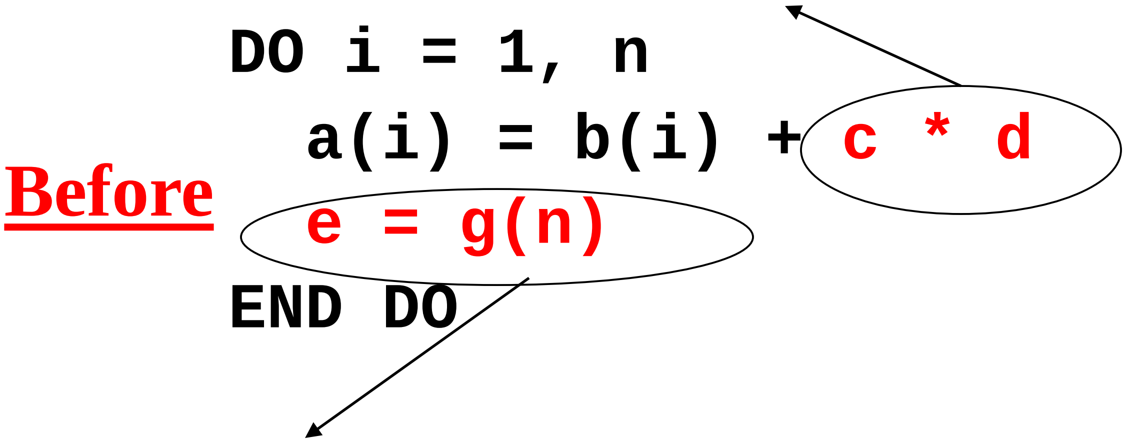
- Hoisting Loop Invariant Code
- Unswitching
- Iteration Peeling
- Index Set Splitting
- Loop Interchange
- Unrolling
- Loop Fusion
- Loop Fission

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 [6].

Hoisting Loop Invariant Code (F90)

Code that doesn’t change inside the loop is known as ***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)
```

Hoisting Loop Invariant Code (C)

Code that doesn’t change inside the loop is known as ***loop invariant***. It doesn’t need to be calculated over and over.

Before

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + c * d;
  e = g[n];
}
```

After

```
temp = c * d;
for (i = 0; i < n; i++) {
  a[i] = b[i] + temp;
}
e = g[n];
```

Unswitching (F90)

```
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

Unswitching (C)

```
for (i = 0; i < n; i++) {
  for (j = 1; j < n; j++) {
    if (t[i] > 0)
      a[i][j] = a[i][j] * t[i] + b[j];
    }
    else {
      a[i][j] = 0.0;
    }
  }
}
```

The condition is `j`-independent.

Before

```
for (i = 0; i < n; i++) {
  if (t[i] > 0) {
    for (j = 1; j < n; j++) {
      a[i][j] = a[i][j] * t[i] + b[j];
    }
  }
  else {
    for (j = 1; j < n; j++) {
      a[i][j] = 0.0;
    }
  }
}
```

So, it can migrate outside the `j` loop.

After

Iteration Peeling (F90)

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)
```

Iteration Peeling (C)

Before

```
for (i = 0; i < n; i++) {
  if ((i == 0) || (i == (n – 1))) {
    x[i] = y[i];
  }
  else {
    x[i] = y[i + 1] + y[i – 1];
  }
}
```

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

After

```
x[0] = y[0];
for (i = 1; i < n – 1; i++) {
  x[i] = y[i + 1] + y[i – 1];
}
x[n-1] = y[n-1];
```

Index Set Splitting (F90)

```
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**.

Index Set Splitting (C)

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + c[i];
  if (i >= 10) {
    d[i] = a[i] + b[i – 10];
  }
}
```

Before

```
for (i = 0; i < 10; i++) {
  a[i] = b[i] + c[i];
}
for (i = 10; i < n; i++) {
  a[i] = b[i] + c[i];
  d[i] = a[i] + b[i – 10];
}
```

After

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

Loop Interchange (F90)

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.)

Loop Interchange (C)

Before

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

After

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

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

Unrolling (F90)

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.

