Systematic Program Decision Logic Optimization Using Predication

David I. August

IMPACT Compiler Research Group

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Instruction-Level Parallelism Overview

- Instruction-Level Parallelism (ILP), the concurrent execution of independent assembly instructions, is a cost effective way to extract performance from programs.

- Exposing large amounts of ILP requires global optimization and code motion.

- Processors are becoming increasingly dependent on the ability of compilers to expose ILP.
  - Current state-of-the-art machines can execute 3 to 6 instructions per cycle. (i.e. Pentium III, DEC Alpha 21264)
  - Expect future processors to execute 6 to 12 instructions per cycle. (i.e. Merced, DEC Alpha 21364)

- Current state-of-the-art compilers cannot expose this level of ILP in integer programs.
Program Computation and Decision Components

- Programs can be viewed as being composed of two components: *Computation* and *Decision*.
- Decision controls which parts of computation are conducted.
- Results of computation comprise the inputs to the decision logic.
- Three classes of optimization:
  - Computation
  - Decision
  - Computation-Decision Interface
Program Optimization Techniques

- Optimization of computation
  - copy propagation, constant folding, expression reformulation

- Manipulation of the computation-decision interface
  - Working around interface as a barrier to compilation:
    \textit{Global scheduling} [Moon, Ebcioğlu]
    \textit{Trace compilation framework} [Fisher, Multiflow]
    \textit{Superblock compilation framework} [IMPACT]
    \textit{Treregion compilation framework} [Conte]
    \textit{PDG compilation framework} [Ferrante, Gupta and Soffa]
  - Breaking control dependences:
    \textit{Compiler-directed control speculation} [Multiflow, Mahlke, August]
  - Managing uncertainty of outcome:
    \textit{Branch prediction and out-of-order execution} [Yeh and Patt, Smith]
    \textit{Multiscalar architectures} [Sohi]
Program Optimization Techniques (cont.)

- Optimization of decision
  - *Control height reduction* [Schlansker]
  - *Branch Combining* [Mahlke]
  - *Tree-style branch VLIW architectures* [Ebcioğlu]

- Previous decision optimization techniques have taken a structural rather than a semantic approach.

- Much unexploited opportunity for optimization of decision:
  - Decision contains large amounts of redundancy.
  - Decision is typically structured by programmer for simplicity.
  - Compilers have not effectively restructured decision for target machine.
Goal of this Work

- **GOAL:** Perform in-depth semantic analysis to enable thorough optimization of the decision component.

- **Requirements:**
  - Be a single general technique. (not ad hoc, pattern match, or peephole)
  - Consider entire decision component at once to find redundancy.
  - Understand available resources of target machine.
  - Adapt to match computation component height and resources.

- *Predication* facilitates optimization of decision. [Pnevmatikatos, ISCA21] [Mahlke, ISCA22]
  - Decision logic is expressed in an easily analyzable form.
  - Interaction between decision and computation is only through flow dependences.
Predication

- Architectures supporting predication:
  - Cydrome Cydra 5 - full predication [Dehnert]
  - HPL PlayDoh - generalized Cydra 5 [Kathail]
  - Intel/HP IA-64 - full predication [Crawford]
- Enables conditional execution of individual instructions.
  - A set of Boolean values called predicates
  - Predicate definition instructions set the predicates
  - All instructions are guarded by a source predicate

- If-Conversion:
  Control Flow ⇒ Predication

- Reverse If-Conversion:
  Predication ⇒ Control Flow
Predication Uses

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

- **Predicated Execution** - An architectural model which supports direct execution of the predicated representation
  - Increases ILP by allowing simultaneous execution of multiple program paths.
  - Enables predicate-specific optimizations such as height reduction.
  - Allows removal of branch mispredictions through elimination of branches.
The IMPACT Predication Compilation Framework

- Modeled after The Partial Reverse If-Conversion Framework. [August, Micro30]

- Early phase predication enables full use of Predicated Representation by optimizer.

- Complete early phase predication translates the program decision component into a single easily manageable form.
Program Decision Logic Optimization

Steps in decision logic optimization:

1. **If-convert all decision logic to predication.**
2. **Analyze predicated code and its interface with computation.**
3. **Optimize Boolean logic representation using standard Boolean minimization techniques and condition information.**
4. **Factorize and Synthesize Boolean logic using schedule information.**
5. **Reverse if-convert predicated code as necessary for target architecture.**
Conversion of Control Flow to Predication

- Conversion likely to be over-aggressive for target architecture.
# Promotion and Essential Predicates

- **Promotion** is the replacement of a guard predicate by another which subsumes it; promotion is speculation in the predicate domain.

