Systematic Program Decision Logic Optimization Using Predication

David I. August

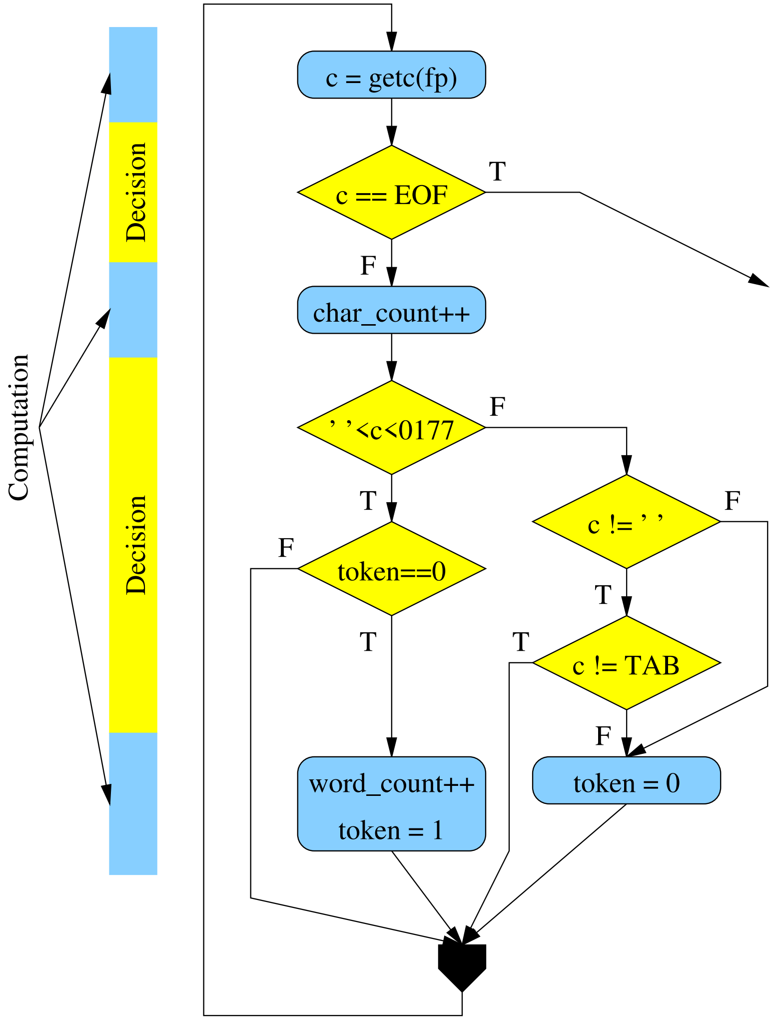
IMPACT Compiler Research Group

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:
    Global scheduling [Moon, Ebcioglu]
    Trace compilation framework [Fisher, Multiflow]
    Superblock compilation framework [IMPACT]
    Treegion compilation framework [Conte]
    PDG compilation framework [Ferrante, Gupta and Soffa]
  - Breaking control dependences:
    Compiler-directed control speculation [Multiflow, Mahlke, August]
  - Managing uncertainty of outcome:
    Branch prediction and out-of-order execution [Yeh and Patt, Smith]
    Multiscalar architectures [Sohi]

Program Optimization Techniques (cont.)

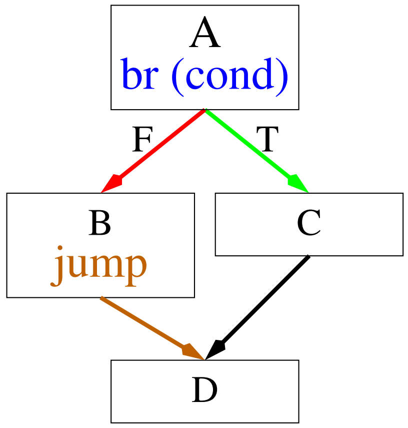
- Optimization of decision
  Control height reduction [Schlansker]
  Branch Combining [Mahlke]
  Tree-style branch VLIW architectures [Ebcioglu]
- Previous decision optimization techniques have taken a structural rather than a semantic approach.
- Much unexploited opportunity for optimization of decision:
  - 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



