Análise do desempenho de execução de aplicações *(2)*
(adaptado das aulas 10 e 11 do Bryant)

- "*Análise de desempenho*": para quê?
  - ...
    - $CPU_{Time} = Nº_{instr} * CPI * T_{clock}$
  - análise de técnicas de optimização
    - independentes da máquina
      - *code motion* (ineficiências de *loops* e de funções,...)
      - *strength reduction* (op's +simples, evitar *mem ref*, ...)
      - partilha de sub-expressões
      - atenção aos bloqueadores de optimização de compiladores!!
    - dependentes da máquina
      - análise sucinta de um CPU (+*cache*) actual (Pentium III/IV)
      - identificação de potenciais limitadores de desempenho
      - *loop unroll* e *inline functions*
      - influência da memória...

Os próximos slides foram retirados da aula do Prof. Bryant

15-213

"The course that gives CMU its Zip!"

Code Optimization I: Machine Independent Optimizations
Sept. 26, 2002

- Topics
  - Machine-Independent Optimizations
    - Code motion
    - Reduction in strength
    - Common subexpression sharing
  - Tuning
    - Identifying performance bottlenecks

Optimization Example

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Procedure
  - Compute sum of all elements of integer vector
  - Store result at destination location
  - Vector data structure and operations defined via abstract data type
- Pentium II/III Performance: Clock Cycles / Element
  - 42.06 (Compiled -g)   31.25 (Compiled -O2)

Understanding Loop

```
void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v))
      goto done;
  loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
      goto loop
  done:
}
```

1 iteration

- Inefficiency
  - Procedure vec_length called every iteration
  - Even though result always the same

Move `vec_length` Call Out of Loop

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Optimization
  - Move call to `vec_length` out of inner loop
    - Value does not change from one iteration to next
    - Code motion
  - CPE: 20.66 (Compiled -O2)
    - `vec_length` requires only constant time, but significant overhead

Optimization Blocker: Procedure Calls

- *Why couldn't the compiler move `vec_len` out of the inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
- *Why doesn't compiler look at code for `vec_len`?*
  - Linker may overload with different version
    - Unless declared static
  - Interprocedural optimization is not used extensively due to cost
- Warning:
  - Compiler treats procedure call as a black box
  - Weak optimizations in and around them

Reduction in Strength

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

- Optimization
  - Avoid procedure call to retrieve each vector element
    - Get pointer to start of array before loop
    - Within loop just do pointer reference
    - Not as clean in terms of data abstraction
  - CPE: 6.00 (Compiled -O2)
    - Procedure calls are expensive!
    - Bounds checking is expensive

Eliminate Unneeded Memory Refs

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

- Optimization
  - Don't need to store in destination until end
  - Local variable `sum` held in register
  - Avoids 1 memory read, 1 memory write per cycle
  - CPE: 2.00 (Compiled -O2)
    - Memory references are expensive!

Detecting Unneeded Memory Refs.

Combine3

```
.L18:
       movl  (%ecx,%edx,4),%eax
       addl  %eax,(%edi)
       incl  %edx
       cmpl  %esi,%edx
       jl  .L18
```

Combine4

```
.L24:
       addl  (%eax,%edx,4),%ecx

       incl  %edx
       cmpl  %esi,%edx
       jl  .L24
```

- Performance
  - Combine3
    - 5 instructions in 6 clock cycles
    - `addl` must read and write memory
  - Combine4
    - 4 instructions in 2 clock cycles

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references specify single location
- Example
  - `v: [3, 2, 17]`
  - `combine3(v, get_vec_start(v)+2)    -->   ?`
  - `combine4(v, get_vec_start(v)+2)    -->   ?`
- Observations
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Os próximos slides foram retirados da aula do Prof. Bryant

15-213

"The course that gives CMU its Zip!"