- **Essential Predicates** are those which guard *computation* after promotion.

<table>
<thead>
<tr>
<th>Before Promotion</th>
<th>After Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1_ut, p2.uf = (r1 &gt;= 0)</td>
<td>p1_ut, p2.uf = (r1 &gt;= 0)</td>
</tr>
<tr>
<td>p3.ot, p4.uf = (r1 == 1)</td>
<td>p3.ot, p4.uf = (r1 == 1)</td>
</tr>
<tr>
<td>p3.ot, p5.uf = (r1 == 2)</td>
<td>p3.ot, p5.uf = (r1 == 2)</td>
</tr>
<tr>
<td>p3.ot, p6.uf = (r1 == 3)</td>
<td>p3.ot, p6.uf = (r1 == 3)</td>
</tr>
<tr>
<td>COMP A</td>
<td>COMP A</td>
</tr>
<tr>
<td>COMP B</td>
<td>COMP B</td>
</tr>
<tr>
<td>br (r8 == 0)</td>
<td>br (r8 == 0)</td>
</tr>
<tr>
<td>COMP C, [r2 = ]</td>
<td>COMP C, [r2 = ]</td>
</tr>
<tr>
<td>p7_ut = (r2 &gt; r3)</td>
<td>p7_ut = (r2 &gt; r3)</td>
</tr>
<tr>
<td>p8.ot, p9.uf = (r4 &gt; r5)</td>
<td>p8.ot, p9.uf = (r4 &gt; r5)</td>
</tr>
<tr>
<td>p8.ot = (r6 &gt; r7)</td>
<td>p8.ot = (r6 &gt; r7)</td>
</tr>
<tr>
<td>COMP D0</td>
<td>COMP D0</td>
</tr>
<tr>
<td>COMP D1</td>
<td>COMP D1</td>
</tr>
<tr>
<td>COMP D2</td>
<td>COMP D2</td>
</tr>
<tr>
<td>COMP D3</td>
<td>COMP D3</td>
</tr>
<tr>
<td>COMP D4</td>
<td>COMP D4</td>
</tr>
<tr>
<td>jump</td>
<td>jump</td>
</tr>
</tbody>
</table>

After Promotion:

**Essential Predicates:** p3, p6, p8

**Non-Essential Predicates:** p1, p2, p4, p5, p7, p9
Accurate Analysis of Predicated Code

- Predicate Analysis determines how predicates relate to one another.
- This information is essential for most compiler functions including optimization, dataflow analysis, and scheduling.
- Constant propagation example:

  | If \( p_1 \) is a superset of \( p_2 \): |
  |---|---|
  | \( r_1 = 10 \) | \( \langle p_1 \rangle \) |
  | \( r_2 = r_1 + 2 \) | \( \langle p_2 \rangle \) |
  | \( \implies r_2 = 12 \langle p_2 \rangle \) |

- Analysis is complicated by limitless variations of predicate expressions:

  \[
  \begin{align*}
  p_{1\_ut} &= \text{cond1} \\
p_{2\_ut} &= \text{cond2} \langle p_1 \rangle \\
  \downarrow & \\
p_1 &= \text{cond1} \\
p_2 &= \text{cond1} \land \text{cond2}
  \end{align*}
  \hspace{1cm}
  \begin{align*}
  p_{1\_ut}, p_{2\_at} &= \text{cond1} \\
p_{2\_at} &= \text{cond2} \\
  \downarrow & \\
p_1 &= \text{cond1} \\
p_2 &= \text{cond1} \land \text{cond2}
  \end{align*}
  \hspace{1cm}
  \begin{align*}
  p_{2\_ut}, p_{1\_ot} &= \text{cond1} \\
p_{1\_ot} &= \text{cond2} \\
  \downarrow & \\
p_1 &= \text{cond1} \lor \text{cond2} \\
p_2 &= \text{cond2}
  \end{align*}
\]
Previous Attempts at Predicate Analysis

- Predicate Analysis has traditionally been done hierarchically.

