A Global Predication Compilation Framework

David I. August

Wen-mei W. Hwu
IMPACT Compiler Group
University of Illinois - Urbana/Champaign
Outline

- Predication Background
- Predication Frameworks
- Predicate Optimization
  - Fully Resolved Predicates
  - Code Specialization
  - Control Logic Optimization
- Ultrablock Predication Framework
- Predicate Analysis
- Predicate Dataflow
Predication Overview

- Conditional execution of an instruction based on a Boolean source operand

- Execution model
  - \( r1 = r1 + 1 \langle p1 \rangle \)
  - If \( p1 \) is TRUE, \( r1 \) is incremented.
  - If \( p1 \) is FALSE, \( r1 \) is unchanged.

- Provides the compiler with an alternative to guarding instructions with conditional branches.

- Levels of predication support
  - Full Predication Support
    * Predicate defining instructions
    * Full set of predicated instructions
    * Separate register file
  - Partial Predication Support - Existing ISA is enhanced with instructions such as CMOV or SELECT.
  - Dynamic Predication Support - ISA is unchanged.
Predication

- Architectures supporting predication:
  - Illiac IV - vector masks
  - Cydrome’s Cydra 5 - full predication
  - HPL’s PlayDoh - generalized Cydra 5
  - Intel and HP’s IA-64 - full predication
- *If-Conversion* is the process by which control flow is removed through the use of predication.
- *Reverse If-Conversion* is the process by which predication is removed through the introduction of control flow.
Uses of Predication

- *Predicated Representation* - A program representation in which instructions can be guarded by a Boolean source operand
  - Efficient model for compiler optimization and scheduling
  - Control transformations can be performed as simple optimizations.
  - Removal of control dependences affords optimization and scheduling freedom.

- *Predicated Execution* - An architectural model which supports direct execution of the predicated representation
  - Allows removal of branch mispredictions through elimination of branches
  - Increases ILP by allowing concurrent execution of multiple program paths
  - Enables predicate-specific optimizations such as height reduction
Predicate Defining Instructions

\[ P_{d0_{type0}}, P_{d1_{type1}} = ( src_0 \text{ cond } src_1 ) \langle P_g \rangle \]

- cond comparison: =, <, ≤, etc.
- type\textsubscript{i} assignment type:
  - UT/UF - Unconditional
  - OT/OF - Wired-or
  - AT/AF - Wired-and
  - CT/CF - Conditional
  - \lor T/\lor F - Disjunctive
  - \land T/\land F - Conjunctive
Unconditional Predicate Define

Generate a predicate for a block which executes on a single condition.

```plaintext
if (a < 10) 
c = c + 1;
else
  if (b < 20)
    d = d + 1;
  else
    e = e + 1;
```

\[ p_{1UT}, p_{2UF} = (a < 10) \]
\[ c = c + 1 \quad \langle p_1 \rangle \]

\[ p_{3UT}, p_{4UF} = (b < 20) \quad \langle p_2 \rangle \]
\[ d = d + 1 \quad \langle p_3 \rangle \]
\[ e = e + 1 \quad \langle p_4 \rangle \]
**Wired-OR Predicate Define**

Generate a predicate for a block which executes on multiple conditions.

\[
\text{if ( a \&\& b )} \\
\quad c = c + 1; \\
\text{else} \\
\quad d = d + 1; \\
\]

\[
p1 = 0 \\
p1_{OT}, p2_{UF} = (a == 0) \\
p1_{OT}, p3_{UF} = (b == 0) \langle p2 \rangle \\
c = c + 1 \langle p3 \rangle \\
d = d + 1 \langle p1 \rangle
\]
Wired-AND Predicate Define

Generate a predicate for a block which executes on multiple conditions.

\[
\begin{align*}
\text{if ( } a \& \& b \text{ ) } & \quad p1 = 0 \\
c = c + 1; & \\
\text{else } & \\
d = d + 1; & \quad p2 = 1 \\
p1_{OT}, p2_{AF} = (a == 0) & \\
p1_{OT}, p2_{AF} = (b == 0) & \\
c = c + 1 & \quad \langle p2 \rangle \\
d = d + 1 & \quad \langle p1 \rangle
\end{align*}
\]

\[
\begin{array}{c|c|c}
\text{Pg} & \text{Comparison} & \text{Pd} \\
\hline
0 & 0 & - & - \\
0 & 1 & - & - \\
1 & 0 & 0 & - \\
1 & 1 & - & 0 \\
\end{array}
\]
The If-Conversion During Scheduling Framework