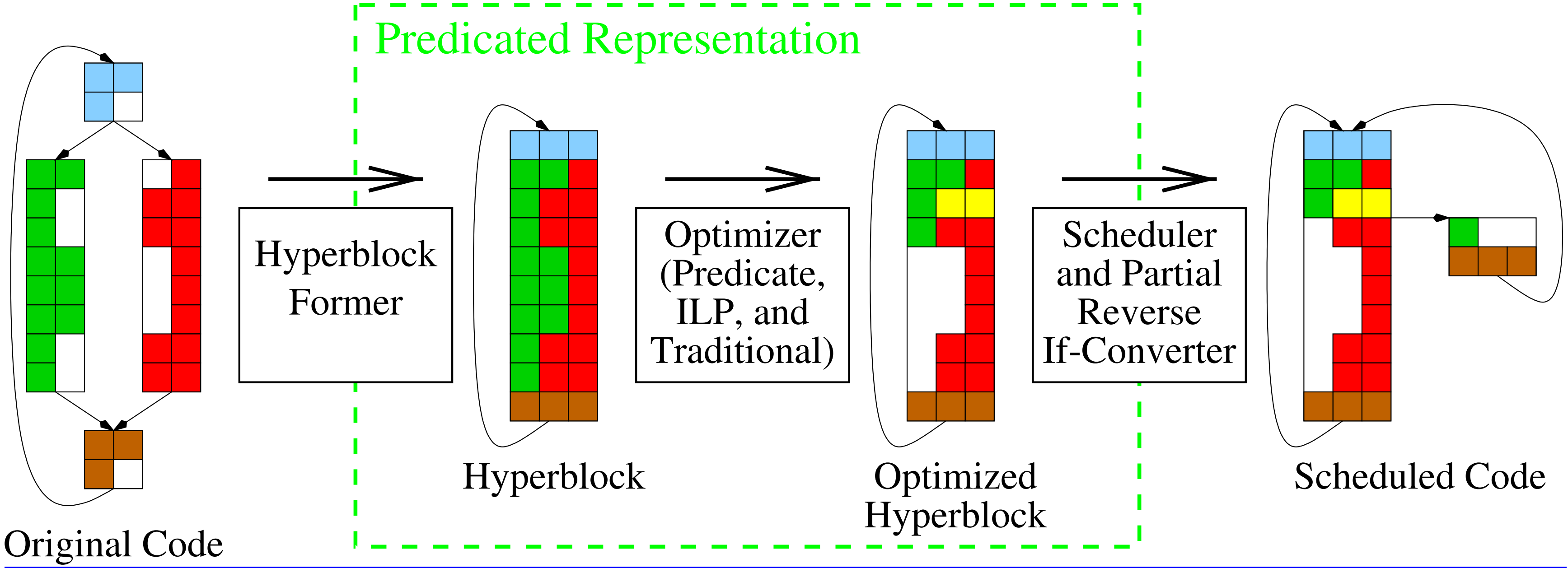
A
p1_T, p2_F = (cond)
B <p2>
C <p1>
D

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

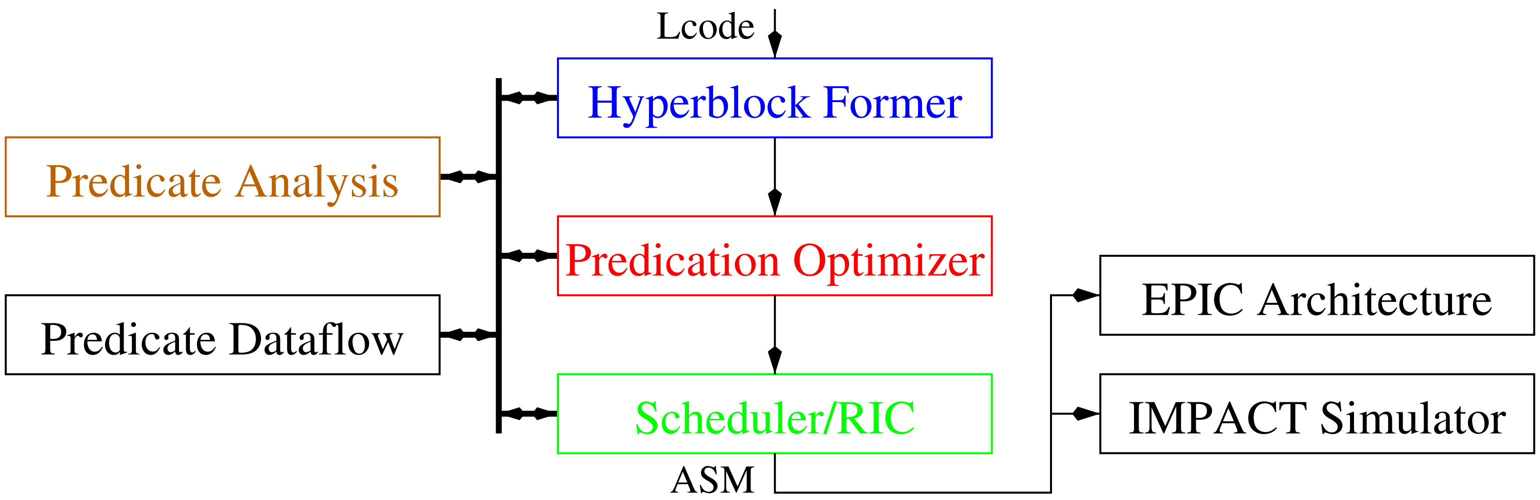
- Modeled after The Partial Reverse If-Conversion Framework. [August, Micro30]
- Early phase predication enables full use of Predicated Representation by optimizer.
- Complete early phase predication translates the program decision component into a single easily manageable form.



Program Decision Logic Optimization

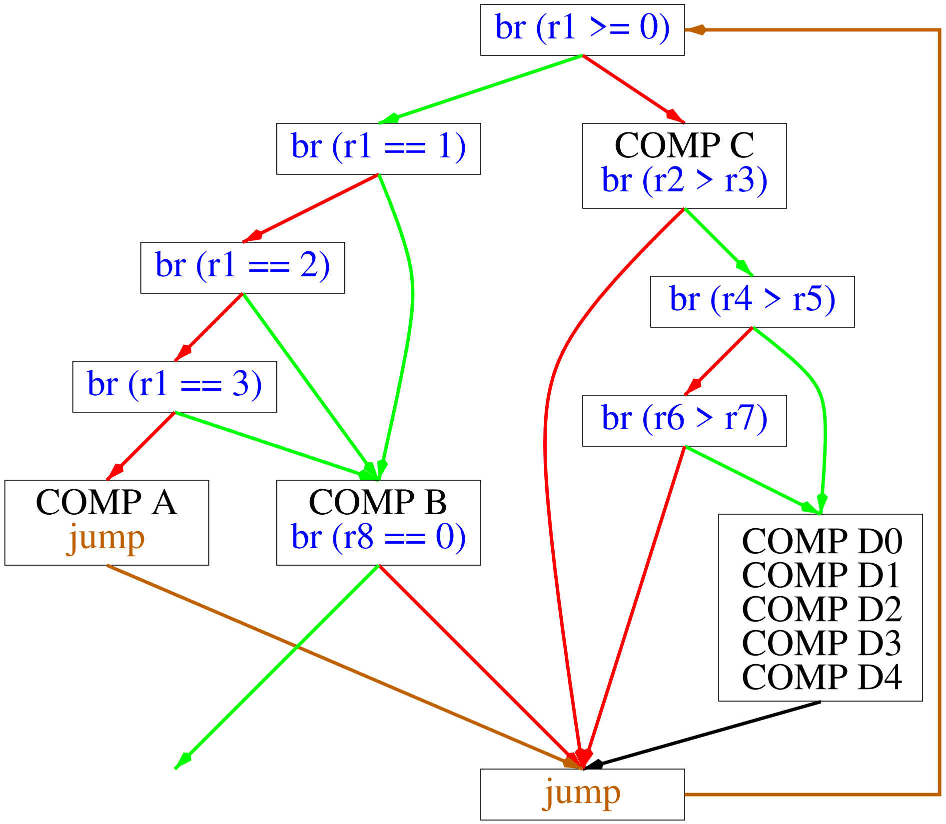
Steps in decision logic optimization:

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.



p1_ut, p2_uf = (r1 >= 0)	
p3_ot, p4_uf = (r1 == 1)	<p1>
p3_ot, p5_uf = (r1 == 2)	<p4>
p3_ot, p6_uf = (r1 == 3)	<p5>
COMP A	<p6>
COMP B	<p3>
br (r8 == 0)	<p3>
COMP C	<p2>
p7_ut = (r2 > r3)	<p2>
p8_ot, p9_uf = (r4 > r5)	<p7>
p8_ot = (r6 > r7)	<p9>
COMP D0	<p8>
COMP D1	<p8>
COMP D2	<p8>
COMP D3	<p8>
COMP D4	<p8>
jump	

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.

```
p1_ut, p2_uf = (r1 >= 0)
p3_ot, p4_uf = (r1 == 1)   <p1>
p3_ot, p5_uf = (r1 == 2)   <p4>
p3_ot, p6_uf = (r1 == 3)   <p5>
COMP A                     <p6>
COMP B                     <p3>
br (r8 == 0)               <p3>
COMP C, [r2 = ]            <p2>
p7_ut = (r2 > r3)          <p2>
p8_ot, p9_uf = (r4 > r5)   <p7>
p8_ot = (r6 > r7)          <p9>
COMP D0                    <p8>
COMP D1                    <p8>
COMP D2                    <p8>
COMP D3                    <p8>
COMP D4                    <p8>
jump
```

```
p1_ut, p2_uf = (r1 >= 0)
p3_ot, p4_uf = (r1 == 1)   <p1>
p3_ot, p5_uf = (r1 == 2)   <p4>
p3_ot, p6_uf = (r1 == 3)   <p5>
COMP A                     <p6>
COMP B                     <p3>
br (r8 == 0)               <p3>
COMP C, [r2 = ]
p7_ut = (r2 > r3)          <p2>
p8_ot, p9_uf = (r4 > r5)   <p7>
p8_ot = (r6 > r7)          <p9>
COMP D0
COMP D1
COMP D2
COMP D3
COMP D4                    <p8>
jump
```

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 $p1$ is a superset of $p2$:

$$\begin{array}{rcll} r1 & = & 10 & \langle p1 \rangle \\ r2 & = & r1 + 2 & \langle p2 \rangle \end{array} \Longrightarrow \quad r2 \; = \; 12 \;\; \langle p2 \rangle$$

- Analysis is complicated by limitless variations of predicate expressions:

$p1\_ut = cond1$ $p2\_ut = cond2 \; \langle p1 \rangle$ $\downarrow$ $p1 = cond1$ $p2 = cond1 \wedge cond2$	$p1\_ut, p2\_at = cond1$ $p2\_at = cond2$ $\downarrow$ $p1 = cond1$ $p2 = cond1 \wedge cond2$	$p2\_ut, p1\_ot = cond1$ $p1\_ot = cond2$ $\downarrow$ $p1 = cond1 \vee cond2$ $p2 = cond2$

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.].

$$\begin{aligned} p1\_ut &= cond1 \\ p2\_ut &= cond2 \;\langle p1 \rangle \end{aligned}$$

$$\begin{aligned} p1 &= cond1 \\ p2 &= cond1 \wedge cond2 \end{aligned}$$

$p2$ 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.

$$\begin{aligned} p2\_ut, p1\_\vee t &= cond1 \\ p1\_\vee t &= cond2 \\ p3\_ut &= cond3 \;\langle p2 \rangle \end{aligned}$$

$$\begin{aligned} p1 &= cond1 \vee cond2 \\ p3 &= cond1 \wedge cond3 \end{aligned}$$

$p1$ is not an ancestor of $p3$, but $p1$ is a superset of $p3$.

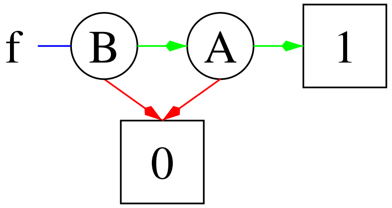
- Predicate Query System (PQS), used in the *Elcor* compiler at HP Labs, makes approximations in other ways [Johnson, Micro29].

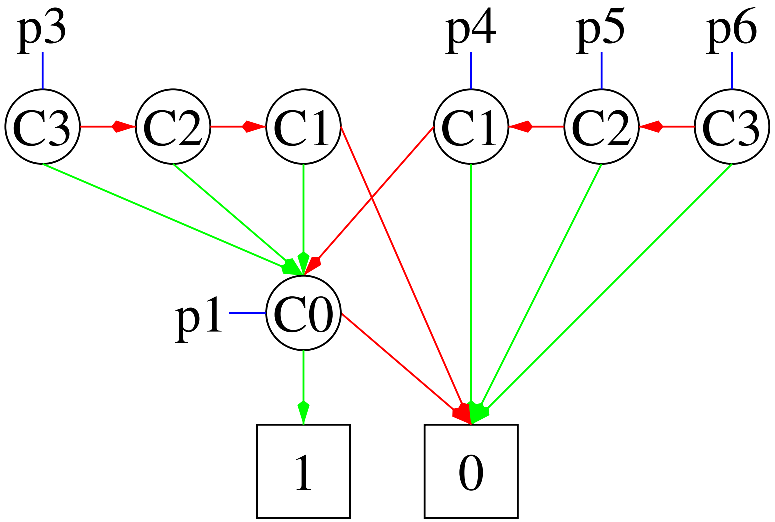
The Predicate Analysis System (PAS)

