General Forms of Combining

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

**Data Types**
- Use different declarations for `data_t`
- Use different definitions of `OP` and `IDEAL`
- `int`
- `float`
- `double`

**Operations**
- Use different definitions of `OP` and `IDEAL`
- `OP = + / 0`
- `IDEAL = * / 1`

**Machine Independent Opt. Results**

**Optimizations**
- Reduce function calls and memory references within loop
- 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 fmmul, register use 80
  - Benchmark data caused overflow of 64 bits, but no 80
**Pointer Code**

```c
void combine(vvec_ptr v, int *dest)
{
    int length = vvec_length(v);
    int *data = get_vvec_start(v);
    int *dend = data + length;
    int sum = 0;
    while (data < dend) {
        *dest += *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**

```assembly
L24: # Loop:
    addl (%eax, %edx, 4), %exx # sum += data[i]
    incl %edx
    cmpl %eax, %edx # i < length
j1 L24 # if < goto Loop
```

**Pointer Code**

```assembly
L30: # Loop:
    addl (%eax, %exx) # sum += *data
    addl $4, %eax # data +=
    cmpl %eax, %data: %dend
    j1 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 Design**

**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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>Double/Single FP Multiply</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Double/Single FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Double/Single FP Divide</td>
<td>38</td>
<td>38</td>
</tr>
</tbody>
</table>
**Instruction Control**

- **Grabs Instruction Bytes From Memory**
  - Based on current PC, predicted targets for predicted branches
  - Hardware dynamically guesses whether branches taken/untaken 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 other operations

---

**Translation Example**

**Version of Combine4**
- Integer data, multiply operation

```plaintext
L24:
  imull (teax, tedx, 4), tecx # Loop:
  t. t. = data[i]
  incl tecx
  cmpl tesi, tedx
  j1 L24
```

**Translation of First Iteration**

```
L24:
imull (teax, tedx, 4), tecx
incl tecx
cmpl tesi, teda
j1 L24
```

**Translation Example #1**
- `imull (teax, tedx, 4), tecx`
  - Split into two operations
    - `load (teax, tedx, 4)` to generate temporary result `t.1`
    - Multiplication operation just operates on registers
  - Operands
  - Registers `teax` does not change in loop. Values will be retrieved from register file during decoding
  - Register `tedx` changes on every iteration. Uniquely identify different versions as `tedx.0, teda.1, teda.2, ...`
  - Register renaming
  - Values passed directly from producer to consumers

**Translation Example #2**
- `incl teda`
  - `incl teda.0` to `teda.1`
  - Register `teda` changes on each iteration. Rename as `teda.0, teda.1, teda.2, ...`
Translation Example #3

\[ \text{cmpl test1, tedi} \rightarrow \text{cc.1} \]

- Condition codes are treated similar to registers
- Assign tag to define connection between producer and consumer

Translation Example #4

\[ \text{jl} \rightarrow \text{cc.1} \]

- Instruction control unit determines destination of jump
- Predicts whether will be taken and target
- Stalls fetching instruction at predicted destination
- Execution unit simply checks whether or not prediction was OK
- If not, it signals instruction control
- Instruction control then “invalidates” any operations generated from mispredicted instructions
- Begins fetching and decoding instructions at correct target

Visualizing Operations

Operations
- Vertical position denotes time at which executed
- Cannot begin operation until operand is available
- Height denotes latency

Operands
- Arrows shown only for operands that are passed within execution unit

Visualizing Operations (cont.)

Operations
- Same as before, except that add1 has latency of 1

\[ \text{load (test, tedi, 4)} \rightarrow \text{t.1} \]
\[ \text{add1 t.1, tedi, 0} \rightarrow \text{test1} \]
\[ \text{load tedi, 0} \rightarrow \text{tedi.1} \]
\[ \text{cmpl test1, tedi.1} \rightarrow \text{cc.1} \]
\[ \text{jl-taken cc.1} \]
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
- Limited 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

Combinig 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

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

```c
void combined(unsigned 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++) {
        sum += data[i] + data[i<<2] + data[i<<3];
    }
    // Finish any remaining elements /*
    for (; i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
```
**Visualizing Unrolled Loop**

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

<table>
<thead>
<tr>
<th>Unrolling Degree</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Sum</td>
<td>2.00</td>
<td>1.50</td>
<td>1.33</td>
<td>1.50</td>
<td>1.25</td>
<td>1.06</td>
</tr>
<tr>
<td>Integer Product</td>
<td>4.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Sum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP Product</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**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**