- Best time to balance control flow and predication
- Minimizes effect on existing compiler
- Naive - doesn’t use predicated representation
The Hyperblock Compilation Framework

- Current state-of-the-art in the IMPACT compiler.
- Framework is designed to generate efficient code for predicated execution.
- Early heuristic hyperblock formation estimates final code characteristics:
Problems with Hyperblock Compilation Framework

- **Phase Ordering**
  - Strict phase-ordered creation of hyperblocks—early heuristic hyperblock formation, optimizations, then scheduling.
  - Interaction between resources and dependences is unpredictable.
  - Subsequent optimizations invalidate decisions made.
  - Estimates used in early heuristic hyperblock formation are not sufficiently fine-grained to include partial paths.

- **Compilation Block Scope**
  - Basic unit of compilation cannot contain loops.
  - Conservative hyperblock formation limits scheduling and optimization potential.
  - Conservative scope limits the types of transformations which can be applied.
Phase Ordering - The Optimization Problem

- Optimization changes a good hyperblock decision into a poor one:

Original Code → Hyperblock Formation (heuristic) → Optimizations (Predicate, ILP, and Traditional) → Optimized Hyperblock → Scheduled Code
Partial Reverse If-Conversion

- Overcomes the phase ordering problem
- Balances control flow and predication at schedule time
- Creates control flow after optimizations in the predicated representation
Partial Reverse If-Conversion

- Partial Reverse If-Conversion Decision:
  - Two Part Decision: Which Predicate, Where In Schedule
  - Consider: Resources, Dependence height, Hazards, Execution frequency

- Partial Reverse If-Conversion Mechanics:

Assume: T1, T2, T3 are not live out
r1, r2, r3 are live out

BEFORE

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T2 =</td>
<td>&lt;TRUE&gt;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T1 = T2</td>
<td>&lt;p1&gt;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>&lt;p2&gt;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r1 =</td>
<td>&lt;p2&gt;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T3 = T1</td>
<td>&lt;TRUE&gt;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r2 =</td>
<td>&lt;TRUE&gt;</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3 = T3</td>
<td>&lt;TRUE&gt;</td>
<td></td>
</tr>
</tbody>
</table>

AFTER

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T2 =</td>
<td>&lt;TRUE&gt;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T1 = T2</td>
<td>&lt;p1&gt;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>&lt;p2&gt;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r1 =</td>
<td>&lt;p2&gt;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T3 = T1</td>
<td>&lt;TRUE&gt;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r2 =</td>
<td>&lt;TRUE&gt;</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3 = T3</td>
<td>&lt;p1&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Assume: r1, r2, r3 are live out
T1, T2, T3 are not live out
Partial Reverse If-Conversion Algorithm

Without Partial Reverse If-Conversion

<table>
<thead>
<tr>
<th>Oper 1</th>
<th>Oper 2</th>
<th>Oper 3</th>
<th>Oper 4</th>
<th>Oper 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 = Cond</td>
<td>&lt;p1&gt;</td>
<td>&lt;p1&gt;</td>
<td>&lt;p1&gt;</td>
<td>&lt;p1&gt;</td>
</tr>
</tbody>
</table>

With Partial Reverse If-Conversion

<table>
<thead>
<tr>
<th>Oper 1</th>
<th>Oper 2</th>
<th>Oper 3</th>
<th>Oper 4</th>
<th>Oper 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 = Cond</td>
<td>Jump p1</td>
<td>&lt;p1&gt;</td>
<td>&lt;p1&gt;</td>
<td>&lt;p1&gt;</td>
</tr>
</tbody>
</table>

Ready:
- p1 = Cond
- Oper 2 <p1>
- Oper 3 <p1>
- Jump p1
Code Example

- In the function \_mark in the benchmark 022.li:
  - 2 of 20 possible reverse if-conversions performed.
  - 58764 cycles → 38942 cycles → 34827 cycles
Performance Improvement

