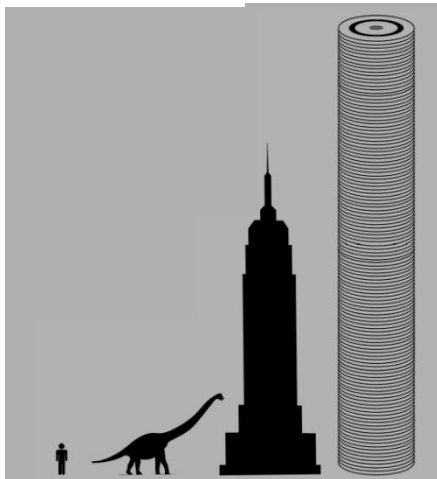


Parallel Programming & Cluster Computing

Instruction Level Parallelism

Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College

Tuesday October 11 2011



EARLHAM
COLLEGE



Outline

- What is Instruction-Level Parallelism?
- Scalar Operation
- Loops
- Pipelining
- Loop Performance
- Superpipelining
- Vectors
- A Real Example



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011

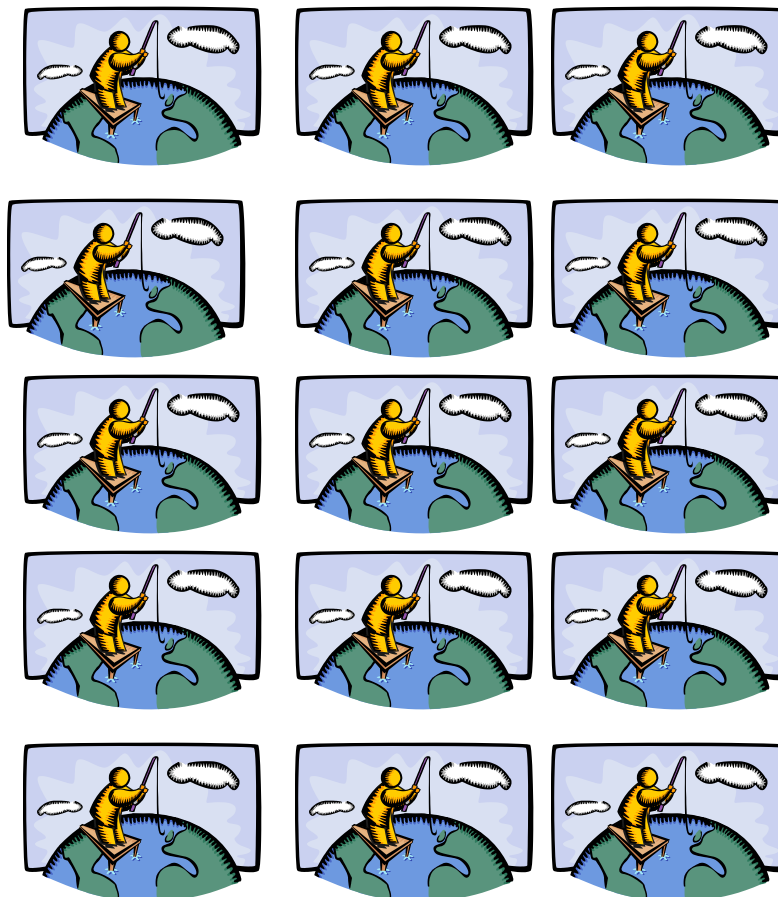




Parallelism

Parallelism means doing multiple things at the same time: You can get more work done in the same time.

Less fish ...



More fish!





What Is ILP?

Instruction-Level Parallelism (ILP) is a set of techniques for **executing multiple instructions at the same time within the same CPU core.**

(Note that ILP has **nothing to do with multicore.**)

The problem: A CPU core has lots of circuitry, and at any given time, most of it is idle, which is wasteful.

The solution: Have different parts of the CPU core work on different operations at the same time: If the CPU core has the ability to work on 10 operations at a time, then the program can, in principle, run as much as 10 times as fast (although in practice, not quite so much).





DON'T PANIC!



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011





Why You Shouldn't Panic

In general, the compiler and the CPU will do most of the heavy lifting for instruction-level parallelism.

BUT:

You need to be aware of ILP, because how your code is structured affects how much ILP the compiler and the CPU can give you.



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011





Kinds of ILP

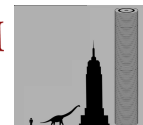
- **Superscalar**: Perform multiple operations at the same time (for example, simultaneously perform an add, a multiply and a load).
- **Pipeline**: Start performing an operation on one piece of data while finishing the same operation on another piece of data – perform different **stages** of the same operation on different sets of operands at the same time (like an assembly line).
- **Superpipeline**: A combination of superscalar and pipelining – perform multiple pipelined operations at the same time.
- **Vector**: Load multiple pieces of data into special registers and perform the same operation on all of them at the same time.





What's an Instruction?

- **Memory**: For example, load a value from a specific address in main memory into a specific register, or store a value from a specific register into a specific address in main memory.
- **Arithmetic**: For example, add two specific registers together and put their sum in a specific register – or subtract, multiply, divide, square root, etc.
- **Logical**: For example, determine whether two registers both contain nonzero values (“**AND**”).
- **Branch**: Jump from one sequence of instructions to another (for example, function call).
- ... and so on





What's a Cycle?