Unrolling (C)

Before

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

After

```
for (i = 0; i < n; i += 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];
}
```

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 (F90)

```
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 Fusion (C)

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
}
for (i = 0; i < n; i++) {
  c[i] = a[i] / 2;
}
for (i = 0; i < n; i++) {
  d[i] = 1 / c[i];
}
```

Before

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] / 2;
  d[i] = 1 / c[i];
}
```

After

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

Loop Fission (F90)

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

Before

```
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
```

After

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

Loop Fission (C)

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] / 2;
  d[i] = 1 / c[i];
}
```

Before

```
for (i = 0; i < n; i++) {
  a[i] = b[i] + 1;
}
for (i = 0; i < n; i++) {
  c[i] = a[i] / 2;
}
for (i = 0; i < n; i++) {
  d[i] = 1 / c[i];
}
```

After

Fission reduces the cache footprint 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 (F90)

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.

Inlining (C)

Before

```
for (i = 0;
     i < n; i++) {
  a[i] = func(i+1);
}
…
float func (x) {
  …
  return x * 3;
}
```

After

```
for (i = 0;
     i < n; i++) {
  a[i] = (i+1) * 3;
}
```

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 XL

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

- Intel

  ```
  ifort –O -march=corei7-avx -xAVX -xhost
  ```

- Portland Group f90

  ```
  pgf90 –O3 -tp=sandybridge
  ```

- NAG f95

  ```
  nagfor –O4 –Ounsafe
  ```

What Does the Compiler Do? #1

Example: NAG **`nagfor`** compiler [4]

```
nagfor –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.
```

The man page is pretty cryptic.

What Does the Compiler Do? #2

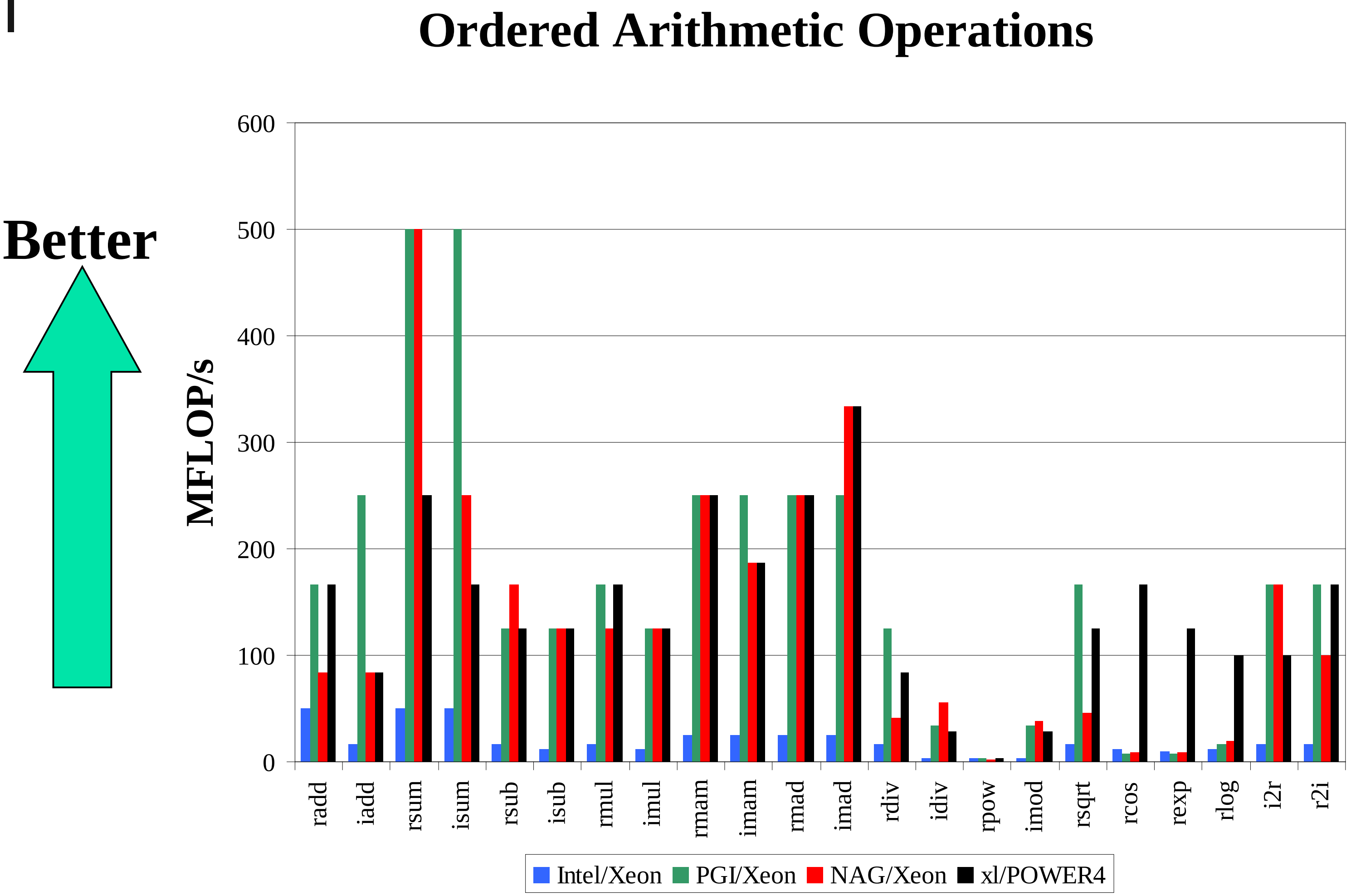
Example: Intel **`ifort`** compiler [5]

```
ifort –O<level> source.f90
```