- 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 *predicate BDD* is constructed.

p1_ut, p2_uf = (r1 >= 0)	
p3_ot, p4_uf = (r1 == 1)	<p1>
p3_ot, p5_uf = (r1 == 2)	<p4>
p3_ot, p6_uf = (r1 == 3)	<p5>
COMP A	<p6>
COMP B	<p3>

C0: r1 >= 0
C1: r1 == 1
C2: r1 == 2
C3: r1 == 3





$$p1 = C0$$
$$p4 = \overline{C1} \wedge p1 = \overline{C1} \wedge C0$$

$$f = A \wedge B = B \ ? \ (A \ ? \ 1 : 0) : 0$$

The Condition Analysis System

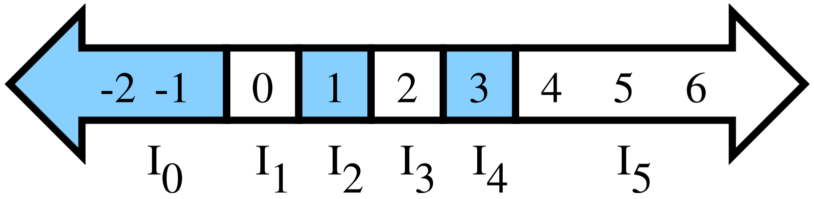
- Comparisons (*i.e.* (r1 $<$ 10), (r1 $>$ r2))
  - Comprise the interface between computation and decision.
  - Are the input variables of the predicate BDD.
- Conditions may be inherently related: (*i.e.* (r1 $<$ 5) implies (r1 $<$ 10))
- Condition relationships create additional predicate relationships.
- The Predicate BDD stores relations among predicates.
- The *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.

$$p1 = C1 \wedge C2 \Rightarrow (r1 > 10) \wedge (r1 < 20) \Rightarrow (10 < r1 < 20)$$

Range Determination
(Collaboration with John Sias)

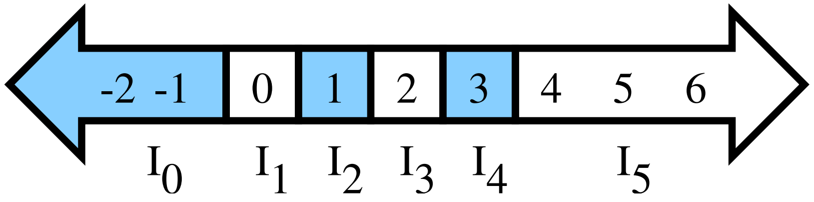
- All register/constant comparisons based on the same register compose a *family*.
- Within each family, the logical interrelation of conditions can be expressed.
- For each family, a number line is divided into disjoint intervals.
- Conditions are composed by a union of intervals.
- In our example:

$$
\begin{array}{lll}
C0 & (\mathtt{r1} \geq 0) & \bigcup_{j=1}^{5} I_j \\
C1 & (\mathtt{r1} = 1) & I_2 \\
C2 & (\mathtt{r1} = 2) & I_3 \\
C3 & (\mathtt{r1} = 3) & I_4
\end{array}
$$



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.



$\lceil \log_2 6 \rceil = 3$

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

	(v_2, v_1, v_0)	BDD Expression
I_0	$(0,0,0)$	$v_2 ? \, 0 : (v_1 ? \, 0 : (v_0 ? \, 0 : 1))$
I_1	$(0,0,1)$	$v_2 ? \, 0 : (v_1 ? \, 0 : (v_0 ? \, 1 : 0))$
I_2	$(0,1,0)$	$v_2 ? \, 0 : (v_1 ? \, (v_0 ? \, 0 : 1) : 0)$
I_3	$(0,1,1)$	$v_2 ? \, 0 : (v_1 ? \, (v_0 ? \, 1 : 0) : 0)$
I_4	$(1,0,-)$	$v_2 ? \, (v_1 ? \, 0 : 1) : 0$
I_5	$(1,1,-)$	$v_2 ? \, (v_1 ? \, 1 : 0) : 0$

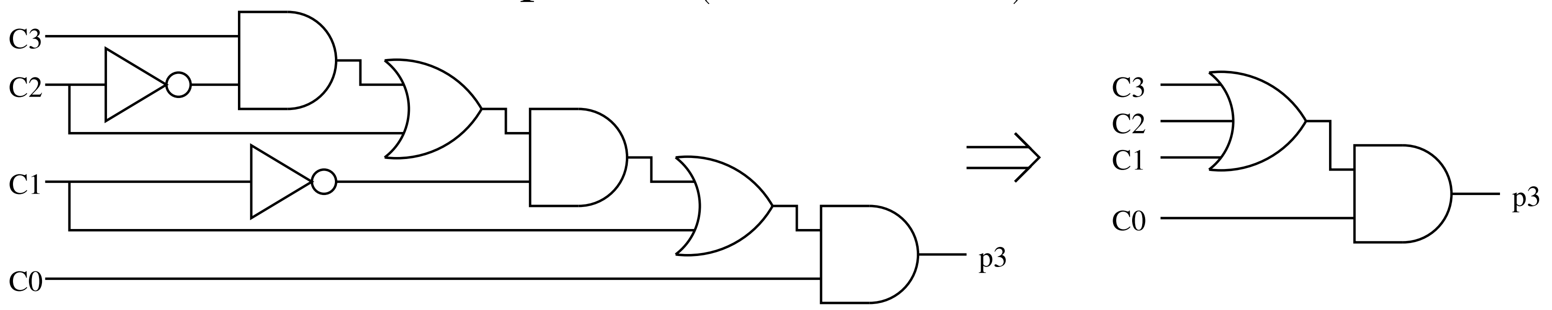
Cond	BDD Expression
$C0$	$I_5 ? \, 1 : (I_4 ? \, 1 : \ldots$ $\ldots (I_3 ? \, 1 : (I_2 ? \, 1 : I_1)))$
$C1$	I_2
$C2$	I_3
$C3$	I_4

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 $x \vee \overline{x}y = x \vee y$.