- No branch prediction penalty
- 4-issue: 1 branch, 2 integer, 2 memory, and 1 float
## Application Statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reverse If-Conversions</th>
<th>Opportunities</th>
</tr>
</thead>
<tbody>
<tr>
<td>008.espresso</td>
<td>204</td>
<td>1552</td>
</tr>
<tr>
<td>022.li</td>
<td>50</td>
<td>393</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>43</td>
<td>443</td>
</tr>
<tr>
<td>026.compress</td>
<td>11</td>
<td>56</td>
</tr>
<tr>
<td>072.sc</td>
<td>33</td>
<td>724</td>
</tr>
<tr>
<td>085.cc1</td>
<td>479</td>
<td>3827</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>134</td>
<td>1021</td>
</tr>
<tr>
<td>134.perl</td>
<td>42</td>
<td>401</td>
</tr>
<tr>
<td>cccp</td>
<td>77</td>
<td>1046</td>
</tr>
<tr>
<td>cmp</td>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>eqn</td>
<td>33</td>
<td>326</td>
</tr>
<tr>
<td>grep</td>
<td>3</td>
<td>103</td>
</tr>
<tr>
<td>wc</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>yacc</td>
<td>247</td>
<td>1976</td>
</tr>
</tbody>
</table>
Fully Resolved Predicates: Motivation

- Typical Hyperblocks and Superblocks have many infrequently taken exit branches.
- Infrequent exit branches
  - impede code motion
  - increase length of path to frequently taken branches
  - consume valuable branch resources
- Goal: Use predication to enhance performance in the presence of easily predicted branches.
Fully Resolved Predicates: Concept

- Partially Resolved Predicates (PRP)
  - Instruction execution is guarded by predicates or branches.
  - Some control dependences remain in predicated code.

- Fully Resolved Predicates (FRP)
  - Instructions are guarded by predicates even if guarded by branches.
  - All control dependences within a region are eliminated.
  - Any instruction can be hoisted above a branch without speculation.
Fully Resolved Predicates: Computation

Partially Resolved Predicates

Fully Resolved Predicates

David I. August
IMPACT Research Group
Fully Resolved Predicates: Optimization Opportunities

- Branch reordering
  - Branches can be placed in any order.
  - Move more frequently taken branches above less frequently taken branches.

- Instruction percolation without speculation
  - Percolated instructions can never have side effects because they are guarded by predicates.
  - Store instructions
    * Speculating stores has traditionally been problematic for most speculation schemes.
    * Inability to speculate stores limits available ILP.
Fully Resolved Predicates: Case Study