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

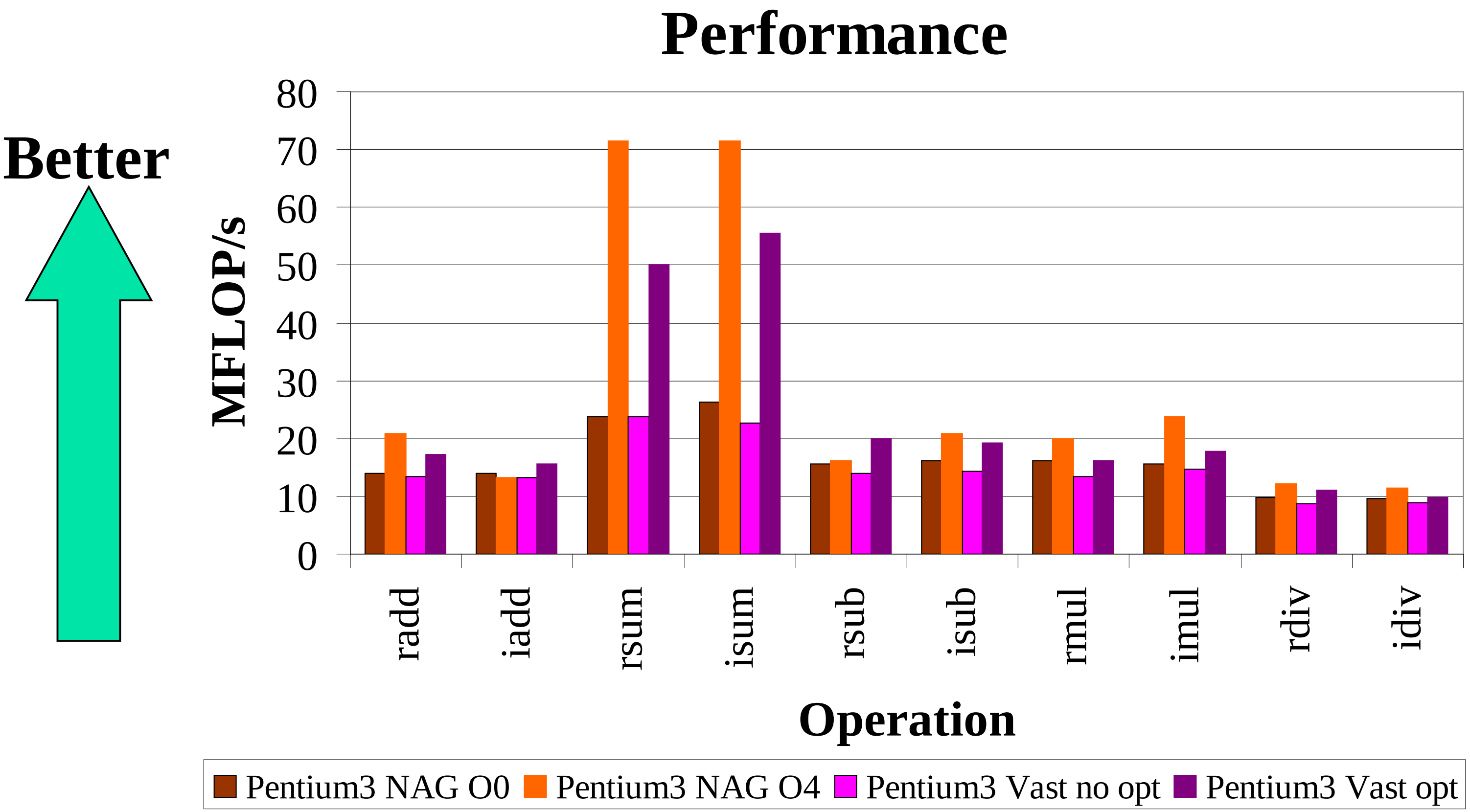
```
-O0    Disables all optimizations. ....
-O1    Enables optimizations for speed ....
-O2    ....
 Inlining of intrinsics.
 Intra-file interprocedural optimizations, which
include: inlining, constant propagation, forward
substitution, routine attribute propagation,
variable address-taken analysis, dead static
function elimination, and removal of unreferenced
variables.
-O3    Performs O2 optimizations and enables more
aggressive   loop transformations such as Fusion,
Block-Unroll-and-Jam,  and collapsing IF statements.
...
```