You've heard people talk about having a 2 GHz processor or a 3 GHz processor or whatever. (For example, consider a laptop with a 1.6 GHz Core 2 Duo.)

Inside every CPU is a little clock that ticks with a fixed frequency.

We call each tick of the CPU clock a clock cycle or a cycle.

So a 2 GHz processor has 2 billion clock cycles per second.

Typically, a primitive operation (for example, add, multiply, divide) takes a fixed number of cycles to execute (assuming no pipelining).





What's the Relevance of Cycles?

Typically, a primitive operation (for example, add, multiply, divide) takes a fixed number of cycles to execute (assuming no pipelining).

- IBM POWER4 ^[1]

- Multiply or add: 6 cycles (64 bit floating point)
- Load: 4 cycles from L1 cache
14 cycles from L2 cache



- Intel Pentium4 EM64T (Core) ^[2]

- Multiply: 7 cycles (64 bit floating point)
- Add, subtract: 5 cycles (64 bit floating point)
- Divide: 38 cycles (64 bit floating point)
- Square root: 39 cycles (64 bit floating point)
- Tangent: 240-300 cycles (64 bit floating point)



Scalar Operation





DON'T PANIC!



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011





Scalar Operation

$z = a * b + c * d;$

How would this statement be executed?

1. Load **a** into register **R0**
2. Load **b** into **R1**
3. Multiply **R2 = R0 * R1**
4. Load **c** into **R3**
5. Load **d** into **R4**
6. Multiply **R5 = R3 * R4**
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**





Does Order Matter?

$z = a * b + c * d;$

1. Load **a** into **R0**
2. Load **b** into **R1**
3. Multiply
R2 = R0 * R1
4. Load **c** into **R3**
5. Load **d** into **R4**
6. Multiply
R5 = R3 * R4
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**

1. Load **d** into **R0**
2. Load **c** into **R1**
3. Multiply **R2**
= R0 * R1
4. Load **b** into **R3**
5. Load **a** into **R4**
6. Multiply
R5 = R3 * R4
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**

In the cases where order doesn't matter, we say that the operations are *independent* of one another.





Superscalar Operation

$$z = a * b + c * d;$$

1. Load **a** into **R0** AND
load **b** into **R1**
2. Multiply **R2 = R0 * R1** AND
load **c** into **R3** AND
load **d** into **R4**
3. Multiply **R5 = R3 * R4**
4. Add **R6 = R2 + R5**
5. Store **R6** into **z**

If order doesn't matter,
then things can happen simultaneously.
So, we go from 8 operations down to 5.
(Note: there are lots of simplifying assumptions here.)





Loops



Loops Are Good

Most compilers are very good at optimizing loops, and not very good at optimizing other constructs.

Why?

```
DO index = 1, length
    dst(index) = src1(index) + src2(index)
END DO
```

```
for (index = 0; index < length; index++) {
    dst[index] = src1[index] + src2[index];
}
```





Why Loops Are Good

- Loops are very common in many programs.
- Also, it's easier to optimize loops than more arbitrary sequences of instructions: when a program does the same thing over and over, it's easier to predict what's likely to happen next.

So, hardware vendors have designed their products to be able to execute loops quickly.





DON'T PANIC!



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011





Superscalar Loops (C)

```
for (i = 0; i < length; i++) {  
    z[i] = a[i] * b[i] + c[i] * d[i];  
}
```

Each of the iterations is completely independent of all of the other iterations; for example,

$$z[0] = a[0] * b[0] + c[0] * d[0]$$

has nothing to do with

$$z[1] = a[1] * b[1] + c[1] * d[1]$$

Operations that are independent of each other can be performed in parallel.





Superscalar Loops (F90)

```
DO i = 1, length  
  z(i) = a(i) * b(i) + c(i) * d(i)  
END DO
```

Each of the iterations is completely independent of all of the other iterations; for example,

$$z(1) = a(1) * b(1) + c(1) * d(1)$$

has nothing to do with

$$z(2) = a(2) * b(2) + c(2) * d(2)$$

Operations that are independent of each other can be performed in parallel.





Superscalar Loops

```
for (i = 0; i < length; i++) {  
    z[i] = a[i] * b[i] + c[i] * d[i];  
}
```

1. Load $a[i]$ into $R0$ AND load $b[i]$ into $R1$
2. Multiply $R2 = R0 * R1$ AND load $c[i]$ into $R3$ AND load $d[i]$ into $R4$
3. Multiply $R5 = R3 * R4$ AND load $a[i+1]$ into $R0$ AND load $b[i+1]$ into $R1$
4. Add $R6 = R2 + R5$ AND load $c[i+1]$ into $R3$ AND load $d[i+1]$ into $R4$
5. Store $R6$ into $z[i]$ AND multiply $R2 = R0 * R1$
6. etc etc etc

Once this loop is “in flight,” each iteration adds only 2 operations to the total, not 8.