- `grep` function “execute” inner loop
- Segment accounts for about 40% of total execution time.
- Source:

```c
for ( ; ; )
{
    if (p2 >= ebp)
        /* Excluded from Hyperblock */
        if ((c = *p2++) == 'n')
            break;
    if (c)
        if (p1 < &linebuf[1024-1])
            *p1++ = c;
}```
## Fully Resolved Predicates: Code Example

### Original Code Segment:

<table>
<thead>
<tr>
<th>CB 6:</th>
<th>Taken Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r35 = MEM[r34]</td>
</tr>
<tr>
<td>2</td>
<td>r34 = r34 + 1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MEM[r33] = r35</td>
</tr>
</tbody>
</table>

### FRP Predicated Code Segment:

<table>
<thead>
<tr>
<th>CB 6:</th>
<th>Taken Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r35 = MEM[r34]</td>
</tr>
<tr>
<td>2</td>
<td>r34 = r34 + 1</td>
</tr>
<tr>
<td>3</td>
<td>p2_{uf}, p3_{uf} = (r35 == 10)</td>
</tr>
<tr>
<td>4</td>
<td>p4_{uf}, p5_{uf} = (r35 == 0)</td>
</tr>
<tr>
<td>5</td>
<td>p6_{uf}, p7_{uf} = (r33 &gt;= r57)</td>
</tr>
<tr>
<td>6</td>
<td>MEM[r33] = r35</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Path Height Reduction: Concept

- Path Classes
  - dependence limited
  - resource limited
- Optimizations can be performed to exchange dependence height for resource usage
- Goal: balance resource height and dependence height to **reduce effective height of path**

Sequential code:
```
+-----+-
|     |   |
+-----+-
|     |   |
+-----+-
|     |   |
```

Saturated code:
```
+-----+-
|     |   |
+-----+-
|     |   |
```

- Height goes from 6 to 2
- Operation count went from 10 to 14
- Extra operations absorbed by processor width
Path Height Reduction: Concept

Original:
\[ T1 = A \circ B \]
\[ T2 = T1 \circ C \]
\[ E = T2 \circ D \]

Single back substitution:
\[ T1 = A \circ B \]
\[ E = T1 \circ C \circ D \]

Final:
\[ E = A \circ B \circ C \circ D \]

Arithmetic Semantics—Tree of Computation:
\[ T1 = A \circ B \]
\[ T2 = C \circ D \]
\[ E = T1 \circ T2 \]

Parallel Semantics:
\[ E\circ= A \circ B \]
\[ E\circ= C \circ D \]

“\(\circ\)” represents the universal associative operator.
## FRP/PHR: Code Example

### FRP Predicated Code Segment:

<table>
<thead>
<tr>
<th>CB 6:</th>
<th>Taken Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r35 = MEM[r34]</td>
</tr>
<tr>
<td>2</td>
<td>r34 = r34 + 1</td>
</tr>
<tr>
<td>3</td>
<td>p2_{uf}, p3_{uf} = (r35 == 10)</td>
</tr>
<tr>
<td>4</td>
<td>p4_{uf}, p5_{uf} = (r35 == 0)</td>
</tr>
<tr>
<td>5</td>
<td>p6_{uf}, p7_{uf} = (r33 &gt;= r57)</td>
</tr>
<tr>
<td>6</td>
<td>MEM[r33] = r35</td>
</tr>
<tr>
<td>7</td>
<td>jump CB 11</td>
</tr>
</tbody>
</table>

### FRP Predicated Code Segment with Height Reduction:

<table>
<thead>
<tr>
<th>CB 6:</th>
<th>Taken Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r35 = MEM[r34]</td>
</tr>
<tr>
<td>2</td>
<td>r34 = r34 + 1</td>
</tr>
<tr>
<td>3</td>
<td>p2_{af}, p3_{af} = (r35 == 10)</td>
</tr>
<tr>
<td>4</td>
<td>p4_{af}, p5_{af} = (r35 == 0)</td>
</tr>
<tr>
<td>5</td>
<td>MEM[r33] = r35</td>
</tr>
<tr>
<td>6</td>
<td>jump CB 11</td>
</tr>
</tbody>
</table>

---

David I. August  
IMPACT Research Group
FRP/PHR: *grep* Code Example Performance

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Original HB</th>
<th>FRP Only</th>
<th>FRP w/ Height Red.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>4035</td>
<td>4035</td>
<td>101148</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4035</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>101148</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>101148</td>
<td>101148</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>101148</td>
<td>101148</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Cycles**
- Original HB: 619007
- FRP Only: 623056
- FRP w/ Height Red.: 424795

**Speedup**
- Original HB: 1.00
- FRP Only: 0.99
- FRP w/ Height Red.: 1.46

- FRP enabled a 46% speedup for a single iteration.
- Performance of this optimization is magnified by unrolling.
Code Specialization: Case Study

