CS 105

"Tour of the Black Holes of Computing"

Machine-Dependent Optimization

Machine-Dependent Optimization

Need to understand the architecture

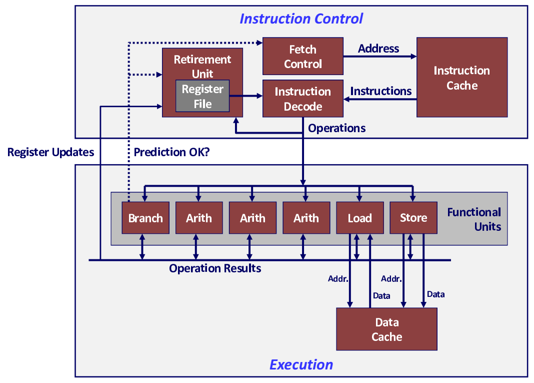
Not portable

Not often needed

…but critically important when it is

Also helps in understanding modern machines

Modern CPU Design



Superscalar Processor

Definition: A superscalar processor can issue and execute ***multiple instructions in one cycle***. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

Benefit: without programming effort, superscalar processor can take advantage of the ***instruction-level parallelism*** that most programs have

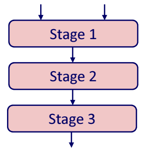
Most modern CPUs are superscalar.

Intel: since Pentium (1993)

What Is a Pipeline?

Mul
Result Bucket

Pipelined Functional Units



```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Haswell CPU

- 8 Total Functional Units

Multiple instructions can execute in parallel

2 load, with address computation
1 store, with address computation
4 integer
2 FP multiply
1 FP add
1 FP divide

Some instructions take > 1 cycle, but can be pipelined

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	3-30	3-30
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	3-15	3-15

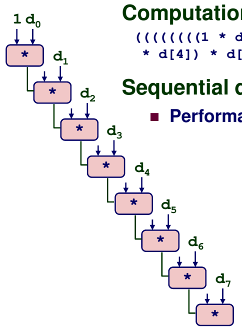
x86-64 Compilation of Combine4

Inner Loop (Case: Integer Multiply)

```
.L519:                          # Loop:
  imull  (%rax,%rdx,4), %ecx    # t = t * d[i]
  addq   $1, %rdx               # i++
  cmpq   %rdx, %rbp             # Compare length:i
  jg     .L519                  # If >, goto Loop
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

Combine4 = Serial Computation (OP = *)



Computation (length=8)

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

Sequential dependence

- Performance: determined by latency of OP

Loop Unrolling (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

```
x = (x OP d[i]) OP d[i+1];
```

Helps integer add by reducing number of overhead instructions

- (Almost) achieves latency bound

Others don't improve. *Why?*

- Still sequential dependency

Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

Can this change result of computation?

Yes, for multiply and floating point. *Why?*

Effect of Reassociation

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Nearly 2x speedup for Int *, FP +, FP *

- Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

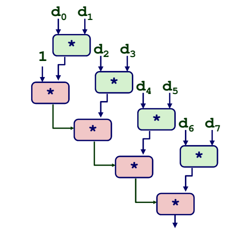
- Why is that? (next slide)

2 functional units for FP *
2 functional units for load

4 functional units for int +
2 functional units for load

Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```



What changed:

- Operations in the next iteration can be started early (no dependency)

Overall Performance

- N elements, D cycles latency/op
- (N/2+1)*D cycles:
  CPE = D/2

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

Different form of reassociation

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	0.81	1.51	1.51	2.51
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

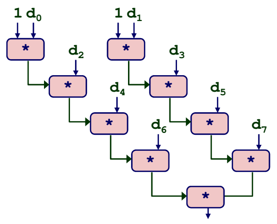
Int + makes use of two load units

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

2x speedup (over unroll2) for Int *, FP +, FP *

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```



- **What changed:**
  - Two independent “streams” of operations
- **Overall Performance**
  - N elements, D cycles latency/operation
  - Should be (N/2+1)*D cycles:
    CPE = D/2
  - CPE matches prediction!

