Notes from Previous Lectures

- **Induction variables:**
  Pegasus already has full analysis, so that if a node in a loop produces a value that changes by a loop-invariant amount each iteration, it is tagged with that information (starting value, stepping value). Besides IVSR, this is useful for optimizing memory accesses (recognizing streams), etc.

**CCP Example**

- We have seen how knowledge that a path could never execute could be exploited for more optimization:

```plaintext
if (s<10)
    x = 5;
    x = w-z;
y = x*x; y = 25;
s = {1}
```

**CCP Example**

- We saw how knowledge that a path could never execute could be exploited for more optimization:

```plaintext
if (s<10)
    x = 5;
    x = w-z;
y = x*x;
y = 25;
s = {1}
```
CCP Example

- What about this example? The path through [*] rarely executes...can we just ignore it? It's tempting...

```plaintext
if (today not leap day)
    x = 5;
    x = w-z;
    y = x*x;
```

CCP Example

- Result: code that is always faster and usually correct
- NOT ACCEPTABLE

CCP Example

- So, how can we exploit "probably" information in const prop? The classic version exploits only "definitely" information...

```plaintext
if (today not leap day)
    x = 5;
    x = w-z;
    y = x*x;
```

cmon, it's obvious once you see it....

CCP Example

- Answer – duplicate a basic block.

```plaintext
if (today not leap day)
    x = 5;
    x = w-z;
    y = x*x;
    y = x*x;
```
### CCP Example
- **Cost:** code expansion (probably)

```
if (today not leap day)

x = 5;

x = w-z;

y = x*x;
y = 25;
```

### General Rule
- "Do the common case fast, the other cases correctly"
- In other words, it makes sense to optimize the common paths even if it makes the uncommon paths slower. The weighted (dynamic) performance will improve.
- How does the compiler know what the "common case" is?

### PROFILING
- Ok, a "leap day" special case doesn't come up often;
- but skewed branches are common
- ...how can this be communicated to the compiler?
- Profiling information: A summary of program execution - run it with a sample input data set and record 'stuff'

### Profiling: How
- **Instrument** the executable, then run with sample data

```
count[1]++;

count[2]++;

count[3]++;

count[1]++;

count[2]++;

count[3]++;
```
Profiling

• Another reason: test coverage
  - Are there latent bugs in your code that have never been discovered because they've never been executed? How would you know? Make sure every block is executed...or possibly every path
  - But, this is a course in optimizing compilers, so back to optimization...

Other classic optimizations, w/ profiling

• We've seen CCP...what's this?

Other classic optimizations, w/ profiling

• We have seen Const Prop...what's this?

Global CSE!

using available expressions
deuja vu all over again...
Other classic optimizations, w/ profiling

- A Trend! It’s not that each optimization is modified to take profile information into account...
- Instead, the CFG is restructured so that infrequent paths don’t interfere with optimizations on the frequent path. Then just apply existing optimizations!

Generalizing...
- Since the common path is now a linear chain, it can be combined into one large block...with tail duplication.

Superblock
- Since the common path is now a linear chain, it can be combined into one large block...with tail duplication.
- This is a superblock - single entry at top, multiple exits.

Superblock Optimizations

super-block:
- constant propagation
- copy propagation
- constant combining
- dead code removal
- common subexpression elimination
- redundant store elimination
- redundant load elimination

super-block loop:
- loop invariant code removal
- loop induction variable elimination
- global variable migration

Chang, Mahlke, and Hwu
Superblock Loop Invariant Hoisting

if (i & 255)

step = i;

x = (step-1)<<2;

...<use x>

...

i = i+1;
if (i<N)

Superblock Loop Invariant Hoisting

if (i & 255)

step = i;

x = (step-1)<<2;

...<use x>

...

i = i+1;
if (i<N)

Superblock Loop Invariant Hoisting

x = (step-1)<<2;

if (i & 255)

x = (step-1)<<2;

...<use x>

...

i = i+1;
if (i<N)

Changing Gears

Other uses for profiling
Instruction Level Parallelism

- When the processor can perform multiple operations in parallel, **AND** the application has operations that can be executed in parallel, you can get a big performance boost:
  \[ \text{exec time} = \text{instructions} \times \text{cyclesPerInstruction} \times \text{cycle_period} \]
- But ILP within basic blocks is limited (2-3) - Foster, Tjaden, Wall
- To achieve more ILP, need to schedule across branches:
  - **superscalar**: branch prediction and dynamic scheduling
  - **VLIW**: compiler-controlled approaches
    - VLIW = Very Long Instruction Word
    - "Expose, Enhance, and Exploit ILP" - Bob Rau

- We will **not** address the details of scheduling here...
- Instead, discuss how to group computation **intelligently** to hand off to the scheduler

Early VLIW

- ELI project, Bulldog compiler (Josh Fisher, John Ellis)
- 1985 ACM dissertation award!
- Clustered VLIW, heterogenous function units
  - Sound familiar?
  - But back then, this was big iron
  - Targeting scientific code, not control-intensive code
  - And more clusters...8ish

- But...no predication! - just branches
- Core idea: **trace scheduling**

Trace Scheduling

- Example: growing backwards
- **Trace**: common sequence of basic blocks
  - to get a bigger group of operations to schedule
- Nice simple algorithm - assumes **profiling information**:
  - **start with "seed"** - most commonly executed bblock
  - **grow the trace** backwards and forwards while the most commonly executed impinging edge is also the most commonly executed impinging edge for the connecting bblock (though never cross a backedge)
  - **when this trace terminates on both ends**, repeat (start a new trace with a new seed), considering only unclaimed bblocks, until all bblocks are part of some trace.
Trace Scheduling