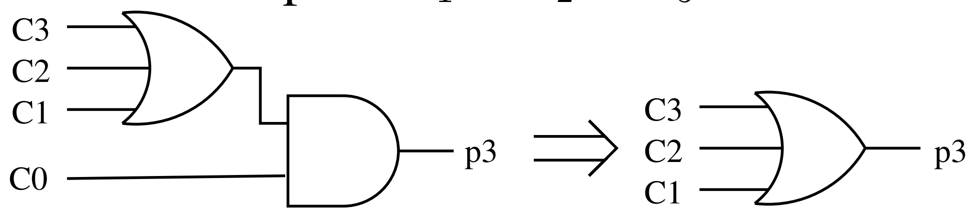
$$\text{p3} = C_0(C_1 \vee \overline{C}_1(C_2 \vee \overline{C}_2 C_3))$$
$$\text{p3} = C_0(C_1 \vee \overline{C}_1(C_2 \vee C_3))$$
$$\text{p3} = C_0(C_1 \vee C_2 \vee C_3)$$



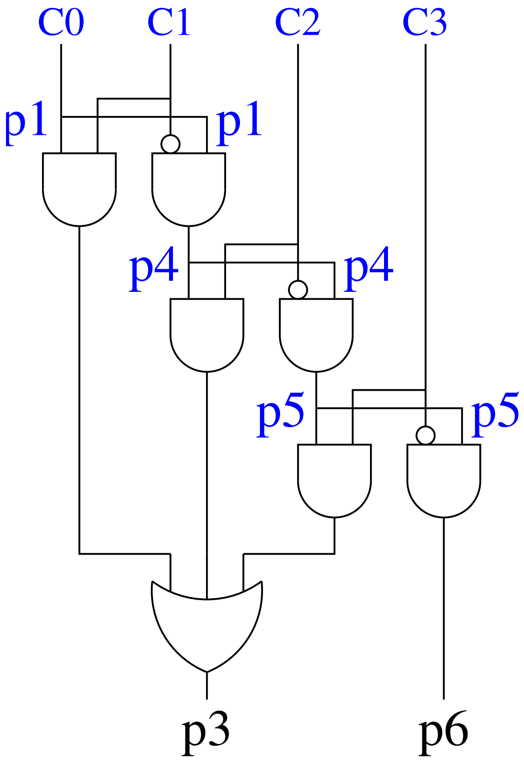
Optimization of Logic Network with Condition Relations

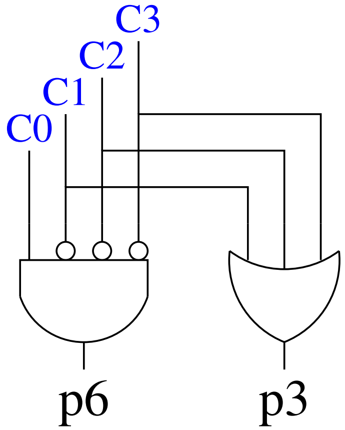
- Due to inherent logical relationships between conditions, even optimized predicate expressions often contain redundant literals, redundant product terms, and constant-false product terms.
- Condition optimization applies information from the condition BDD to eliminate these inefficiencies.

$$\text{p3} = C_0(C_1 \vee C_2 \vee C_3)$$
$$C_0 = (r1 \geq 0), C_1 = (r1 == 1), C_2 = (r1 == 2), C_3 = (r1 == 3)$$
$$C_1 \vee C_2 \vee C_3 \Rightarrow C_0$$
$$\text{p3} = C_1 \vee C_2 \vee C_3$$



Logic Optimization





- 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

p1_ut, p2_uf = (r1 >= 0)	COMP C, [r2 =]	COMP D0	COMP D1
p3_ot, p4_uf = (r1 == 1) <p1>	p7_ut = (r2 > r3) <p2>	COMP D2	COMP D3
p3_ot, p5_uf = (r1 == 2) <p4>	p8_ot, p9_uf = (r4 > r5) <p7>		
p3_ot, p6_uf = (r1 == 3) <p5>	p8_ot = (r6 > r7) <p9>		
COMP A <p6>	COMP B <p3>	COMP D4 <p8>	br (r8 == 0) <p3>
			jump