- Predicate Hierarchy Graph (PHG), the original system in IMPACT, is purely hierarchical [Mahlke, Ph.D.].

\[
\begin{align*}
p1_{\text{ut}} &= \text{cond1} \\
p2_{\text{ut}} &= \text{cond2} \langle p1 \rangle
\end{align*}
\]

\[
\begin{align*}
p1 &= \text{cond1} \\
p2 &= \text{cond1} \land \text{cond2}
\end{align*}
\]

\(p2\) is a child of \(p1\), and \(p2\) is a subset of \(p1\).

- Unfortunately, predicates are not always related in a hierarchical fashion and these systems cannot accurately represent all relationships.

\[
\begin{align*}
p2_{\text{ut}}, p1_{\lor t} &= \text{cond1} \\
p1_{\lor t} &= \text{cond2} \\
p3_{\text{ut}} &= \text{cond3} \langle p2 \rangle
\end{align*}
\]

\[
\begin{align*}
p1 &= \text{cond1} \lor \text{cond2} \\
p3 &= \text{cond1} \land \text{cond3}
\end{align*}
\]

\(p1\) is not an ancestor of \(p3\), but \(p1\) is a superset of \(p3\).

- Predicate Query System (PQS), used in the Elcor compiler at HP Labs, makes approximations in other ways [Johnson, Micro29].
The Predicate Analysis System (PAS)

- Predicate definitions are essentially Boolean expressions—leverage CAD research in Boolean representations to represent all predicate relations.
- The PAS is built upon Cudd Binary Decision Diagrams (BDD’s). [Somenzi] Canonical form provides efficient response to queries.
- The predicate network is examined and the \textit{predicate BDD} is constructed.

\[
p1_{ut}, p2_{uf} = (r1 \geq 0) \\
p3_{ot}, p4_{uf} = (r1 == 1) \ <p1> \\
p3_{ot}, p5_{uf} = (r1 == 2) \ <p4> \\
p3_{ot}, p6_{uf} = (r1 == 3) \ <p5> \\
\text{COMP A} \ \ <p6> \\
\text{COMP B} \ \ <p3>
\]

\begin{align*}
p1 &= C0 \\
p4 &= \overline{C1} \land p1 = \overline{C1} \land C0
\end{align*}

\[
f = A \land B = B \ ? \ (A \ ? \ 1 : 0) : 0
\]
The Condition Analysis System

- Comparisons (\textit{i.e.} \( r1 < 10 \), \( r1 > r2 \))
  - Comprise the interface between computation and decision.
  - Are the input variables of the predicate BDD.
- Conditions may be inherently related: (\textit{i.e.} \( r1 < 5 \) implies \( r1 < 10 \))
- Condition relationships create additional predicate relationships.
- The Predicate BDD stores relations among predicates.
- The \textit{Condition BDD} stores relations among conditions.
- For compiler queries, the predicate and condition BDD’s are combined.
- For use in expression reformulation, the BDD’s must be separate.

\[ p1 = C1 \land C2 \Rightarrow (r1 > 10) \land (r1 < 20) \Rightarrow (10 < r1 < 20) \]
Range Determination
(Collaboration with John Sias)

- All register/constant comparisons based on the same register compose a family.
- Within each family, the logical interrelation of conditions can be expressed.
- For each family, a number line is divided into disjoint intervals.
- Conditions are composed by a union of intervals.

In our example:

\[ C'0 \quad (r1 \geq 0) \quad \bigcup_{j=1}^{5} I_j \]
\[ C'1 \quad (r1 = 1) \quad I_2 \]
\[ C'2 \quad (r1 = 2) \quad I_3 \]
\[ C'3 \quad (r1 = 3) \quad I_4 \]
Finite Domain Mapping

- Relationships between conditions are expressed in the condition BDD using a finite domain technique [Bryant 92].

- Each family’s set of \( n \) enumerated intervals is mapped exhaustively to an unique Boolean \( \lceil \log_2 n \rceil \)-space in the condition BDD.