Example: IBM POWER4

8-way Superscalar: can execute up to 8 operations at the same time^[1]

- 2 integer arithmetic or logical operations, and
- 2 floating point arithmetic operations, and
- 2 memory access (load or store) operations, and
- 1 branch operation, and
- 1 conditional operation





Pipelining



Pipelining

Pipelining is like an assembly line or a bucket brigade.

- An operation consists of multiple stages.
- After a particular set of operands

$$z(i) = a(i) * b(i) + c(i) * d(i)$$

completes a particular stage, they move into the next stage.

- Then, another set of operands

$$z(i+1) = a(i+1) * b(i+1) + c(i+1) * d(i+1)$$

can move into the stage that was just abandoned by the previous set.





DON'T PANIC!

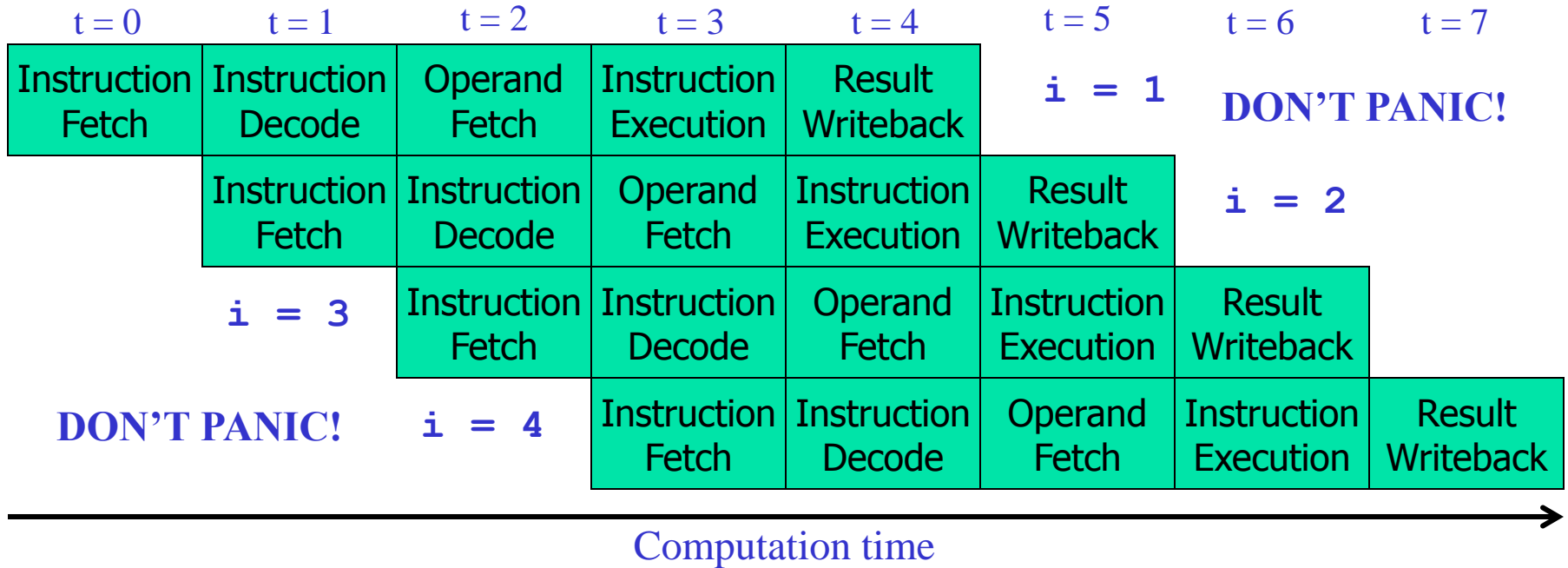


Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011

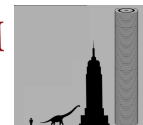




Pipelining Example



If each stage takes, say, one CPU cycle, then once the loop gets going, each iteration of the loop increases the total time by only one cycle. So a loop of length 1000 takes only 1004 cycles. ^[3]





Pipelines: Example

- IBM POWER4: pipeline length \cong 15 stages ^[1]





Some Simple Loops (F90)

```
DO index = 1, length
  dst(index) = src1(index) + src2(index)
END DO
```

```
DO index = 1, length
  dst(index) = src1(index) - src2(index)
END DO
```

```
DO index = 1, length
  dst(index) = src1(index) * src2(index)
END DO
```

```
DO index = 1, length
  dst(index) = src1(index) / src2(index)
END DO
```

```
DO index = 1, length
  sum = sum + src(index)
END DO
```

Reduction: convert
array to scalar





Some Simple Loops (C)

```
for (index = 0; index < length; index++) {  
    dst[index] = src1[index] + src2[index];  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = src1[index] - src2[index];  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = src1[index] * src2[index];  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = src1[index] / src2[index];  
}
```

```
for (index = 0; index < length; index++) {  
    sum = sum + src[index];  
}
```