- compress function "compress" inner loop

- Source:

  probe:
  
  ```
  if ((i -= disp) < 0)
      i += hsize_reg;
  if (htabof(i) == fcode)
      /* Excluded from Hyperblock */
  if (htabof(i) > 0)
      goto probe;
  ```
# Code Specialization: Code Example

## Original Code Segment

<table>
<thead>
<tr>
<th>Original Code Segment</th>
<th>CB 38:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( r_9 = r_9 - r_{12} )</td>
<td>( r_9 = r_9 - r_{12} )</td>
</tr>
<tr>
<td>2. ((p1_{uf}) = (r_9 &lt; 0))</td>
<td>((p1_{uf}, p2_{ut}) = (r_9 &lt; 0))</td>
</tr>
<tr>
<td>3. ( r_9 = r_9 + r_{13} )</td>
<td>( r_9 = r_9 + r_{13} )</td>
</tr>
<tr>
<td>4. ( r_{10} = r_9 &lt;&lt; 2 )</td>
<td>( r_{110} = r_9 &lt;&lt; 2 )</td>
</tr>
<tr>
<td>5. ( r_{114} = \text{MEM}[r_{10}] )</td>
<td>( r_{114} = \text{MEM}[r_{110}] )</td>
</tr>
<tr>
<td>6. ( \text{branch (r}<em>{14} &lt;&gt; r</em>{8}) ) CB 38</td>
<td>( \text{branch (r}<em>{14} &lt;&gt; r</em>{8}) ) CB 38</td>
</tr>
</tbody>
</table>

## Specialized Code Segment

<table>
<thead>
<tr>
<th>Specialized Code Segment</th>
<th>CB 38:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( r_9 = r_9 - r_{12} )</td>
<td>( r_9 = r_9 - r_{12} )</td>
</tr>
<tr>
<td>2. ((p1_{uf}, p2_{ut}) = (r_9 &lt; 0))</td>
<td>((p1_{uf}, p2_{ut}) = (r_9 &lt; 0))</td>
</tr>
<tr>
<td>3. ( r_{110} = r_9 &lt;&lt; 2 )</td>
<td>( r_9 = r_9 + r_{13} )</td>
</tr>
<tr>
<td>4. ( r_{114} = \text{MEM}[r_{110}] )</td>
<td>( r_{10} = r_9 &lt;&lt; 2 )</td>
</tr>
<tr>
<td>5.</td>
<td>( r_{14} = \text{MEM}[r_{10}] )</td>
</tr>
<tr>
<td>6. ( \text{branch (r}<em>{14} &lt;&gt; r</em>{8}) ) CB 38</td>
<td>( \text{branch (r}<em>{14} &lt;&gt; r</em>{8}) ) CB 38</td>
</tr>
</tbody>
</table>

## Specialized Code Segment After Optimization

<table>
<thead>
<tr>
<th>Specialized Code Segment After Optimization</th>
<th>CB 38:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_{1312} = r_{13} - r_{12} )</td>
<td>( r_9 = r_9 - r_{12} )</td>
</tr>
<tr>
<td></td>
<td>( r_{1009} = r_{10} + r_{1312} )</td>
</tr>
<tr>
<td>2. ( r_{110} = r_9 &lt;&lt; 2 )</td>
<td>((p1_{uf}, p2_{ut}) = (r_9 &lt; 0))</td>
</tr>
<tr>
<td>3. ( r_{14} = \text{MEM}[r_{110}] )</td>
<td>( r_{110} = r_9 &lt;&lt; 2 )</td>
</tr>
<tr>
<td></td>
<td>( r_{14} = \text{MEM}[r_{10}] )</td>
</tr>
<tr>
<td>4.</td>
<td>( r_{9} = r_{1009} )</td>
</tr>
<tr>
<td>5. ( \text{branch (r}<em>{14} &lt;&gt; r</em>{8}) ) CB 38</td>
<td>( \text{branch (r}<em>{14} &lt;&gt; r</em>{8}) ) CB 38</td>
</tr>
</tbody>
</table>

---

David I. August  
IMPACT Research Group
Advanced Control Flow Transformation

Original predicate definition schedule

<table>
<thead>
<tr>
<th>Original predicate expressions</th>
<th>Expressed in terms of conditions</th>
<th>Minimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>p12</td>
<td>c1</td>
<td>c1</td>
</tr>
<tr>
<td>p13</td>
<td>p12 &amp; c2</td>
<td>c1 &amp; c2</td>
</tr>
<tr>
<td>p14</td>
<td>p13 &amp; c3</td>
<td>c1 &amp; c2 &amp; c3</td>
</tr>
<tr>
<td>p15</td>
<td>!c1</td>
<td>p12 &amp; !c2</td>
</tr>
<tr>
<td>p16</td>
<td>p15 &amp; c4</td>
<td>!c1</td>
</tr>
<tr>
<td>p17</td>
<td>p15 &amp; !c4</td>
<td>(!c1</td>
</tr>
<tr>
<td>p18</td>
<td>p17 &amp; !c5</td>
<td>(!c1</td>
</tr>
<tr>
<td>p19</td>
<td>p15 &amp; c4</td>
<td>p17 &amp; c5</td>
</tr>
</tbody>
</table>

Predicate definition schedule after range analysis and and-type parallelization

<table>
<thead>
<tr>
<th>p14_at = (r4 &gt; 32)</th>
<th>&lt;T&gt;</th>
<th>p14_at = (r4 &lt; 127)</th>
<th>&lt;T&gt;</th>
<th>p14_at = (r2 == 0)</th>
<th>&lt;T&gt;</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>p19_ot, p16_ut = (r4 == 10)</td>
<td>&lt;T&gt;</td>
<td>p19_ot = (r4 == 32)</td>
<td>&lt;T&gt;</td>
<td>p19_ot = (r4 == 9)</td>
<td>&lt;T&gt;</td>
<td>...</td>
</tr>
</tbody>
</table>
Advanced Control Flow Transformation

- The predicated representation enables extraction and manipulation of program control logic.

- Optimization of predicate defines can be formulated as a specialized logic synthesis problem.