- Each condition node is composed of a disjunction of interval nodes.

\[
\begin{array}{cccccc}
& -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
I_0 & I_1 & I_2 & I_3 & I_4 & I_5 \\
\end{array}
\]

\[\lceil \log_2 6 \rceil = 3\]

Mapping is to a Boolean 3-space.
(3 Independent Domain Variables)

<table>
<thead>
<tr>
<th>Cond</th>
<th>BDD Expression</th>
</tr>
</thead>
</table>
| \( C_0 \) | \( I_5? 1 : (I_4? 1 : \ldots \)  
\( \ldots (I_3? 1 : (I_2? 1 : I_1)) \) |
| \( C_1 \) | \( I_2 \) |
| \( C_2 \) | \( I_3 \) |
| \( C_3 \) | \( I_4 \) |

<table>
<thead>
<tr>
<th>((v_2, v_1, v_0))</th>
<th>BDD Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_0 ) ((0, 0, 0))</td>
<td>( v_2? 0 : (v_1? 0 : (v_0? 0 : 1)) )</td>
</tr>
<tr>
<td>( I_1 ) ((0, 0, 1))</td>
<td>( v_2? 0 : (v_1? 0 : (v_0? 1 : 0)) )</td>
</tr>
<tr>
<td>( I_2 ) ((0, 1, 0))</td>
<td>( v_2? 0 : (v_1? (v_0? 0 : 1) : 0) )</td>
</tr>
<tr>
<td>( I_3 ) ((0, 1, 1))</td>
<td>( v_2? 0 : (v_1? (v_0? 1 : 0) : 0) )</td>
</tr>
<tr>
<td>( I_4 ) ((1, 0, -))</td>
<td>( v_2? (v_1? 0 : 1) : 0 )</td>
</tr>
<tr>
<td>( I_5 ) ((1, 1, -))</td>
<td>( v_2? (v_1? 1 : 0) : 0 )</td>
</tr>
</tbody>
</table>
Boolean Optimization of Logic Networks
(Collaboration with Jean-Michel Puiatti)

- Complete sums are generated for Boolean predicate expressions using the iterative-consensus method. [Wakerly, Prentice94]
- Subsequently, a sum of products comprising a minimal cover is selected.
- These algorithms are expensive; for large problems, a heuristic approach must be used.
- Iterative consensus repeatedly applies the theorem \( x \lor \overline{x}y = x \lor y \).

\[
p3 = C_0(C_1 \lor \overline{C}_1(C_2 \lor \overline{C}_2C_3))
\]
\[
p3 = C_0(C_1 \lor \overline{C}_1(C_2 \lor C_3))
\]
\[
p3 = C_0(C_1 \lor C_2 \lor C_3)
\]
Optimization of Logic Network with Condition Relations

- Due to inherent logical relationships between conditions, even optimized predicate expressions often contain redundant literals, redundant product terms, and constant-false product terms.

- Condition optimization applies information from the condition BDD to eliminate these inefficiencies.

\[ p_3 = C_0(C_1 \lor C_2 \lor C_3) \]
\[ C_0 = (r1 \geq 0), \quad C_1 = (r1 == 1), \quad C_2 = (r1 == 2), \quad C_3 = (r1 == 3) \]
\[ C_1 \lor C_2 \lor C_3 \Rightarrow C_0 \]
\[ p_3 = C_1 \lor C_2 \lor C_3 \]
Effect of Boolean techniques and condition-based optimizations:

- This logic subnetwork is reduced from seven to two gates.
- Height is also reduced from four gate delays to one.

Intelligent define generation is required to capitalize on this gain.
Regeneration

- Another area in which solutions from CAD can be applied to the new framework.

- Optimization of predicate defines can be formulated as a logic synthesis problem.
  - Predicate definitions are analogous to gates. They consume resources.
  - Predicate computation height is analogous to total gate delay.

- Idea is to compute the result as quickly as possible, while consuming resources prudently.

- Complicating Factors:
  - Inputs may be available at different times.
  - Resource availability changes with the schedule.
# Factorization During Regeneration

<table>
<thead>
<tr>
<th>Condition</th>
<th>Path</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r1 &lt; 0)</td>
<td>6</td>
<td>6 cycles</td>
</tr>
<tr>
<td>(r1 &gt;= 0, r8 == 0)</td>
<td>5</td>
<td>5 cycles</td>
</tr>
<tr>
<td>(r1 &gt;= 0, r8 != 0)</td>
<td>6</td>
<td>6 cycles</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>5.8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th>Paths</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r1 &lt; 0)</td>
<td>30%</td>
<td>5 cycles</td>
</tr>
<tr>
<td>(r1 &gt;= 0, r8 == 0)</td>
<td>20%</td>
<td>2 cycles</td>
</tr>
<tr>
<td>(r1 &gt;= 0, r8 != 0)</td>
<td>50%</td>
<td>5 cycles</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>60%</td>
<td>4.4 cycles</td>
</tr>
</tbody>
</table>

David I. August  
IMPACT Research Group
• 8-issue: 1 branch, no branch penalty. Code is still aggressively predicated.