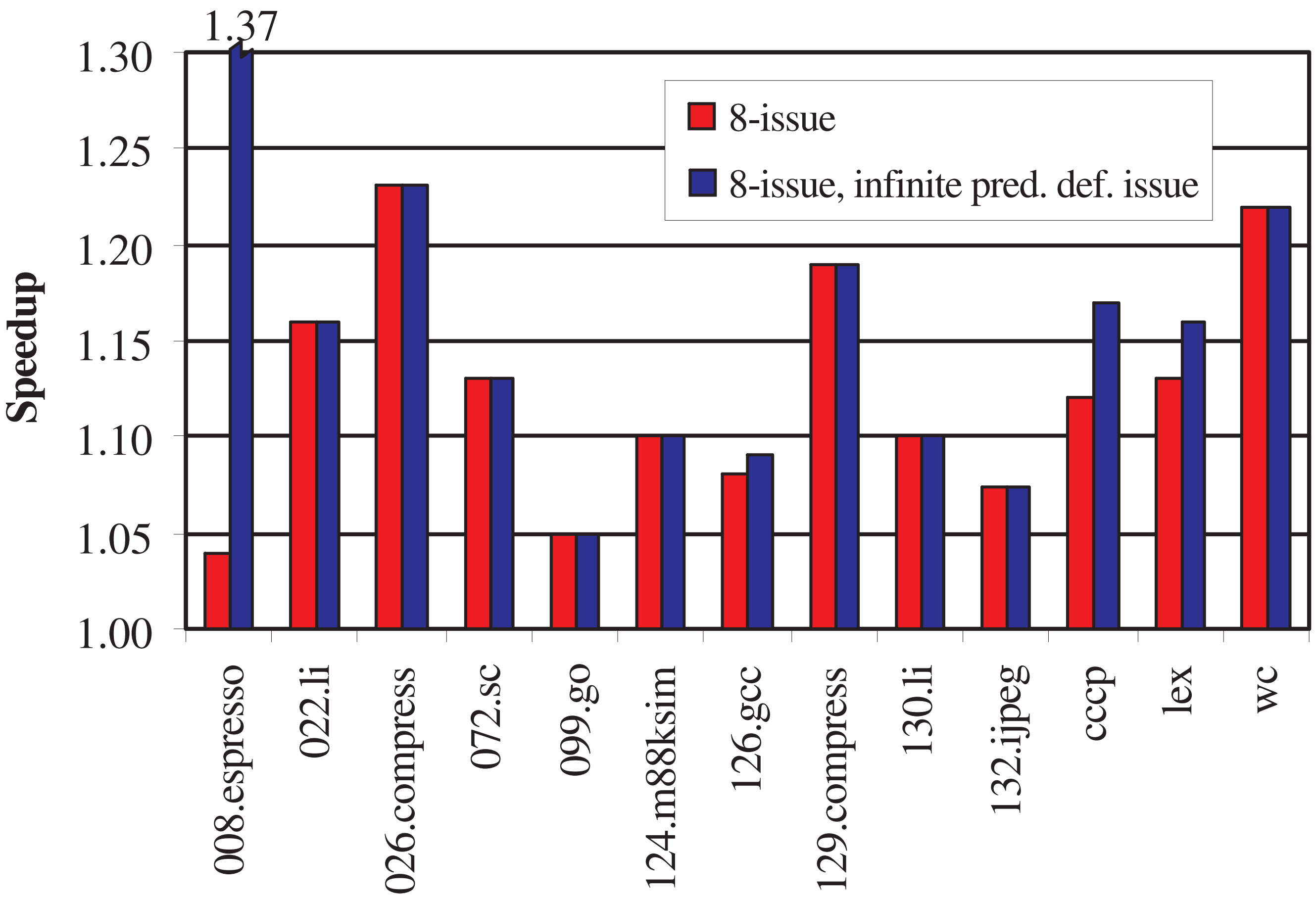
(r1 < 0) path [30%]:	6 cycles	
(r1 >= 0, r8 == 0) path [20%]:	5 cycles	
(r1 >= 0, r8 != 0) path [50%]:	6 cycles	
Total:	5.8 cycles	

p3_ot, p6_af = (r1 == 1)	p3_ot, p6_af = (r1 == 2)	p3_ot, p6_af = (r1 == 3)	COMP D0
p2_uf, p6_at = (r1 >= 0)	COMP B <p3>	COMP D1	br (r8 == 0) <p3>
p7_ot = (r4 > r5) <p2>	p7_ot = (r6 > r7) <p2>	COMP D2	COMP C, [r2=]
p8_ut = (r2 > r3) <p7>		COMP D3	COMP A <p6>
		COMP D4 <p8>	jump

(r1 < 0) path [30%]:	5 cycles	17%
(r1 >= 0, r8 == 0) path [20%]:	2 cycles	60%
(r1 >= 0, r8 != 0) path [50%]:	5 cycles	17%
Total:	4.4 cycles	24%

David I. August
IMPACT Research Group

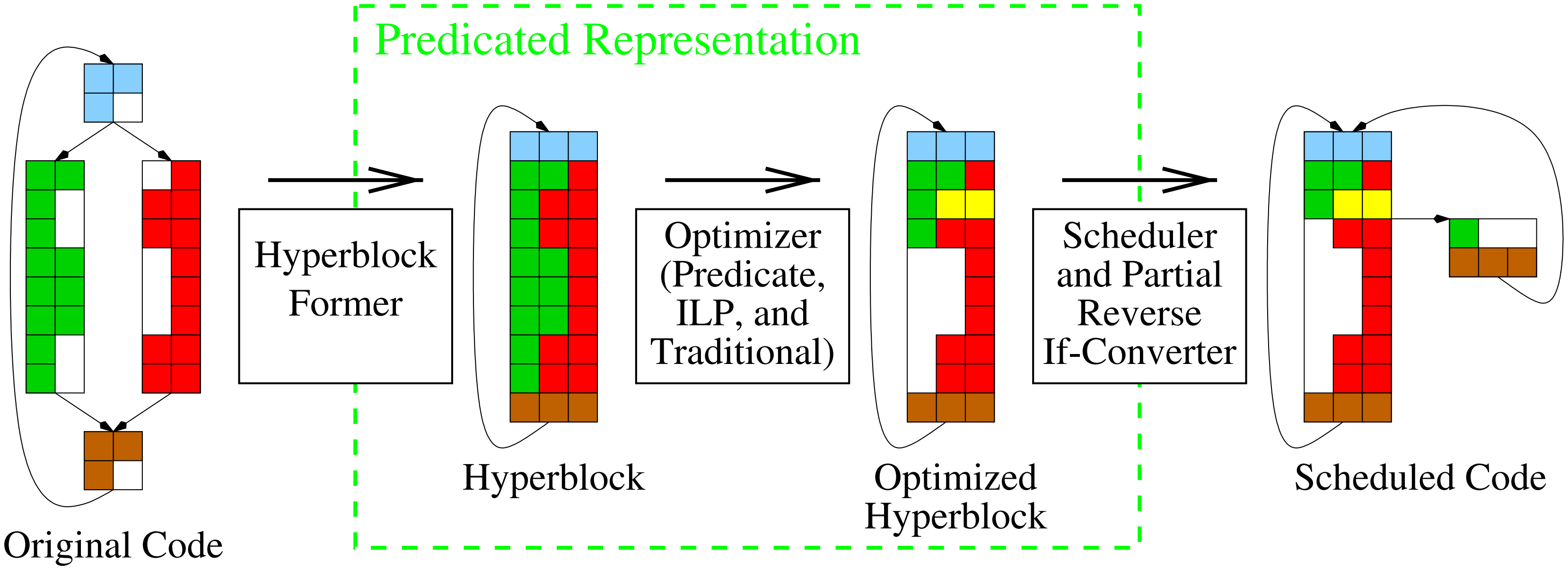
Speedup due to Logic Optimization



- 8-issue: 1 branch, no branch penalty. Code is still aggressively predicated.
- Performance of demonstrated technique and theoretical maximum.