Code Optimization II:
Machine Dependent Optimizations
Oct. 1, 2002

Topics
- Machine-Dependent Optimizations
  - Pointer code
  - Unrolling
  - Enabling instruction level parallelism
- Understanding Processor Operation
  - Translation of instructions into operations
  - Out-of-order execution of operations
- Branches and Branch Prediction
- Advice

General Forms of Combining

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP data[i];
  *dest = t;
}
```

- Data Types
  - Use different declarations for `data_t`
  - `int`
  - `float`
  - `double`
- Operations
  - Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`

Machine Independent Opt. Results

Optimizations

- Reduce function calls and memory references within loop

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
data access	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00

Performance Anomaly

- Computing FP product of all elements exceptionally slow.
- Very large speedup when accumulate in temporary
- Caused by quirk of IA32 floating point
  - Memory uses 64-bit format, register use 80
  - Benchmark data caused overflow of 64 bits, but not 80

Pointer Code

```
void combine4p(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int *dend = data+length;
  int sum = 0;
  while (data < dend) {
    sum += *data;
    data++;
  }
  *dest = sum;
}
```

Optimization

- Use pointers rather than array references
- CPE: 3.00 (Compiled -O2)
  - Oops! We're not making progress here!

Warning: Some compilers do better job optimizing array code

Pointer vs. Array Code Inner Loops

Array Code

```
.L24:                          # Loop:
  addl (%eax,%edx,4),%ecx      # sum += data[i]
  incl %edx                    # i++
  cmpl %esi,%edx               # i:length
  jl .L24                      # if < goto Loop
```

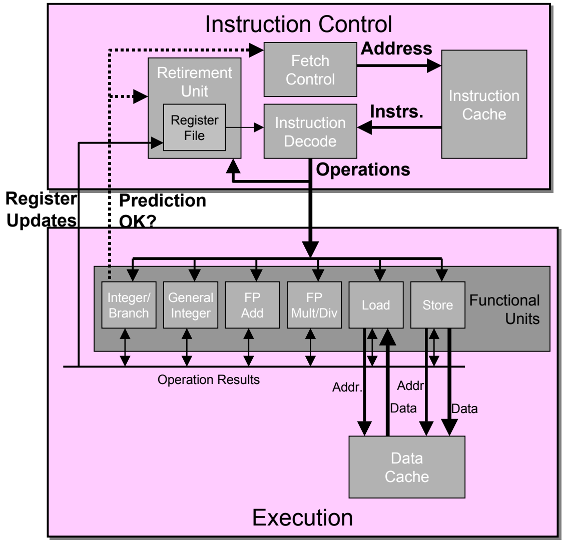
Pointer Code

```
.L30:                    # Loop:
  addl (%eax),%ecx       #  sum += *data
  addl $4,%eax           #  data ++
  cmpl %edx,%eax         #  data:dend
  jb .L30                #  if < goto Loop
```

Performance

- Array Code: 4 instructions in 2 clock cycles
- Pointer Code: Almost same 4 instructions in 3 clock cycles

Modern CPU-chip Design (Pentium III)

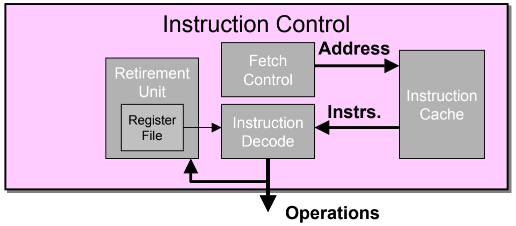


CPU Capabilities of Pentium III

- Multiple Instructions Can Execute in Parallel
  - 1 load
  - 1 store
  - 2 integer (one may be branch)
  - 1 FP Addition
  - 1 FP Multiplication or Division
- Some Instructions Take > 1 Cycle, but Can be Pipelined

– Instruction	Latency	Cycles/Issue
– Load / Store	3	1
– Integer Multiply	4	1
– Integer Divide	36	36
– Double/Single FP Multiply	5	2
– Double/Single FP Add	3	1
– Double/Single FP Divide	38	38

Instruction Control



- Grabs Instruction Bytes From Memory
  - Based on current PC + predicted targets for predicted branches
  - Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target
- Translates Instructions Into *Operations*
  - Primitive steps required to perform instruction
  - Typical instruction requires 1–3 operations
- Converts Register References Into *Tags*
  - Abstract identifier linking destination of one operation with sources of later operations

---

Translation Example

- Version of Combine4
  - Integer data, multiply operation