• Performance of demonstrated technique and theoretical maximum.
Reverse If-Conversion

- At this point, the reformulated code’s decision component is in predicated code.

- Balance between control flow and predication for target machine:
  - Predication, do Partial Reverse If-Conversion. [August, Micro30]
  - No predication, do Full Reverse If-Conversion. [Warter, PLDI93]
A Benefit of the Partial Reverse If-Conversion Framework

- Framework enables complex control flow transformation to be performed by simple data flow optimizations.
- Analogous to a Fourier transform:
  - Transform all code into the data flow domain.
  - Simple operations are performed.
  - Inverse Transform selected code into control flow domain.
- Loop versioning, loop fusion, if-then-else interchange, if-then-else fusion, and others possible with current data flow optimizations.
## Partial Reverse If-Conversion Applied

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_3_{\text{ot}}, p_6_{\text{af}} = (r1 == 1) )</td>
<td>COMP D0</td>
</tr>
<tr>
<td>( p_2_{\text{uf}} = (r1 &gt;= 0) )</td>
<td>COMP B</td>
</tr>
<tr>
<td>( p_7_{\text{ot}} = (r4 &gt; r5) )</td>
<td>COMP D1</td>
</tr>
<tr>
<td>( p_8_{\text{ut}} = (r2 &gt; r3) )</td>
<td>COMP D4</td>
</tr>
</tbody>
</table>

(r1 < 0) path [30%]: 5 cycles  
(r1 >= 0, r8 == 0) path [20%]: 2 cycles  
(r1 >= 0, r8 != 0) path [50%]: 5 cycles  

Total: 4.4 cycles

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1_{\text{ut}}, p_2_{\text{uf}} = (r1 &gt;= 0) )</td>
<td>COMP D1</td>
</tr>
<tr>
<td>( p_3_{\text{ot}}, p_6_{\text{af}} = (r1 == 1) )</td>
<td>COMP B</td>
</tr>
<tr>
<td>( p_7_{\text{ot}} = (r4 &gt; r5) )</td>
<td>COMP A</td>
</tr>
<tr>
<td>( p_8_{\text{ut}} = (r2 &gt; r3) )</td>
<td>COMP D3</td>
</tr>
</tbody>
</table>

(r1 < 0) path [30%]: 4 cycles  
(r1 >= 0, r8 == 0) path [20%]: 3 cycles  
(r1 >= 0, r8 != 0) path [50%]: 3 cycles  

Total: 3.3 cycles  

David I. August  
IMPACT Research Group
Conclusions

- Traditional branch handling techniques have not been able to dramatically restructure program control.

- Early introduction of predication into the compilation process enables simple yet effective optimization techniques.

- The proposed program decision logic optimization achieves near optimal performance and is closely limited by program computation.

- Partial reverse if-conversion balances predication with control flow at a level appropriate for the target architecture.

- The combination of decision logic optimization and partial reverse if-conversion shows promise for achieving a new level of ILP. Results for combined techniques are pending.
Future Directions

- Related work:
  - Explore the promise of the program decision logic optimization in the partial reverse if-conversion framework.
  - Apply lessons learned about branch handling to function boundaries.
  - Computation reformulation in the presence of memory operations.

- Exploration of EPIC:
  - Use EPIC to reduce cost of I-Cache in embedded systems. [Mahlke]
  - Use Information Theory and EPIC to explore sampling of other architectural state to break the asymptotic trend in branch prediction. [Connors]
  - Systematically explore the application of EPIC approach to all costly hardware structures.