What Now?

Unrolling & Accumulating

Idea
- Can unroll to any degree L
- Can accumulate K results in parallel
- L must be multiple of K

Limitations
- Diminishing returns
  - Cannot go beyond throughput limitations of execution units
- May run out of registers for accumulators
- Large overhead for short lengths
  - Finish off iterations sequentially

Unrolling & Accumulating: Double *

Case
- Intel Haswell
- Double FP Multiplication
- Latency bound: 5.00. Throughput bound: 0.50

Number of Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Unrolling & Accumulating: Int +

Case
- Intel Haswell
- Integer addition
- Latency bound: 1.00. Throughput bound: 1.00

Number of Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Limited only by throughput of functional units

Up to 42X improvement over original, unoptimized code

What About Branches?

Challenge

- Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

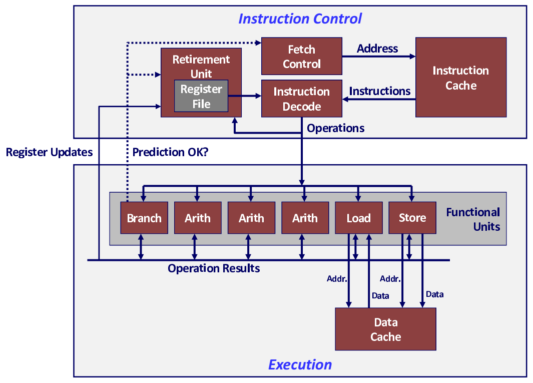
```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

  . . .

404685:  repz retq
```

Executing

How to continue?

- When encounters conditional branch, cannot reliably determine where to continue fetching

Modern CPU Design



Branch Outcomes

- When encounter conditional branch, cannot determine where to continue fetching
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- Cannot resolve until outcome determined by branch/integer unit

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

  . . .

404685:  repz retq
```

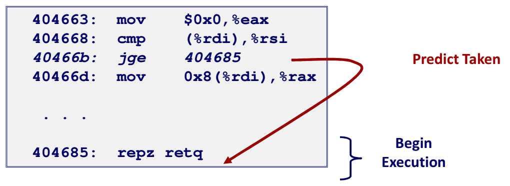
Branch Not-Taken

Branch Taken

Branch Prediction

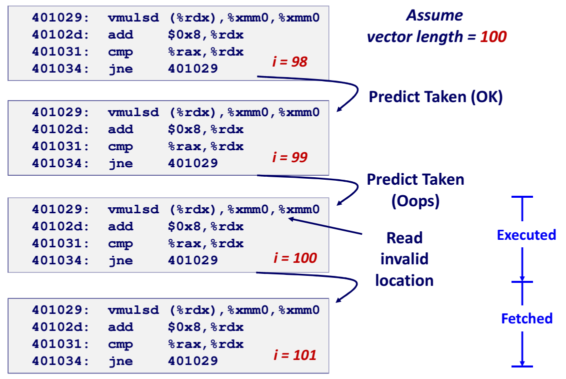
Idea

- Guess which way branch will go
- Begin executing instructions at predicted position
  - But don't actually modify register or memory data



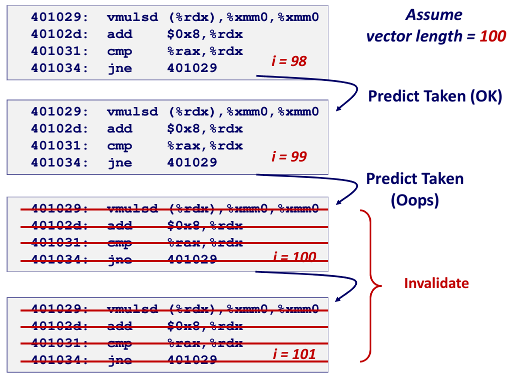
```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

  . . .