Slightly Less Simple Loops (F90)

```
DO index = 1, length
  dst(index) = src1(index) ** src2(index) !! src1 ^ src2
END DO
```

```
DO index = 1, length
  dst(index) = MOD(src1(index), src2(index))
END DO
```

```
DO index = 1, length
  dst(index) = SQRT(src(index))
END DO
```

```
DO index = 1, length
  dst(index) = COS(src(index))
END DO
```

```
DO index = 1, length
  dst(index) = EXP(src(index))
END DO
```

```
DO index = 1, length
  dst(index) = LOG(src(index))
END DO
```



Slightly Less Simple Loops (C)

```
for (index = 0; index < length; index++) {  
    dst[index] = pow(src1[index], src2[index]);  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = src1[index] % src2[index];  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = sqrt(src[index]);  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = cos(src[index]);  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = exp(src[index]);  
}
```

```
for (index = 0; index < length; index++) {  
    dst[index] = log(src[index]);  
}
```




Loop Performance



Performance Characteristics

- Different operations take different amounts of time.
- Different processor types have different performance characteristics, but there are some characteristics that many platforms have in common.
- Different compilers, even on the same hardware, perform differently.
- On some processors, floating point and integer speeds are similar, while on others they differ.

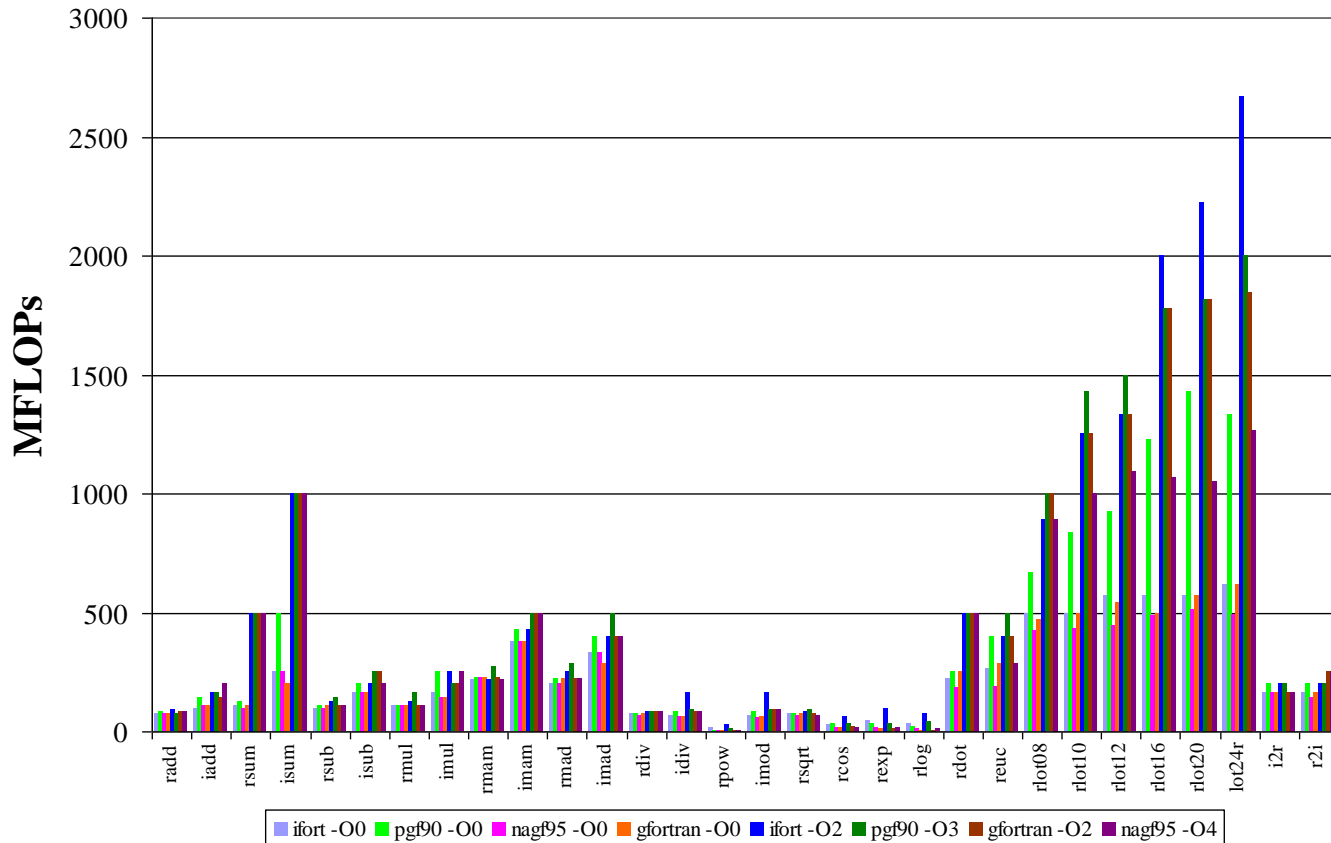
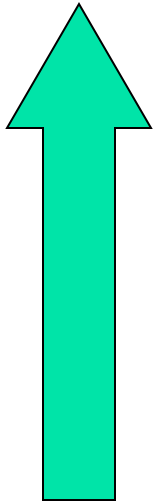




Arithmetic Operation Speeds

Arithmetic Performance on Pentium4 EM64T
(Irwindale 3.2 GHz)

Better



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011





Fast and Slow Operations

- **Fast**: sum, add, subtract, multiply
- **Medium**: divide, mod (that is, remainder)
- **Slow**: transcendental functions (sqrt, sin, exp)
- **Incredibly slow**: power x^y for real x and y

On most platforms, divide, mod and transcendental functions are not pipelined, so a code will run faster if most of it is just adds, subtracts and multiplies.