**Serial Computation**

- Computation
  \[ (x_1 + x_2) \times (x_3 + x_4) \]

- Performance
  - \( N \) elements, \( D \) cycles/operation
  - \( N \times D \) cycles
void combined(void_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 < limit; i++) {
        x0 *= data[i];
    }
    *dest = x0 * x1;
}

---

### Parallel Loop Unrolling

**Code Version**

- **Integer product**

**Optimization**

- **Accumulate in two different products**
- **Can be performed simultaneously**
- **Combine as end**

**Performance**

- **CPE = 2.0**
- **2x performance**

---

### Dual Product Computation

**Computation**

\[
((x_0 \times x_1) \times (x_2 \times x_3)) \times (x_4 \times x_5) 
\]

**Performance**

- **N element, D operation**
- **(N+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 dependency**

**Allows them to pipeline**

---
Executing with Parallel Loop

Optimization Results for Combining

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract-e,g</td>
<td>42.06</td>
<td>41.66</td>
</tr>
<tr>
<td>Abstract-Q2</td>
<td>31.25</td>
<td>33.25</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>20.66</td>
<td>21.25</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Pointer</td>
<td>3.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.50</td>
<td>1.25</td>
</tr>
<tr>
<td>2 * 2</td>
<td>1.50</td>
<td>2.00</td>
</tr>
<tr>
<td>4 * 4</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td>Theoretical Opt.</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td>Worst / Best</td>
<td>38.7</td>
<td>33.2</td>
</tr>
</tbody>
</table>

Parallel Unrolling: Method #2

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

Code Version
* Integer product

Optimization
* Multiply pairs of elements together
* And then update product
* "Tree height reduction"

Performance
* CPE = 2.5
```

Method #2 Computation

```
CCCCC * (a1 * a1) * (a2 * a2) * (a3 * a3)

Performance
* N elements, D cycles per operation
* Should be (N+1)D cycles
* CPE = 2.0
* Measured CPE worse

<table>
<thead>
<tr>
<th>Unrolling</th>
<th>CPE (measured)</th>
<th>CPE (theoretical)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.50</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>1.67</td>
<td>1.33</td>
</tr>
<tr>
<td>4</td>
<td>1.50</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>1.78</td>
<td>1.00</td>
</tr>
</tbody>
</table>
```
Understanding Parallelism

```c
// Combine 2 elements at a time
for (i = 0; i < limit; i++) {
    x = (x & data[i]) | data[i+1];
}
```
- CPE = 4.00
- All multiplies performed in sequence

```c
// Combine 2 elements at a time
for (i = 0; i < limit; i++) {
    x = x & (data[i] + data[i+1]);
}
```
- CPE = 2.50
- Multiplies overlap

---

Limitations of Parallel Execution

Need Lots of Registers
- To hold sums/products
- Only usable integer registers
- Also needed for pointers, loop conditions
- BFP 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. st - 8 (%ebp)