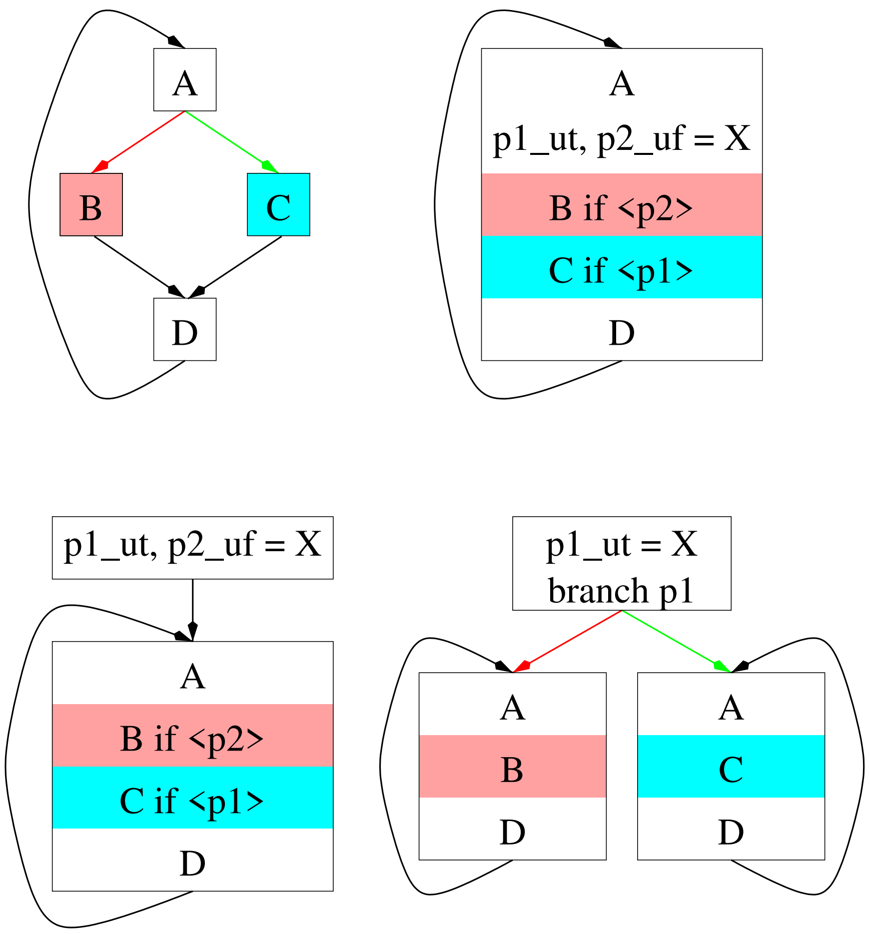
Reverse If-Conversion

- 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

p3_ot, p6_af = (r1 == 1)	p3_ot, p6_af = (r1 == 2)	p3_ot, p6_af = (r1 == 3)	COMP D0
p2_uf = (r1 >= 0)	COMP B <p3>	COMP D1	br (r8 == 0) <p3>
p7_ot = (r4 > r5) <p2>	p7_ot = (r6 > r7) <p2>	COMP D2	COMP C, [r2=]
p8_ut = (r2 > r3) <p7>		COMP D3	COMP A <p6>
		COMP D4 <p8>	jump

(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

p1_ut, p2_uf = (r1 >= 0)	p3_ot, p6_af = (r1 == 1)	p3_ot, p6_af = (r1 == 2)	p3_ot, p6_af = (r1 == 3)
p7_ot = (r4 > r5) <p2>	p7_ot = (r6 > r7) <p2>	COMP C, [r2=]	jump <p1>
p8_ut = (r2 > r3) <p7>	COMP D0	COMP D1	COMP D2
COMP D3	COMP D4 <p8>		jump

COMP A <p6>	COMP B <p3>	br (r8 == 0) <p3>	jump

(r1 < 0) path [30%]: 4 cycles 20%
(r1 >= 0, r8 == 0) path [20%]: 3 cycles +50%
(r1 >= 0, r8 != 0) path [50%]: 3 cycles 40%

Total: 3.3 cycles 25% (43% Overall)

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/*
- **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.
  - Information and compiler source available at: *http://www.trimaran.org/*

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:

$$p1 = \left\{ \begin{array}{cccc} a & b & c & d \\ - & 1 & 1 & - \\ 1 & - & 1 & - \end{array} \right. \qquad p2 = \left\{ \begin{array}{cccc} a & b & c & d \\ - & 1 & - & 1 \\ 1 & - & - & 1 \end{array} \right.$$