For example, solving an $N \times N$ system of linear equations by LU decomposition uses on the order of N^3 additions and multiplications, but only on the order of N divisions.





What Can Prevent Pipelining?

Certain events make it very hard (maybe even impossible) for compilers to pipeline a loop, such as:

- array elements accessed in **random order**
- loop body **too complicated**
- **if statements** inside the loop (on some platforms)
- premature **loop exits**
- function/subroutine **calls**
- **I/O**





How Do They Kill Pipelining?

- **Random access order**: Ordered array access is common, so pipelining hardware and compilers tend to be designed under the assumption that most loops will be ordered. Also, the pipeline will constantly *stall* because data will come from main memory, not cache.
- **Complicated loop body**: The compiler gets too overwhelmed and can't figure out how to schedule the instructions.





How Do They Kill Pipelining?

- if statements in the loop: On some platforms (but not all), the pipelines need to perform exactly the same operations over and over; **if** statements make that impossible.

However, many CPUs can now perform speculative execution: both branches of the **if** statement are executed while the condition is being evaluated, but only one of the results is retained (the one associated with the condition's value).

Also, many CPUs can now perform branch prediction to head down the most likely compute path.





How Do They Kill Pipelining?

- Function/subroutine calls interrupt the flow of the program even more than **if** statements. They can take execution to a completely different part of the program, and pipelines aren't set up to handle that.
- Loop exits are similar. Most compilers can't pipeline loops with premature or unpredictable exits.
- I/O: Typically, I/O is handled in subroutines (above). Also, I/O instructions can take control of the program away from the CPU (they can give control to I/O devices).





What If No Pipelining?

SLOW!

(on most platforms)



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011

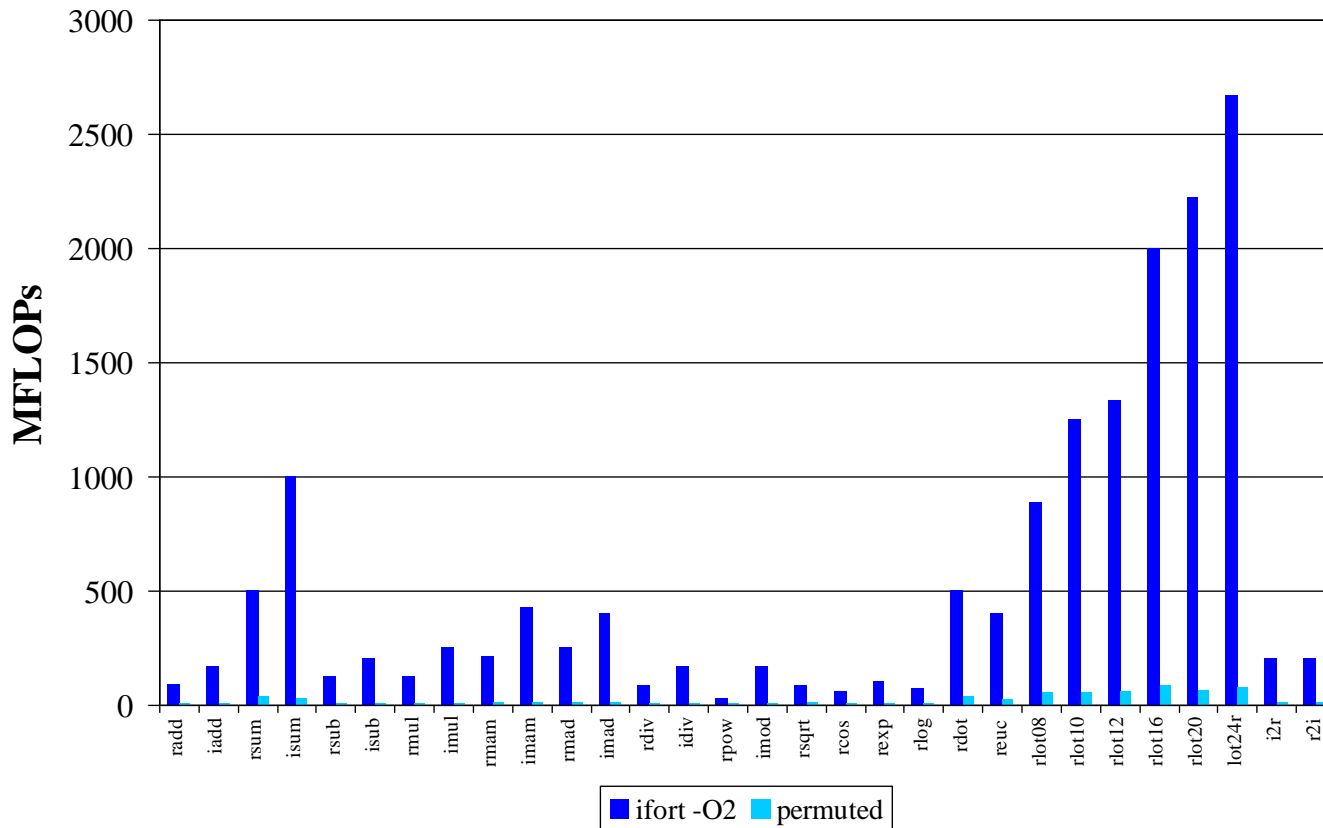
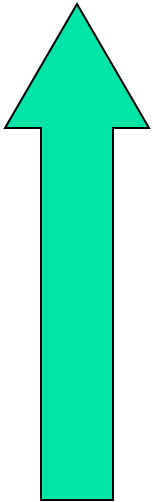