Arithmetic Operation Speeds

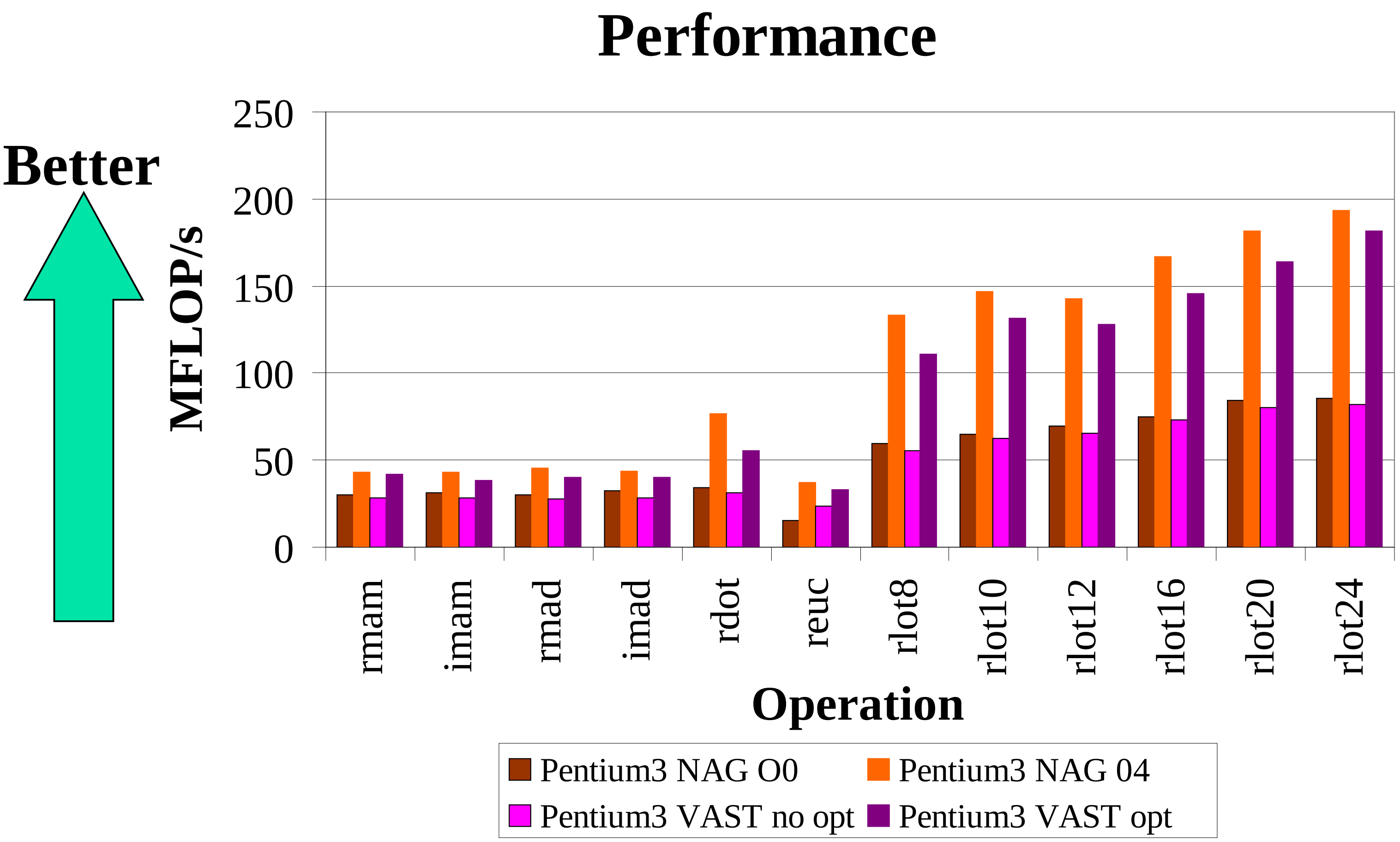


Optimization Performance

Better



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 GNU compilers systems:

```
gcc –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]
...
```

