Supercomputing in Plain English

Part IV:

Henry Neeman, Director

OU Supercomputing Center for Education & Research
University of Oklahoma
Wednesday September 19 2007

Outline

- Dependency Analysis
  - What is Dependency Analysis?
  - Control Dependencies
  - Data Dependencies
- Stupid Compiler Tricks
  - Tricks the Compiler Plays

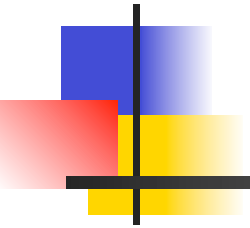
Borrowed Slides!!!

- This entire section of slides was taken from

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

- They have a course much like the one offered here at ARSC
- I particularly liked their treatment of dependencies and stupid compiler tricks
- This is an abridged version of their talk

http://www.oscer.ou.edu/Workshops/Compilers/sipe_compilers_20070919.ppt



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
```

Note that `(x /= 0)` means "`x` not equal to zero."

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

- If the condition `(x /= 0)` evaluates to `.TRUE.`, then `y` is set to `1.0 / x`. (1 divided by `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 `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.

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
```

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.

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**.

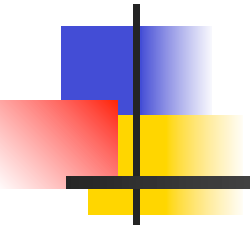
Loop Dependency Example

```
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];
  }
}
```

The various versions of the loop either:

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

Stupid Compiler Tricks



Stupid Compiler Tricks

- Tricks Compilers Play
  - Scalar Optimizations
  - Loop Optimizations
  - Inlining

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.

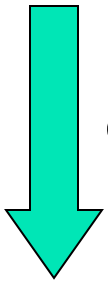
Much of this discussion is from [2] and [5].

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

```
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."

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

```
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 [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 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

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.

To Learn More Supercomputing

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

http://symposium2007.oscer.ou.edu/

References

[1] Steve Behling et al, *The POWER4 Processor Introduction and Tuning Guide*, IBM, 2001.

[2] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Order Number: 248966-015 May 2007
http://www.intel.com/design/processor/manuals/248966.pdf

[3] Kevin Dowd and Charles Severance, *High Performance Computing,* 2nd ed. O'Reilly, 1998.

[4] Code courtesy of Dan Weber, 2001.