Randomly Permuted Loops

Arithmetic Performance: Ordered vs Random
(Irwindale 3.2 GHz)

Better



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011



Superpipelining





Superpipelining

Superpipelining is a combination of superscalar and pipelining.

So, a superpipeline is a collection of multiple pipelines that can operate simultaneously.

In other words, several different operations can execute simultaneously, and each of these operations can be broken into stages, each of which is filled all the time.

So you can get multiple operations per CPU cycle.

For example, a IBM Power4 can have over 200 different operations “in flight” at the same time.^[1]





More Operations At a Time

- If you put more operations into the code for a loop, you can get better performance:
 - more operations can execute at a time (use more pipelines), and
 - you get better register/cache reuse.
- On most platforms, there's a limit to how many operations you can put in a loop to increase performance, but that limit varies among platforms, and can be quite large.





Some Complicated Loops

```
DO index = 1, length
```

```
  dst(index) = src1(index) + 5.0 * src2(index)
```

```
END DO
```

madd (or FMA):
mult then add
(2 ops)

```
dot = 0
```

```
DO index = 1, length
```

```
  dot = dot + src1(index) * src2(index)
```

```
END DO
```

dot product
(2 ops)

```
DO index = 1, length
```

```
  dst(index) = src1(index) * src2(index) + &  
& src3(index) * src4(index)
```

```
END DO
```

from our
example
(3 ops)

```
DO index = 1, length
```

```
  diff12 = src1(index) - src2(index)
```

```
  diff34 = src3(index) - src4(index)
```

```
  dst(index) = SQRT(diff12 * diff12 + diff34 * diff34)
```

```
END DO
```

Euclidean distance
(6 ops)





A Very Complicated Loop

```
lot = 0.0
DO index = 1, length
  lot = lot +
    &      src1(index) * src2(index) +      &
    &      src3(index) * src4(index) +      &
    &      (src1(index) + src2(index)) *      &
    &      (src3(index) + src4(index)) *      &
    &      (src1(index) - src2(index)) *      &
    &      (src3(index) - src4(index)) *      &
    &      (src1(index) - src3(index) +      &
    &      src2(index) - src4(index)) *      &
    &      (src1(index) + src3(index) -      &
    &      src2(index) + src4(index)) +      &
    &      (src1(index) * src3(index)) +      &
    &      (src2(index) * src4(index))
END DO
```