  - Predicate definitions are analogous to gates. They consume resources.
  - Predicate computation height is analogous to total gate delay.
  - Inputs may be available at different times.
  - Resource availability changes with the schedule.

- Algorithm overview:
  - Analyze conditions for interrelation.
  - Extract program control logic from extant predicate defines.
  - Minimize logical expressions using Boolean optimization techniques.
  - Factor control expressions based on condition availability and schedule freedom.
  - Re-express control as a new, optimized predicate define network.
Compilation Block Scope - The Loop Boundary Problem

- Acyclic nature of hyperblocks precludes pre-loop and post-loop block subsumption.
The UltrablockCompilation Framework

- Best use of predicated representation: Early aggressive formation which can support generalized regions

- Best use of predicated execution: Partial Reverse If-Conversion for scheduler adjustment of predication and reinstatement of control flow
Intermediate Representation

- IR needs to be extended to represent ultrablocks which can represent internal cycles to support compilation of general regions.
- Special purpose control flow and loop transformations can be replaced by data flow optimizations.
- A few techniques possible with current data flow optimizations are: loop versioning, loop fusion, if-then-else fusion, if-then-else interchange.
Few compilers do loop versioning, probably because it is a complicated and/or expensive control flow transformation.

- 3,608,541 dynamic loop iterations in 085.cc1
- 1,309,548 (36%) of these iterations have loop invariant, program variant branches and predicates.
- 374,279 (10%) of these iterations have loop invariant, program variant predicates.
Predicate Analysis

- Predicate Analysis analyzes predicate definitions to understand how predicates relate to one another.
- This information is essential for the compilation process.
  - Optimization
  - Register Allocation
  - Scheduling
- Predicate analysis applied to optimization—constant propagation example:

<table>
<thead>
<tr>
<th>$p^{1_ut}$ = $cond_1$</th>
<th>$p^{2_ut}$, $p^{1_ot}$ = $cond_1$</th>
<th>$p^{1_ut}$, $p^{2_at}$ = $cond_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p^{2_ut}$ = $cond_2$ &lt; $p_1$ &gt;</td>
<td>$p^{1_ot}$ = $cond_2$</td>
<td>$p^{2_at}$ = $cond_2$</td>
</tr>
</tbody>
</table>

If $p_1$ is a superset of $p_2$:

- $r_1 = 10$ < $p_1$ >  
- $r_2 = r_1 + 2$ < $p_2$ >  $\Rightarrow$  $r_2 = 12$ < $p_2$ >
Predicate Analysis—Related Work

- Predicate Analysis has traditionally been done hierarchically.
- Predicate Hierarchy Graph (PHG), the original system in IMPACT, is purely hierarchical.
- Unfortunately, predicates are not always related in a hierarchical fashion and these systems cannot accurately represent all relationships.

\[
\begin{align*}
p^2_{ut}, p^1_{ot} &= cond_1 \\
p^1_{ot} &= cond_2 \\
p^3_{ut} &= cond_3 < p^2 >
\end{align*}
\]

\(p^1\) is not an ancestor of \(p^3\), but \(p^1\) is a superset of \(p^3\).

- Predicate Query System (PQS) - used in the Elcor compiler at HP Labs makes approximations in other ways.
The Predicate Analysis System (PAS)

- Predicate definitions are essentially Boolean expressions — leverage CAD work in Boolean representations to represent all predicate relations.
- The PAS is built upon Binary Decision Diagrams (BDDs) — specifically, PAS was built upon Cudd. [Somenzi]
- In addition to being unable to represent all relations, the PHG and PQS are:
  - limited locally to a single hyperblock.
  - not able to understand branch guards.

\[
p1 = \text{cond0 \& cond1} \mid \neg\text{cond0 \& cond2}
\]

- PAS can represent instruction guarding by branches and predicates.
- Each instruction in the program has a complete expression of its execution, with the exception of loops.
Dataflow Analysis

- Dataflow can be performed without regard to predicates; results are conservative.
- Conservative results make optimizations, scheduling, and register allocation less effective.