404685:  repz retq
```

Branch Prediction Through Loop



```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029            i = 98

401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029            i = 99

401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029            i = 100

401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029            i = 101
```

Branch Misprediction Invalidation

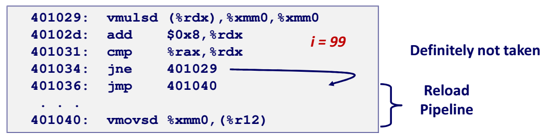


```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029            i = 98

401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029            i = 99
```

~~401029: vmulsd (%rdx),%xmm0,%xmm0~~
~~40102d: add $0x8,%rdx~~
~~401031: cmp %rax,%rdx~~
~~401034: jne 401029 i = 100~~

~~401029: vmulsd (%rdx),%xmm0,%xmm0~~
~~40102d: add $0x8,%rdx~~
~~401031: cmp %rax,%rdx~~
~~401034: jne 401029 i = 101~~

Branch Misprediction Recovery



```
401029:  vmulsd  (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx            i = 99
401031:  cmp     %rax,%rdx
401034:  jne     401029
401036:  jmp     401040
  . . .
401040:  vmovsd  %xmm0,(%r12)
```

Performance Cost

- Multiple clock cycles on modern processor
- Can be a major performance limiter
- Current CPUs (2019) speculate *150 or more* instructions ahead!

Getting High Performance

Use a good compiler and appropriate flags

Don't do anything stupid

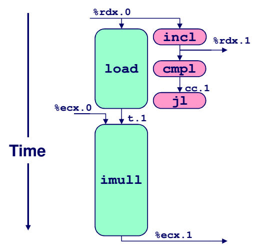
- Watch out for hidden algorithmic inefficiencies
- Write compiler-friendly code
  - Watch out for optimization blockers: procedure calls & memory references
- Look carefully at innermost loops (where most work is done)

Tune code for machine

- Exploit instruction-level parallelism
- Avoid unpredictable branches
- Make code cache-friendly

But DON'T OPTIMIZE UNTIL IT'S DEBUGGED!!!

Visualizing Operations



```
load (%rax,%rdx.0,4)➔ t.1
imull t.1, %ecx.0    ➔ %ecx.1
incl %rdx.0          ➔ %rdx.1
cmpl %rsi, %rdx.1    ➔ cc.1
jl-taken cc.1
```

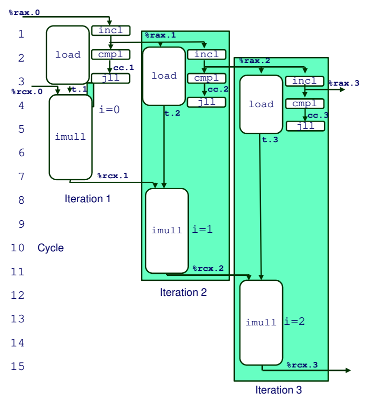
Operations

- Vertical position denotes time at which executed
  - Cannot begin operation until operands available
- Height denotes latency

Operands

- Arcs shown only for operands that are passed within execution unit

3 Iterations of Combining Product



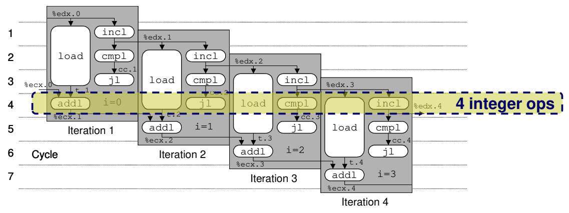
Unlimited-Resource Analysis

- Assume operation can start as soon as operands available
- Operations for multiple iterations overlap in time

Performance

- Limiting factor becomes latency of integer multiplier
- Gives CPE of 4.0

4 Iterations of Combining Sum

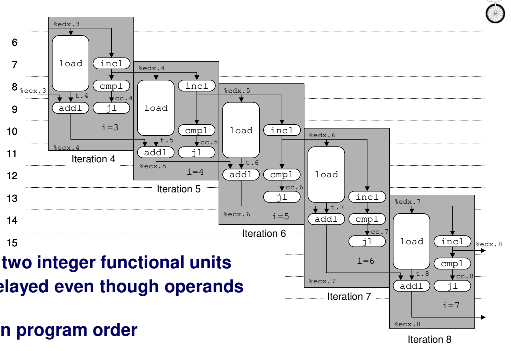


Unlimited-Resource Analysis

Performance

- Can begin a new iteration on each clock cycle
- Should give CPE of 1.0
- Would require executing 4 integer operations in parallel

Combining Sum: Resource Constraints



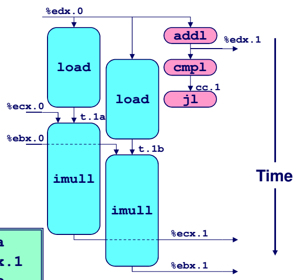
- Suppose only have two integer functional units
- Some operations delayed even though operands available
- Set priority based on program order

Performance

- Sustains CPE of 2.0

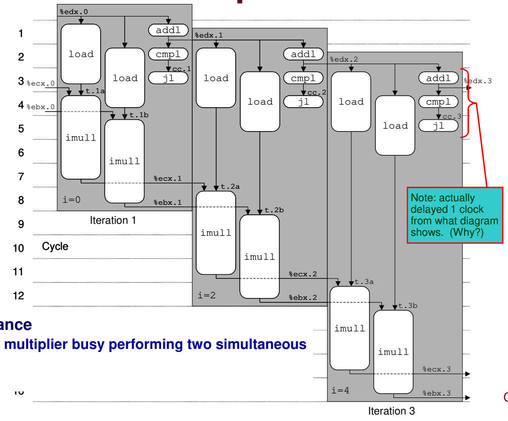
Visualizing Parallel Loop

- Two multiplies within loop no longer have data dependency
- Allows them to pipeline