- Other Collaboration:
  - Use powerful range analysis to enhance value prediction. [Sias]
  - Capitalize on lessons learned in Information, CAD, and Compiler Theory to various domains in computer architecture.
For Further Information

- **Illinois Microprocessor Project using Advanced Compiler Technology (IMPACT)**
  - GOAL: To provide critical research, architecture expertise, and compiler prototypes for the microprocessor industry.
  - Compiler framework is a product of many graduate students during the last ten years under the direction of Professor Wen-mei Hwu.
  - Information available at: [http://www.crhc.uiuc.edu/IMPACT/](http://www.crhc.uiuc.edu/IMPACT/)

- **Trimaran**
  - GOAL: To promote architecture research in the academic community by providing a freely available advanced compiler framework.
  - Composed of the Illinois IMPACT, Hewlett-Packard Laboratories CAR, and NYU ReaCT-ILP research and development teams.
Application of Condition Queries to Factorization

- Since condition relationships are not explicit in predicate define expressions, the condition BDD must be consulted to find all possible factorizations.

- Unifying the predicate and condition BDD’s would have the benefit of making these relationships explicit, but would complicate regeneration.

Example: Simplified terms, separate predicate and condition BDD’s:

\[
p1 = \begin{cases} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{cases} \quad p2 = \begin{cases} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{cases}
\]

\[d \Rightarrow c\] allows the factorization: \( p2 = p1 \cdot d \)

Combined predicate and condition BDD’s:

\[
c = \{1, -\} \quad p1 = \begin{cases} a & b & v_0 & v_1 \\ - & 1 & 1 & 1 \\ 1 & - & 1 & 1 \end{cases} \quad p2 = \begin{cases} a & b & v_0 & v_1 \\ - & 1 & 1 & 1 \\ 1 & - & 1 & 1 \end{cases}
\]

Here, the possibility of factorization is obvious, but the condition to be used is not explicit.
Performance of Partial Reverse If-Conversion Framework

- Results are baseline without any decision logic optimization.
- 4-issue: 1 branch, 2 integer, 2 memory, and 1 float. No branch penalty.
Partial Reverse If-Conversion Results

No Branch Prediction Penalty:

4 Cycle Branch Predication Penalty:
Original IMPACT Predication Framework

- Early heuristic hyperblock formation estimates final code characteristics:

Original Code → Hyperblock Formation (heuristic) → Optimizations (Predicate, ILP, and Traditional) → Scheduling
Problems with Original Predication Framework

- Heuristic to estimate final schedule can be inaccurate.
  - Unpredictable nature of interaction between resources and dependences
  - Subsequent optimizations

- Partial Path Inclusion - Estimates provided by heuristic not fine-grained enough to deal with partial paths.

- Complex Control Flow - Large number of path combinations can further complicate selection.

- Non-aggressive hyperblock formation may limit scheduling and optimization potential.
The Optimization Problem

- Optimization changes a good hyperblock decision into a poor one:
The If-Conversion During Scheduling Framework

• Excessive Scheduler Complexity
  – ILP Scheduling already complex
  – Forces almost all optimizations into the scheduler

Original Code  Scheduled Code

Scheduling
Hyperblock Formation
Optimizations (Predicate, ILP, and Traditional)
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:  
<TRUE>  
<p1>    <p2>  
T1, T2, T3 are not live out  
r1, r2, r3 are live out

BEFORE  

1  T2 = <TRUE>  
2  T1 = T2 <p1>  
3  <p2>  
4  r1 = <p2>  
5  T3 = T1 <TRUE>  
6  r2 = <TRUE>  
7  r3 = T3 <p1>  

AFTER  

1  T2 = <TRUE>  
2  T1 = T2 <p1>  
3  <p2>  
4  r1 = <p2>  
5  T3 = T1 <TRUE>  
6  r2 = <TRUE>  
7  r3 = T3 <p1>  
```

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

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

With Partial Reverse If-Conversion

Without Partial Reverse If-Conversion

David I. August
IMPACT Research Group
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
## 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>
Predicate Define Instructions

\[ pD_{0 \_type0}, \ pD_{1 \_type1} = (\text{src}_0 \ \text{cond} \ \text{src}_1) \langle p\text{SRC} \rangle \]

<table>
<thead>
<tr>
<th>pSRC</th>
<th>Comp</th>
<th>PlayDoh types</th>
<th>New types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UT</td>
<td>UF</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Predicate deposit semantics provide several modes in which condition and predicate values can be combined into predicate registers.
- Predicate define instructions are based on the HPL PlayDoh model [Kathail, HPL Tech.], with the addition of two new deposit types [August, ISCA26].
- New deposit types allow computation on arbitrary sets of condition and predicate values.
  - \(\lor T\) and \(\lor F\) compute disjunctions.
  - \(\land T\) and \(\land F\) compute conjunctions.