- Example: growing backwards

```
1000 10 110000
```

no further growth going backwards

• The pain: "bookkeeping": when the scheduler moves code in a trace above or below a branch exit/entry.

```
i := n+1
IF e
k := i+4
k := i+5
```

original scheduled trace

```
i := n+1
IF e
k := i+4
k := i+5
```

original scheduled trace, fixed

```
i := n+1
IF e
k := i+5
```

join

```
i := n+1
IF e
k := i+4
```

split
Trace Scheduling

• The pain: "bookkeeping": when the scheduler moves code in a trace above or below a branch exit/entry.

```
i := n+1
IF e
   k := i+4
k := i+4
```

• and that’s the easiest case; joins cause more problems
• get lots of code expansion

Superblocks

• Simplifying assumption: the only entry is at the top (no side joins allowed)

Superblocks

• Formation: same as traces: start with a seed, then grow forward and backwards
  - but cut off growth when count drops below a threshold (e.g. 100) to control code expansion
Superblock, Hyperblock

• Superblock summary: a single common path
  – More ILP – and also optimization benefits

• What if no single dominant path – but you have predication support?
• In that case, combine the common paths into a hyperblock

• Still get the optimization benefits of path exclusion, +more ILP,
  +great benefits from branch elimination

Hyperblock

• If-conversion converts control flow to predication
• Predicate expressions for each basic block in the hyperblock are calculated, then used to guard execution.

Hyperblock

• If-conversion converts control flow to predication
• Predicate expressions for each basic block in the hyperblock are calculated, then used to guard execution.
Hyperblock

- If-conversion converts control flow to predication
- Predicate expressions for each basic block in the hyperblock are calculated, then used to guard execution.

```
cmp r1=p1
add r2

cmpeq 4,r2=p2
(p1)add r4
(!p1)sub r4

and p1,p2=p3
and p1,!p2=p4
(p1)ld r5
(p3)ld r6
(p4)ld r6
(p1)add r4
(!p1)add r4
```

notes: reg; pred calculation; promotion

Predicate Promotion

- Predicate promotion means giving an operation a weaker predicate, so that it might execute when not needed.
- It breaks a dependence from the predicate calculation to the operation
- This is "speculation" - corresponds to moving an operation above a branch in control flow.
- Examples:
  
  (p1 & p2)  ->  p1
  (p1 & p2)  ->  T

- Useful when the predicate calculation is on the critical path

```
\text{Predicate Promotion}
\begin{align*}
\text{cmplt } B6, B7 &\to B0 \\
[ B0 ] &\text{ld } A3 \to A5 \\
[!B0 ] &\text{ld } A4 \to A5 \\
&\text{nop } 4 \\
&\text{addi } A5, 8 \to A8
\end{align*}
\begin{align*}
\text{ld } A3 \to A5 \\
\text{ld } A4 \to A6 \\
\text{cmplt } B6, B7 \to B0 \\
\text{nop } 3 \\
\text{mv } A6 \to A5 \\
\text{addi } A5, 8 \to A8
\end{align*}
```

Some architectures have only partial predicate support; they only have conditional moves, but no other predicated instructions. These require full predicate promotion...

```
Pegasus IR assumes full predicate promotion (multiplexors / muxes are conditional moves). Since we’re targeting an architecture with full predicate support, we might want to go in the other direction – predicate demotion? Benefits?
```
Hyperblock Formation

- Mahlke et al., IMPACT, Illinois: assuming VLIW with predication support

- General approach: start with the common trace, then add on parallel paths until it seems detrimental - don’t slow down the common path

- Heuristics, not a detailed analysis

- Cost factors when considering a path to be included:
  - execution frequency
  - number of instructions
  - dependence height
  - hazard conditions (call or unresolvable memory store)

Hyperblock Formation

- Hwu, Mahlke, et al., IMPACT, Illinois

- Conflicting considerations for including another path:
  - Good: reduce branches, increase parallelism
  - Bad: could saturate resources or increase dep height

- Remember the golden rule:
  Do the common case fast

Hyperblock Formation

- Examples: 4-wide VLIW

Hyperblock Formation

- Examples: 4-wide VLIW

Existing path(s)  Add this path?

Combined  Sure!
Hyperblock Formation

- Examples: 4-wide VLIW

Combined

NO – saturated the resources, and extends execution time of the common path.

Hyperblock Formation

- Examples: 4-wide VLIW

Combined

NO – dependence height of the infrequent path has extended execution time of the frequent path.
Hyperblock Formation

- Examples: 4-wide VLIW

Existing path(s)  Add this path?

Combined

Sure – you can always spread a “short fat” schedule out more vertically – if there’s room

Hyperblock Formation

- Heuristic for path i in region:

\[
\text{dep\_ratio}_i = 1.0 - \left(\frac{\text{dep\_height}_i}{\text{max}(\text{dep\_height}_j)}\right)
\]

\[
\text{op\_ratio}_i = 1.0 - \left(\frac{\text{num\_ops}_i}{\text{max}(\text{num\_ops}_j)}\right)
\]

hazard$_i$ = (call or store on i) ? 0.25 : 1.00

priority$_i$ = (prob$_i$ * hazard$_i$) * (dep$_\text{ratio}_i + \text{op\_ratio}_i + K$)

add paths in order of priority until resources saturated or priority drops too quickly