```
load (%eax,%edx.0,4)  → t.1a
imull t.1a, %ecx.0    → %ecx.1
load 4(%eax,%edx.0,4) → t.1b
imull t.1b, %ebx.0    → %ebx.1
iaddl $2,%edx.0       → %edx.1
cmpl %esi, %edx.1     → cc.1
jl-taken cc.1
```

Executing with Parallel Loop



Iteration 1
Iteration 3
Cycle
Note: actually delayed 1 clock from what diagram shows. (Why?)

- Predicted Performance
  - Can keep 4-cycle multiplier busy performing two simultaneous multiplications
  - Gives CPE of 2.0

Meltdown and Spectre

Consider a few things

- Access to cached things is *much* faster than to non-cached ones
- Programs have access to detailed timing information
  - Intel offers free-running cycle counter to all programs
  - Thus, can tell whether something was cached
- OS has access to everything
  - Carefully checks whether *you* have access before giving stuff to you
- CPU speculates many instructions ahead
  - Must guess about branch directions
- User programs can either flush cache (`clflush` instruction) or clobber with loop

Meltdown and Spectre

Trick OS into doing these steps:

- Check whether you have access to arbitrary location *x* (you don't)
- Mispredict that branch
- Read location *x* and use its contents as follows:
  - Extract bit *b*
  - Multiply (shift left) bit *b* by, e.g., 1024
  - Access array *y[b*1024]* that you *do* have access to
- Hardware will eventually discover mispredicted branch and cancel all those instructions
  - …but cache now contains *y[b*1024]*

Scan cache to see whether *y[0]* or *y[1024]* is fast (i.e., in cache)

- You now know bit *b* of location *x*
- Lather, rinse, repeat until you know all bits of *x*
- Lather, rinse, repeat for all locations you want to read

So What?

Can read arbitrary memory at about 2K bits/second

- No biggie on your laptop
- Huge issue in the cloud
  - Physical machines often shared
  - Supposedly isolated by virtual-machine technology
- Grab people's encryption keys, passwords, all sorts of stuff
- Next stop: Putin

What to do?

- Disabling speculation kills performance
- Only certain branches are vulnerable
  - Can do special things for those branches
  - But hard to find (millions of lines in kernel)
- Compiler can try to identify risky branches
  - But will be conservative → OS will slow down