24 arithmetic ops per iteration

4 memory/cache loads per iteration

Parallel Programming: Instr Level Parallel

OK Supercomputing Symposium, Tue Oct 11 2011

EARLHAM
COLLEGE

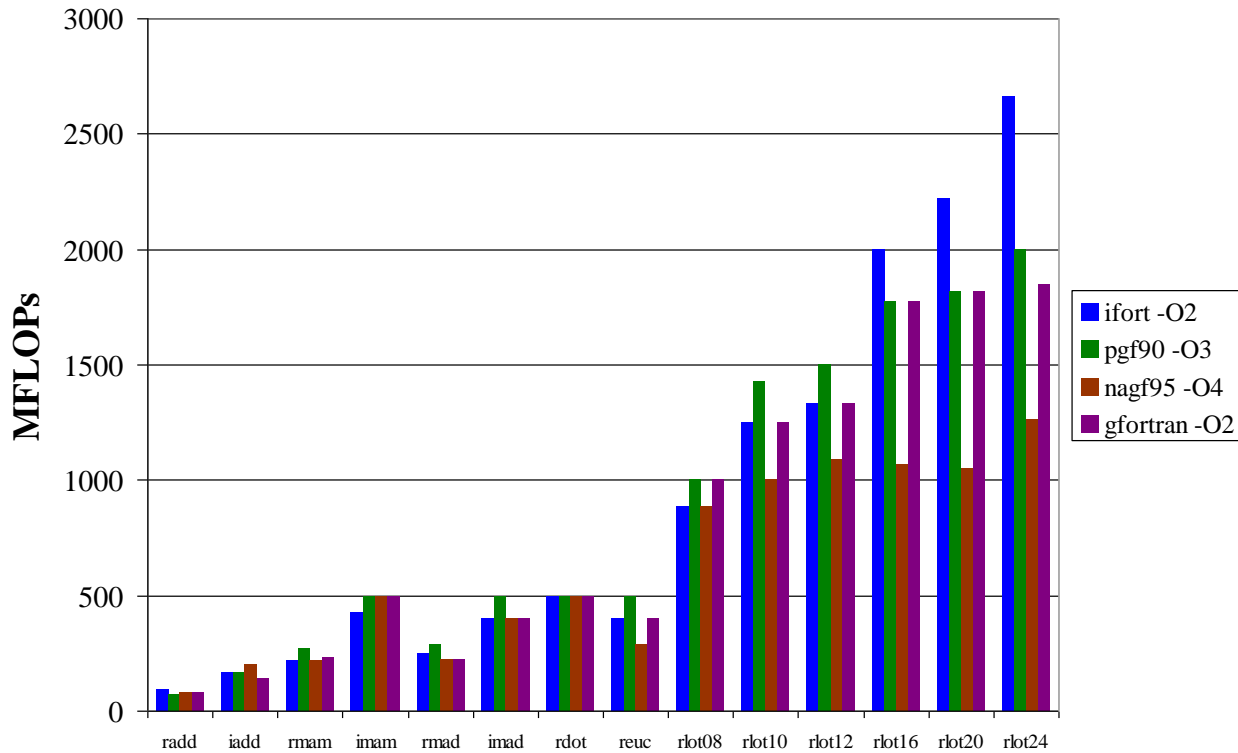
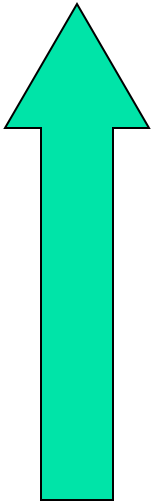




Multiple Ops Per Iteration

Arithmetic Performance: Multiple Operations
(Irwindale 3.2 GHz)

Better





Vectors



What Is a Vector?

A *vector* is a giant register that behaves like a collection of regular registers, except these registers all simultaneously perform the same operation on multiple sets of operands, producing multiple results.

In a sense, vectors are like operation-specific cache.

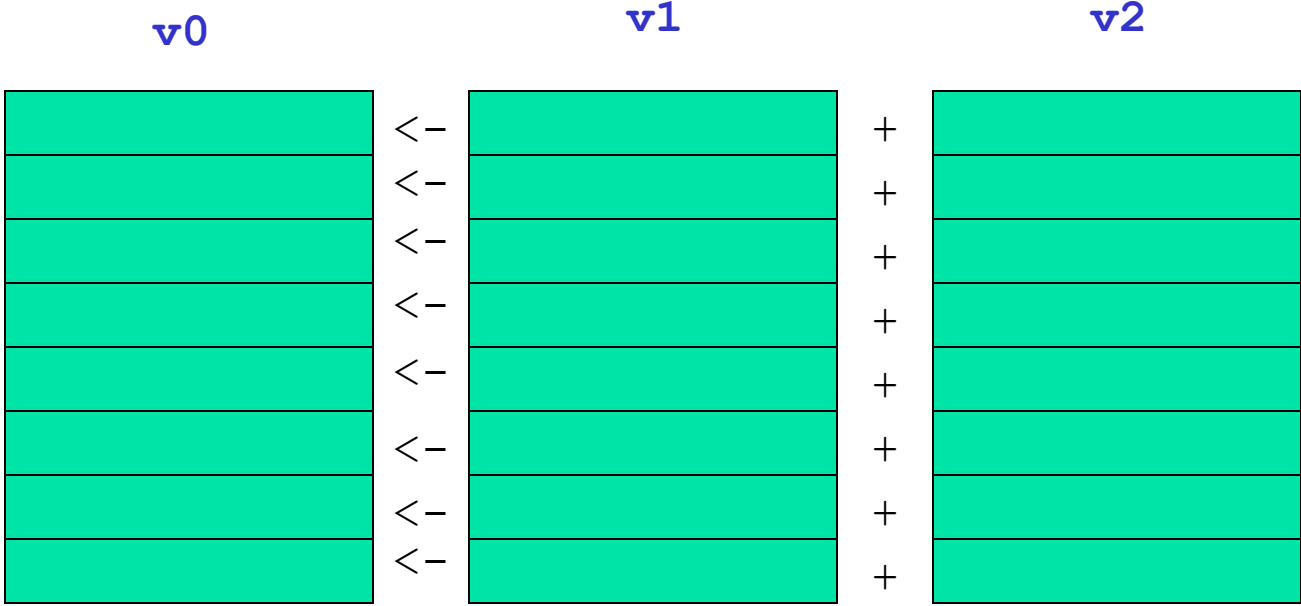
A *vector register* is a register that's actually made up of many individual registers.

A *vector instruction* is an instruction that performs the same operation simultaneously on all of the individual registers of a vector register.

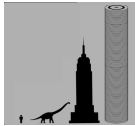




Vector Register



$$v0 \leftarrow v1 + v2$$





Vectors Are Expensive

Vectors were very popular in the 1980s, because they're very fast, often faster than pipelines.

In the 1990s, though, they weren't very popular. Why?