```
L165:
imull (team), kexec
movl -32(%ebp), %edi
imull 4(team), %edi
movl %edi, -4(%ebp)
movl -12(%ebp), %edi
imull 12(team), %edi
movl %edi, -12(%ebp)
movl -16(%ebp), %edi
imull 16(team), %edi
movl %edi, -16(%ebp)
```

Summary: Results for Pentium III

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>+</td>
</tr>
<tr>
<td>Ab same -g</td>
<td>42.06</td>
<td>41.86</td>
</tr>
<tr>
<td>Ab same -O2</td>
<td>31.25</td>
<td>33.25</td>
</tr>
<tr>
<td>Move wc_length</td>
<td>20.66</td>
<td>21.25</td>
</tr>
<tr>
<td>data access</td>
<td>6.00</td>
<td>9.00</td>
</tr>
<tr>
<td>Accum in temp</td>
<td>2.00</td>
<td>4.00</td>
</tr>
<tr>
<td>Ukmroll 4</td>
<td>1.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Ukmroll 16</td>
<td>1.06</td>
<td>4.00</td>
</tr>
<tr>
<td>4 x 2</td>
<td>1.50</td>
<td>2.00</td>
</tr>
<tr>
<td>8 x 4</td>
<td>1.25</td>
<td>1.25</td>
</tr>
<tr>
<td>8 x 8</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td>Word -best</td>
<td>35.07</td>
<td>33.33</td>
</tr>
</tbody>
</table>

- Biggest gain doing basic optimizations
- But, last little bit helps
Results for Alpha Processor

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>40.14</td>
<td>47.14</td>
</tr>
<tr>
<td>Abstract -Q2</td>
<td>25.08</td>
<td>36.05</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>19.19</td>
<td>32.18</td>
</tr>
<tr>
<td>data access</td>
<td>6.25</td>
<td>12.52</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>1.76</td>
<td>9.01</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.51</td>
<td>9.01</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.25</td>
<td>9.01</td>
</tr>
<tr>
<td>X 2</td>
<td>1.19</td>
<td>4.69</td>
</tr>
<tr>
<td>X 4</td>
<td>1.15</td>
<td>4.12</td>
</tr>
<tr>
<td>X 8</td>
<td>1.17</td>
<td>4.24</td>
</tr>
</tbody>
</table>

**Total:** 36.2 11.4 22.3 26.7

- Overall trends very similar to those for Pentium III.
- Even though very different architecture and compiler.

Results for Pentium 4

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Abstract -g</td>
<td>35.25</td>
<td>35.34</td>
</tr>
<tr>
<td>Abstract -Q2</td>
<td>26.52</td>
<td>30.26</td>
</tr>
<tr>
<td>Move vec_length</td>
<td>18.00</td>
<td>25.71</td>
</tr>
<tr>
<td>data access</td>
<td>3.39</td>
<td>31.56</td>
</tr>
<tr>
<td>Accum. in temp</td>
<td>2.00</td>
<td>14.00</td>
</tr>
<tr>
<td>Unroll 4</td>
<td>1.01</td>
<td>14.00</td>
</tr>
<tr>
<td>Unroll 16</td>
<td>1.00</td>
<td>14.00</td>
</tr>
<tr>
<td>X 2</td>
<td>1.02</td>
<td>7.00</td>
</tr>
<tr>
<td>X 4</td>
<td>1.01</td>
<td>3.99</td>
</tr>
<tr>
<td>X 8</td>
<td>1.62</td>
<td>4.50</td>
</tr>
</tbody>
</table>

**Total:** 35.2 8.9 18.7 18.0

- Higher latencies (int * x 14, fp * x 5.0, fp * x 7.0)
- Clock runs at 2.0 GHz
- Not an improvement over 1.0 GHz P3 for integer
- Avoids FP multiplication anomaly

What About Branches?

**Challenge**
- Instruction Control Unit must work well ahead of Exec. Unit.
- To generate enough operations to keep EU busy.

```
00489f8: movl $0x1, %ecx
00489f8: xorl %edx, %edx
00489f8: cmpl %esi, %edi
00489f8: jnl 0048e26
0048e00: imull (%esi, %edi, 4), %edi
```

- Executing
- Fetching & Decoding

**Branch Outcomes**
- When encountering 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.

```
0048a25: cmpl %edi, %edi
0048a27: jle 0048a20
0048a29: movl $0x0, %ebp
0048a2c: leal $0xffffffff (%ebp), %esi
0048a2f: movl %ecx, (%%esi)
```

Branch Not-Taken

```
Branch 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

Branch Prediction Through Loop

Assume vector length = 100
Predict Taken (OK)

Branch Misprediction Invalidation

Assume vector length = 100
Predict Taken (OK)

Branch Misprediction Recovery

Assume vector length = 100
Predict Taken (OK)

Performance Cost
- Misprediction on Pentium III wastes ~14 clock cycles
- That’s a lot of time on a high performance processor
Avoiding Branches

Avoiding Branches with Bit Tricks

On Modern Processor, Branches Very Expensive

- Unless prediction can be reliable
- When possible, use to avoid altogether

Example

- Compute maximum of two values
  - 14 cycles when prediction correct
  - 29 cycles when incorrect

```c
int max(int x, int y)
{
    return (x < y) ? y : x;
}
```

Avoiding Branches with Bit Tricks

- Force compiler to generate desired code
- Volatile declaration forces value to be written to memory
- Volatile declaration forces value to be written to memory
- Not very elegant!
  - Hack to get control over compiler
  - 22 clock cycles on all data
  - Better than misprediction

Conditional Move

- Added with P6 microarchitecture (PentiumPro onward)
- cmovXXI tdx, tdx
- If condition xx holds, copy tdx to tdx
- Doesn’t involve any branching
- Handed as operation within Execution Unit

```c
movl $(',tdx),%edx
movl $(',tdx),%edx
```

Current version of GCC won’t use this instruction
- Thinks it’s compiling for a 386

Performance

- 14 cycles on all data
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 IA-32/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