```
.L24:                         # Loop:
  imull (%eax,%edx,4),%ecx    # t *= data[i]
  incl %edx                   # i++
  cmpl %esi,%edx              # i:length
  jl .L24                     # if < goto Loop
```

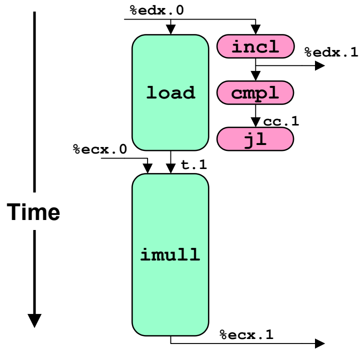
- Translation of First Iteration

```
.L24:
  imull (%eax,%edx,4),%ecx

  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```
load (%eax,%edx.0,4)  ➔ t.1
imull t.1, %ecx.0     ➔ %ecx.1
incl %edx.0           ➔ %edx.1
cmpl %esi, %edx.1     ➔ cc.1
jl-taken cc.1
```

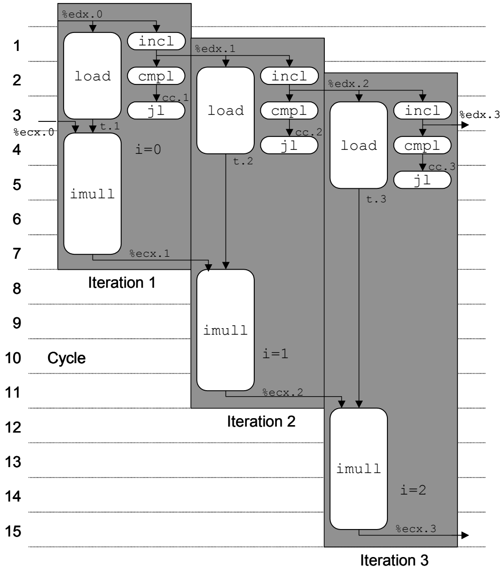
Visualizing Operations



```
load (%eax,%edx,4)  ➔ t.1
imull t.1, %ecx.0   ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.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