Well, vectors aren't used by many commercial codes (for example, MS Word). So most chip makers didn't bother with vectors.

So, if you wanted vectors, you had to pay a lot of extra money for them.

The Pentium III Intel reintroduced very small integer vectors (2 operations at a time),. The Pentium4 added floating point vector operations, also of size 2. The Core family has doubled the vector size to 4, and Sandy Bridge (2011) to 8.



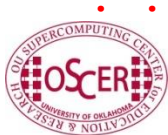
A Real Example





A Real Example^[4]

```
DO k=2,nz-1
  DO j=2,ny-1
    DO i=2,nx-1
      tem1(i,j,k) = u(i,j,k,2) * (u(i+1,j,k,2) - u(i-1,j,k,2)) * dxinv2
      tem2(i,j,k) = v(i,j,k,2) * (u(i,j+1,k,2) - u(i,j-1,k,2)) * dyinv2
      tem3(i,j,k) = w(i,j,k,2) * (u(i,j,k+1,2) - u(i,j,k-1,2)) * dzinv2
    END DO
  END DO
END DO
DO k=2,nz-1
  DO j=2,ny-1
    DO i=2,nx-1
      u(i,j,k,3) = u(i,j,k,1) -      &
& dtbig2 * (tem1(i,j,k) + tem2(i,j,k) + tem3(i,j,k))
    END DO
  END DO
END DO
```

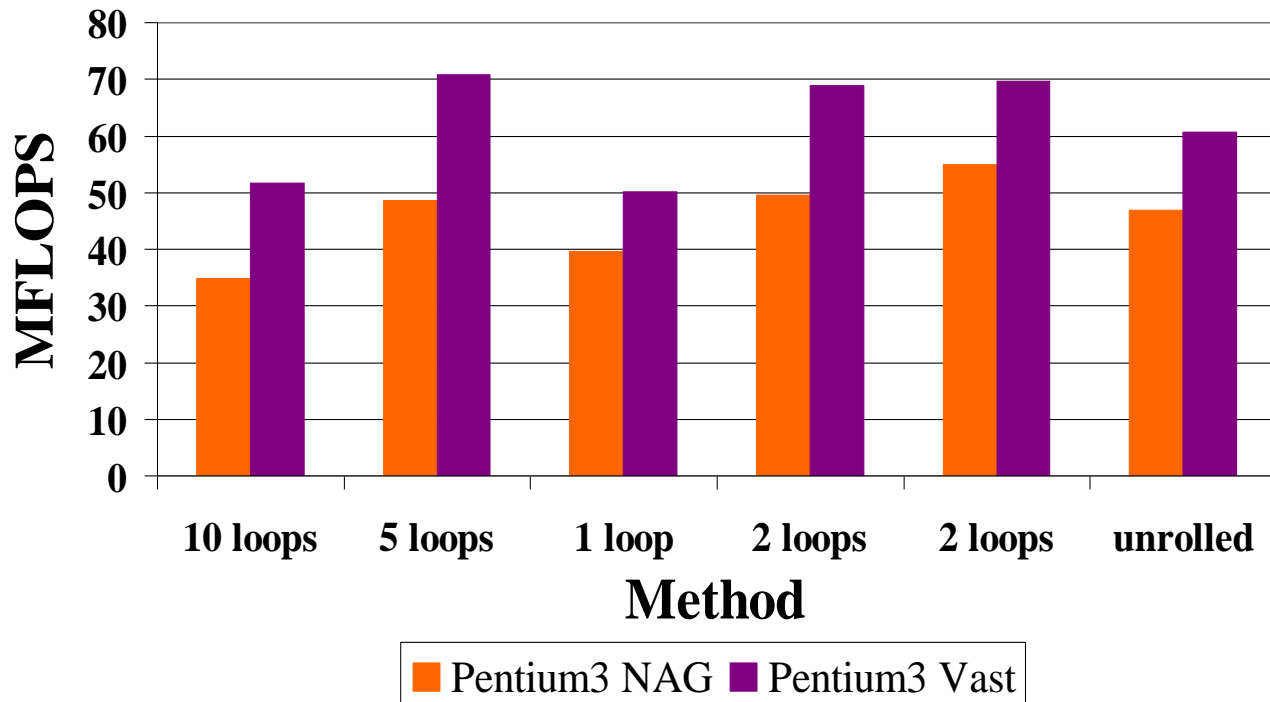
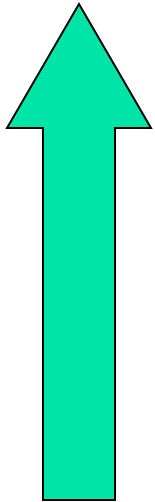




Real Example Performance

Performance By Method

Better





DON'T PANIC!



Parallel Programming: Instr Level Parallel
OK Supercomputing Symposium, Tue Oct 11 2011





Why You Shouldn't Panic

In general, the compiler and the CPU will do most of the heavy lifting for instruction-level parallelism.

BUT:

You need to be aware of ILP, because how your code is structured affects how much ILP the compiler and the CPU can give you.



**Thanks for your
attention!**



Questions?



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.