- Conservative dataflow
  - Only instructions on TRUE can KILL.
  - $r_3$ is not killed by instruction 3 because it is predicated.
  - The live range of $r_3 = \{1, 2, 3, 4\}$.
- Predicate-aware dataflow
  - Instructions on a predicate KILL on that predicate.
  - The live range of $r_3 = \{3, 4\}$.

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$p_{1un} = (r_1 &lt; 0)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$p_{2un} = (r_2 &lt; 0)$</td>
<td>$&lt;p_1&gt;$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$r_3 = r_4 + r_5$</td>
<td>$&lt;p_1&gt;$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$r_8 = r_3 + 1$</td>
<td>$&lt;p_2&gt;$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$r_7 = r_4 + r_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$r_4 = r_7 - 1$</td>
<td>$&lt;p_1&gt;$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$r_9 = r_9 / 2$</td>
<td>$&lt;p_2&gt;$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dataflow Analysis—Predicate Flow Graph

- Developed the Predicate Flow Graph (PFG) which can perform predicate-sensitive dataflow analysis.

- Idea was to change the underlying graph so that traditional dataflow analysis techniques would generate correct results.

- Results have shown that accurate dataflow analysis has been achieved.
Dataflow Analysis Path Explosion Problem

- Predication eliminates the need for many paths to exist in control flow.
- Using the PFG based approach these paths become materialized.
- As a general rule, the path width of the PFG is greater than $2^n$, where $n$ is the number of independent predicates with overlapping live ranges.
- Assuming $p_1$, $p_2$, and $p_3$ are independent we have $2^3 = 8$ paths.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$p_{1un} = (r1 &lt; 0)$</td>
</tr>
<tr>
<td>2</td>
<td>$p_{2un} = (r2 &lt; 0)$</td>
</tr>
<tr>
<td>3</td>
<td>$p_{3un} = (r3 &lt; 0)$</td>
</tr>
<tr>
<td>4</td>
<td>$r_5 = X$ &lt;p1&gt;</td>
</tr>
<tr>
<td>5</td>
<td>$r_6 = Y$ &lt;p2&gt;</td>
</tr>
<tr>
<td>6</td>
<td>$Z = r_5$ &lt;p3&gt;</td>
</tr>
</tbody>
</table>
Dataflow Analysis: Disjunctive Compositions

- The key to eliminating the exponential nature of dataflow analysis is a partition graph of disjunctive expressions.
- By operating on a partition graph, interactions between independent predicates can be expressed without enumerating all paths.
- Predicates are composed of nodes, any two of which exist in exactly one of three relationships: implication, independence, or exclusivity.
- Using only such nodes guarantees that complex relationships between predicates can be represented exactly, yielding accurate dataflow results.

<table>
<thead>
<tr>
<th>SSA pred. def.</th>
<th>Resulting disjunctive expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_{i,j} = \text{ut} C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g C )</td>
</tr>
<tr>
<td>( p_{i,j} = \text{uf} C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g C' )</td>
</tr>
<tr>
<td>( p_{i,j} = p_{i,j-1} \text{ ot } C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g p_{i,j-1} \lor p_g p_{i,j-1} C' \lor p_g C )</td>
</tr>
<tr>
<td>( p_{i,j} = p_{i,j-1} \text{ of } C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g p_{i,j-1} \lor p_g p_{i,j-1} C' \lor p_g C )</td>
</tr>
<tr>
<td>( p_{i,j} = p_{i,j-1} \text{ ut } C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g p_{i,j-1} \lor p_g C )</td>
</tr>
<tr>
<td>( p_{i,j} = p_{i,j-1} \text{ uf } C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g p_{i,j-1} \lor p_g C' )</td>
</tr>
<tr>
<td>( p_{i,j} = p_{i,j-1} \text{ at } C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g p_{i,j-1} \lor p_g p_{i,j-1} C )</td>
</tr>
<tr>
<td>( p_{i,j} = p_{i,j-1} \text{ af } C \langle p_g \rangle )</td>
<td>( p_{i,j} = p_g p_{i,j-1} \lor p_g p_{i,j-1} C' )</td>
</tr>
</tbody>
</table>
A Global Predication Compilation Framework

David I. August

Wen-mei W. Hwu
IMPACT Compiler Group
University of Illinois - Urbana/Champaign