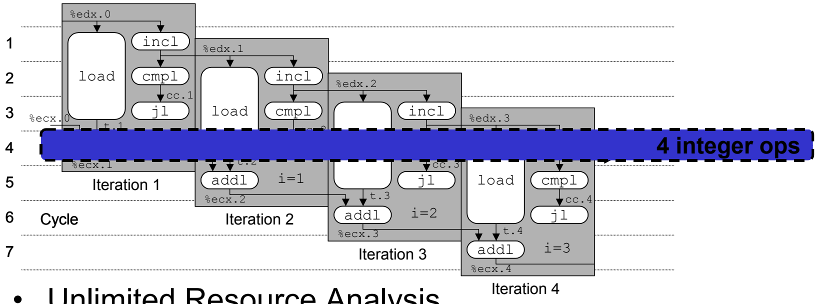
3 Iterations of Combining Product



- 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

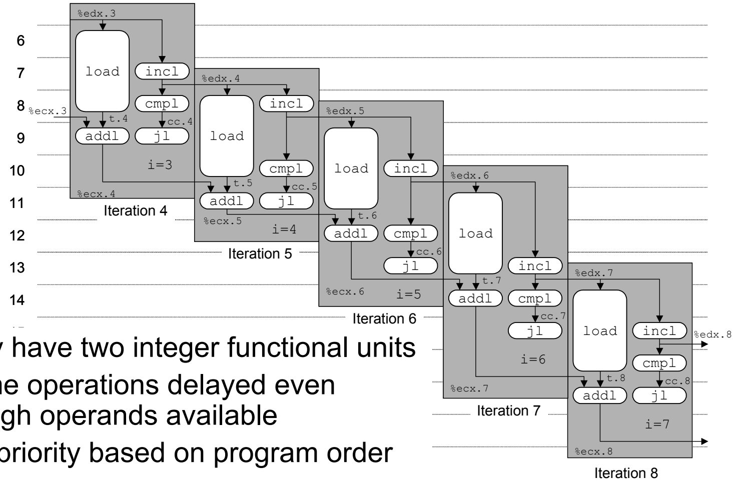
---

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



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

- Performance
  - Sustain CPE of 2.0

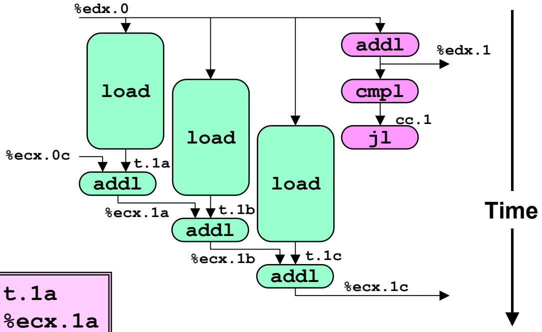
Loop Unrolling

```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* Combine 3 elements at a time */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+2]
           + data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    sum += data[i];
  }
  *dest = sum;
}
```

- Optimization
  - Combine multiple iterations into single loop body
  - Amortizes loop overhead across multiple iterations
  - Finish extras at end
  - Measured CPE = 1.33

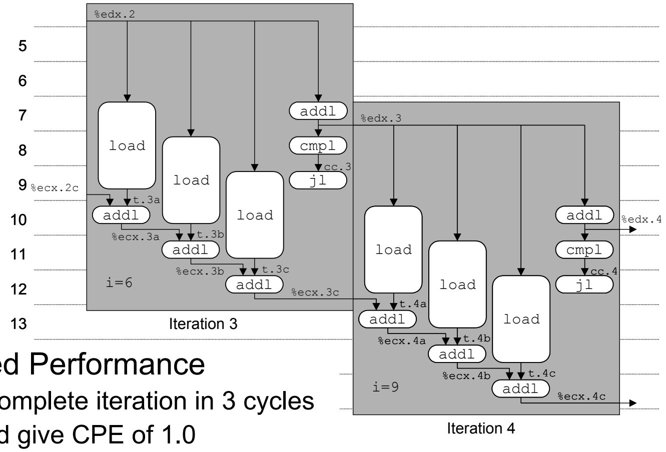
Visualizing Unrolled Loop

- Loads can pipeline, since don’t have dependencies
- Only one set of loop control operations



```
load (%eax,%edx.0,4)   → t.1a
iaddl t.1a, %ecx.0c    → %ecx.1a
load 4(%eax,%edx.0,4)  → t.1b
iaddl t.1b, %ecx.1a    → %ecx.1b
load 8(%eax,%edx.0,4)  → t.1c
iaddl t.1c, %ecx.1b    → %ecx.1c
iaddl $3,%edx.0        → %edx.1
cmpl %esi, %edx.1      → cc.1
jl-taken cc.1
```

Executing with Loop Unrolling



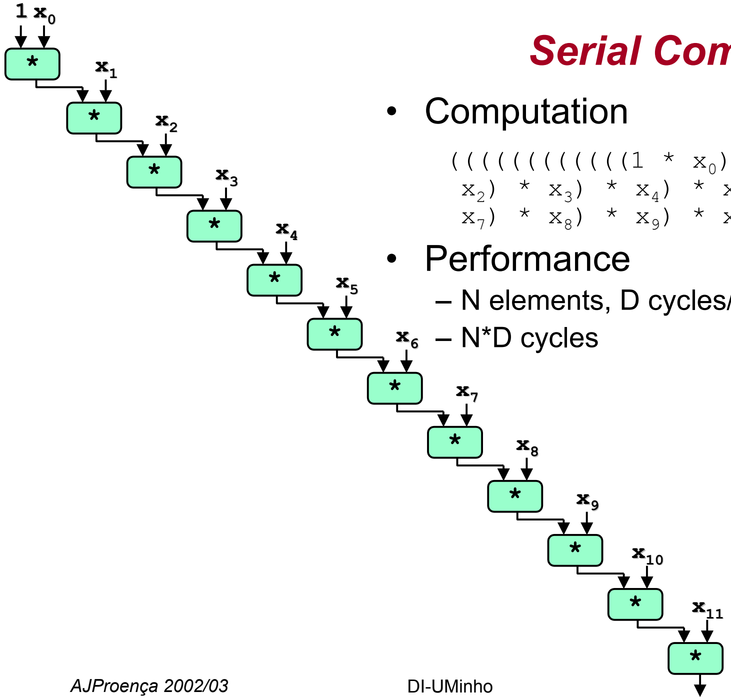
- Predicted Performance
  - Can complete iteration in 3 cycles
  - Should give CPE of 1.0
- Measured Performance
  - CPE of 1.33
  - One iteration every 4 cycles