TENTATIVE Schedule

Tue Jan 20: Overview: What the Heck is Supercomputing?
Tue Feb 3: The Tyranny of the Storage Hierarchy
Tue Feb 3: Instruction Level Parallelism
Tue Feb 10: Stupid Compiler Tricks
Tue Feb 17: Shared Memory Multithreading
Tue Feb 24: Distributed Multiprocessing
Tue March 3: Applications and Types of Parallelism
Tue March 10: Multicore Madness
Tue March 17: **NO SESSION** (OU's Spring Break)
Tue March 24: **NO SESSION** (Henry has a huge grant proposal due)
Tue March 31: High Throughput Computing
Tue Apr 7: GPGPU: Number Crunching in Your Graphics Card
Tue Apr 14: Grab Bag: Scientific Libraries, I/O Libraries, Visualization

Thanks for helping!

- OU IT
  - OSCER operations staff (Brandon George, Dave Akin, Brett Zimmerman, Josh Alexander, Patrick Calhoun)
  - Horst Severini, OSCER Associate Director for Remote & Heterogeneous Computing
  - Debi Gentis, OSCER Coordinator
  - Jim Summers
  - The OU IT network team
- James Deaton, Skyler Donahue, Jeremy Wright and Steven Haldeman, OneNet
- Kay Avila, U Iowa
- Stephen Harrell, Purdue U

Coming in 2015!

Red Hat Tech Day, Thu Jan 22 2015 @ OU

http://goo.gl/forms/jORZCz9xh7

Linux Clusters Institute workshop May 18-22 2015 @ OU

http://www.linuxclustersinstitute.org/workshops/

Great Plains Network Annual Meeting, May 27-29, Kansas City

Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency May 31 - June 6 2015

XSEDE2015, July 26-30, St. Louis MO

https://conferences.xsede.org/xsede15

IEEE Cluster 2015, Sep 23-27, Chicago IL

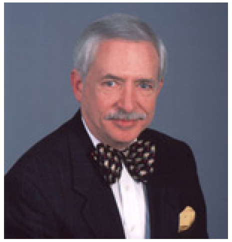
http://www.mcs.anl.gov/ieeecluster2015/

OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2015, **Sep 22-23 2015** @ OU

SC13, Nov 15-20 2015, Austin TX

http://sc15.supercomputing.org/

OK Supercomputing Symposium 2015


2003 Keynote:
Peter Freeman
NSF
Computer & Information Science & Engineering
Assistant Director

2004 Keynote:
Sangtae Kim
NSF Shared Cyberinfrastructure
Division Director

2005 Keynote:
Walt Brooks
NASA Advanced Supercomputing
Division Director

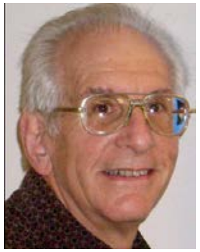
2006 Keynote:
Dan Atkins
Head of NSF's
Office of Cyberinfrastructure

2007 Keynote:
Jay Boisseau
Director
Texas Advanced Computing Center
U. Texas Austin

2008 Keynote:
José Munoz
Deputy Office Director/Senior Scientific Advisor
NSF Office of Cyberinfrastructure

2009 Keynote:
Douglass Post
Chief Scientist
US Dept of Defense
HPC Modernization Program

2010 Keynote:
Horst Simon
Deputy Director
Lawrence Berkeley National Laboratory


2011 Keynote:
Barry Schneider
Program Manager
National Science Foundation

2012 Keynote:
Thom Dunning
Director
National Center for Supercomputing Applications

2013 Keynote:
John Shalf
Dept Head CS
Lawrence Berkeley Lab
CTO, NERSC

2014 Keynote:
Irene Qualters
Division Director
Advanced Cyberinfarstructure Division, NSF

FREE!
Wed Sep 23 2015
@ OU

Reception/Poster Session
Tue Sep 22 2015 @ OU
Symposium
Wed Sep 23 2015 @ OU

Thanks for your attention!

Questions?

www.oscer.ou.edu

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, version 5.1.

[5] Intel **`ifort`** man page, version 10.1.

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

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