Effect of Unrolling

Unrolling Degree		1	2	3	4	8	16
Integer	Sum	2.00	1.50	1.33	1.50	1.25	1.06
Integer	Product	4.00					
FP	Sum	3.00					
FP	Product	5.00					

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is nonlinear with degree of unrolling
  - Many subtle effects determine exact scheduling of operations

---



Serial Computation

- Computation

$((((((((((((1 * x_0) * x_1) * x_2) * x_3) * x_4) * x_5) * x_6) * x_7) * x_8) * x_9) * x_{10}) * x_{11})$

- Performance
  - N elements, D cycles/operation
  - N*D cycles

Parallel Loop Unrolling

```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x0 = 1;
  int x1 = 1;
  int i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x0 *= data[i];
  }
  *dest = x0 * x1;
}
```

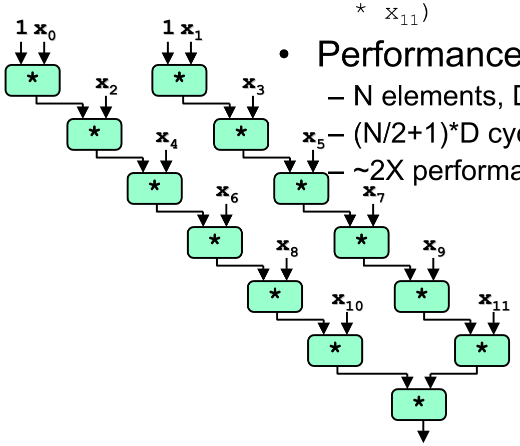
- Code Version
  - Integer product
- Optimization
  - Accumulate in two different products
    - Can be performed simultaneously
  - Combine at end
- Performance
  - CPE = 2.0
  - 2X performance

Dual Product Computation

- Computation

$((((((1 * x_0) * x_2) * x_4) * x_6) * x_8) * x_{10}) *$

$((((((1 * x_1) * x_3) * x_5) * x_7) * x_9) * x_{11})$



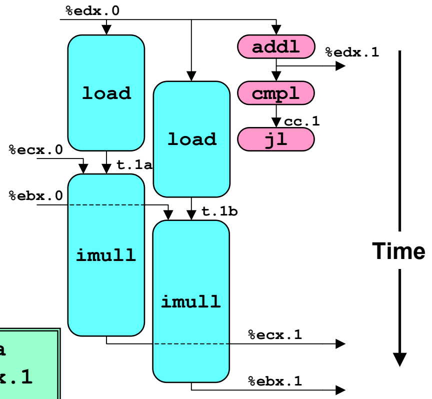
- Performance
  - N elements, D cycles/operation
  - (N/2+1)*D cycles
  - ~2X performance improvement

Requirements for Parallel Computation

- Mathematical
  - Combining operation must be associative & commutative
    - OK for integer multiplication
    - Not strictly true for floating point
      - OK for most applications
- Hardware
  - Pipelined functional units
  - Ability to dynamically extract parallelism from code

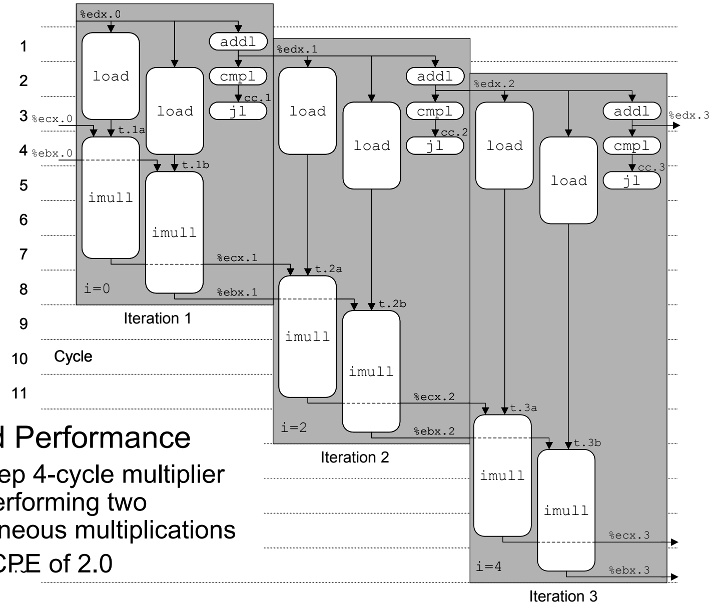
Visualizing Parallel Loop

- Two multiplies within loop no longer have data depency
- 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



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

Optimization Results for Combining

Method	**Integer**		**Floating Point**	
	+	*****	**+**	*****
Abstract -g	**42.06**	**41.86**	**41.44**	**160.00**
Abstract -O2	**31.25**	**33.25**	**31.25**	**143.00**
Move vec_length	**20.66**	**21.25**	**21.15**	**135.00**
data access	**6.00**	**9.00**	**8.00**	**117.00**
Accum. in temp	**2.00**	**4.00**	**3.00**	**5.00**
Pointer	**3.00**	**4.00**	**3.00**	**5.00**
Unroll 4	**1.50**	**4.00**	**3.00**	**5.00**
Unroll 16	***1.06***	**4.00**	**3.00**	**5.00**
2 X 2	**1.50**	**2.00**	**2.00**	**2.50**
4 X 4	**1.50**	***2.00***	***1.50***	**2.50**
8 X 4	**1.25**	***1.25***	**1.50**	***2.00***
Theoretical Opt.	**1.00**	**1.00**	**1.00**	**2.00**
Worst : Best	***39.7***	***33.5***	***27.6***	***80.0***

Limitations of Parallel Execution

- Need Lots of Registers
  - To hold sums/products
  - Only 6 usable integer registers
    - Also needed for pointers, loop conditions
  - 8 FP registers
  - When not enough registers, must spill temporaries onto stack
    - Wipes out any performance gains
  - Not helped by renaming
    - Cannot reference more operands than instruction set allows
    - Major drawback of IA32 instruction set

Register Spilling Example

- Example
  - 8 X 8 integer product
  - 7 local variables share 1 register
  - See that are storing locals on stack
  - E.g., at `-8(%ebp)`

```
.L165:
      imull (%eax),%ecx
      movl -4(%ebp),%edi
      imull 4(%eax),%edi
      movl %edi,-4(%ebp)
      movl -8(%ebp),%edi
      imull 8(%eax),%edi
      movl %edi,-8(%ebp)
      movl -12(%ebp),%edi
      imull 12(%eax),%edi
      movl %edi,-12(%ebp)
      movl -16(%ebp),%edi
      imull 16(%eax),%edi
      movl %edi,-16(%ebp)
   …
      addl $32,%eax
      addl $8,%edx
      cmpl -32(%ebp),%edx
      jl .L165
```

Machine-Dependent Opt. Summary

- Pointer Code
  - Look carefully at generated code to see whether helpful
- Loop Unrolling
  - Some compilers do this automatically
  - Generally not as clever as what can achieve by hand
- Exposing Instruction-Level Parallelism
  - Very machine dependent
- Warning:
  - Benefits depend heavily on particular machine
  - Best if performed by compiler
    - But GCC on IA32/Linux is not very good
  - Do only for performance-critical parts of code

Role of Programmer

How should I write my programs, given that I have a good, optimizing compiler?

- Don't: Smash Code into Oblivion
  - Hard to read, maintain, & assure correctness
- Do:
  - Select best algorithm
  - Write code that's readable & maintainable
    - Procedures, recursion, without built-in constant limits
    - Even though these factors can slow down code
  - Eliminate optimization blockers
    - Allows compiler to do its job
- Focus on Inner Loops
  - Do detailed optimizations where code will be executed repeatedly
  - Will get most performance gain here