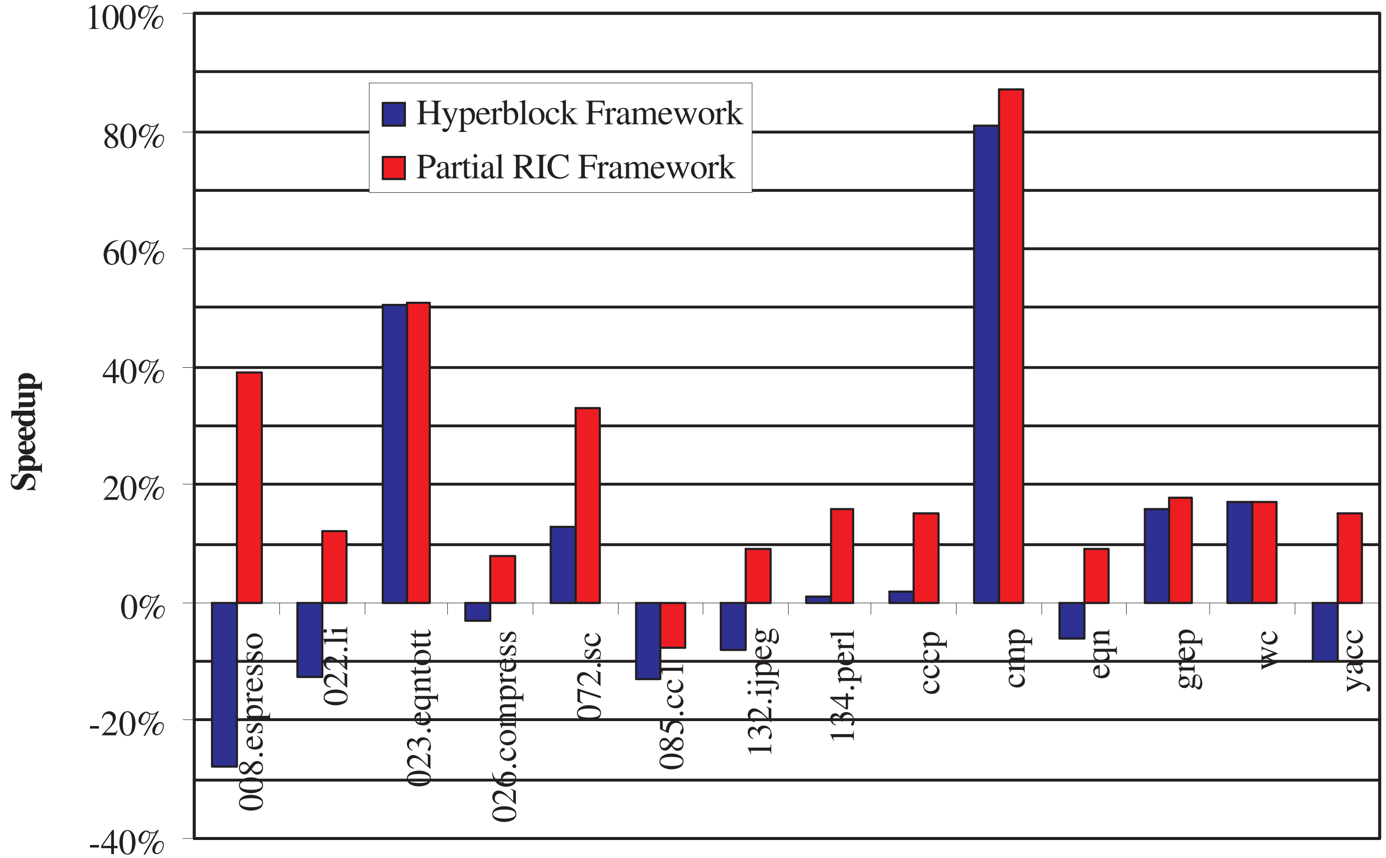
$d \Rightarrow c$ allows the factorization: $p2 = p1 \cdot d$

Combined predicate and condition BDD's:

$$\begin{array}{l} c = \{1, -\} \\ d = \{1, 1\} \end{array} \quad p1 = \left\{ \begin{array}{cccc} a & b & v_0 & v_1 \\ - & 1 & 1 & - \\ 1 & - & 1 & - \end{array} \right. \qquad p2 = \left\{ \begin{array}{cccc} a & b & v_0 & v_1 \\ - & 1 & 1 & 1 \\ 1 & - & 1 & 1 \end{array} \right.$$

Here, the possibility of factorization is obvious, but the condition to be used is not explicit

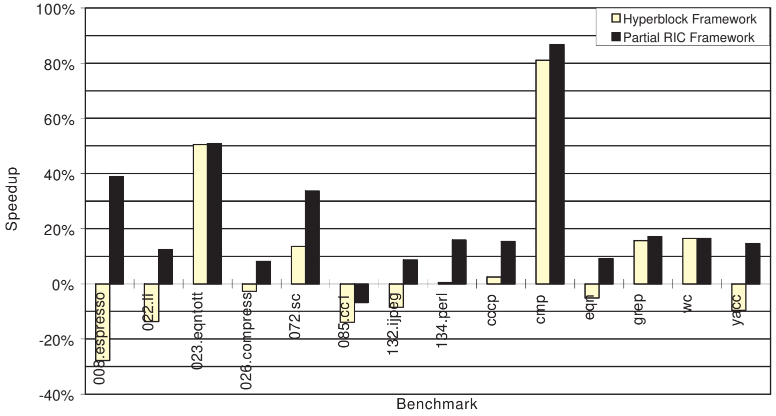
Performance of Partial Reverse If-Conversion Framework



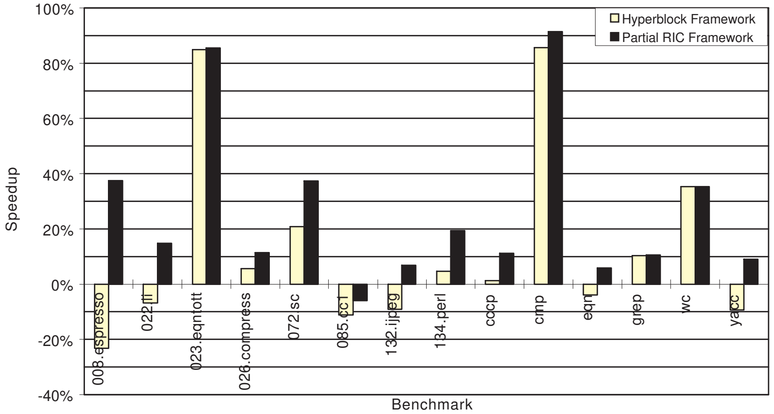
- Results are baseline without any decision logic optimization.
- 4-issue: 1 branch, 2 integer, 2 memory, and 1 float. No branch penalty.

Partial Reverse If-Conversion Results

No Branch Prediction Penalty:

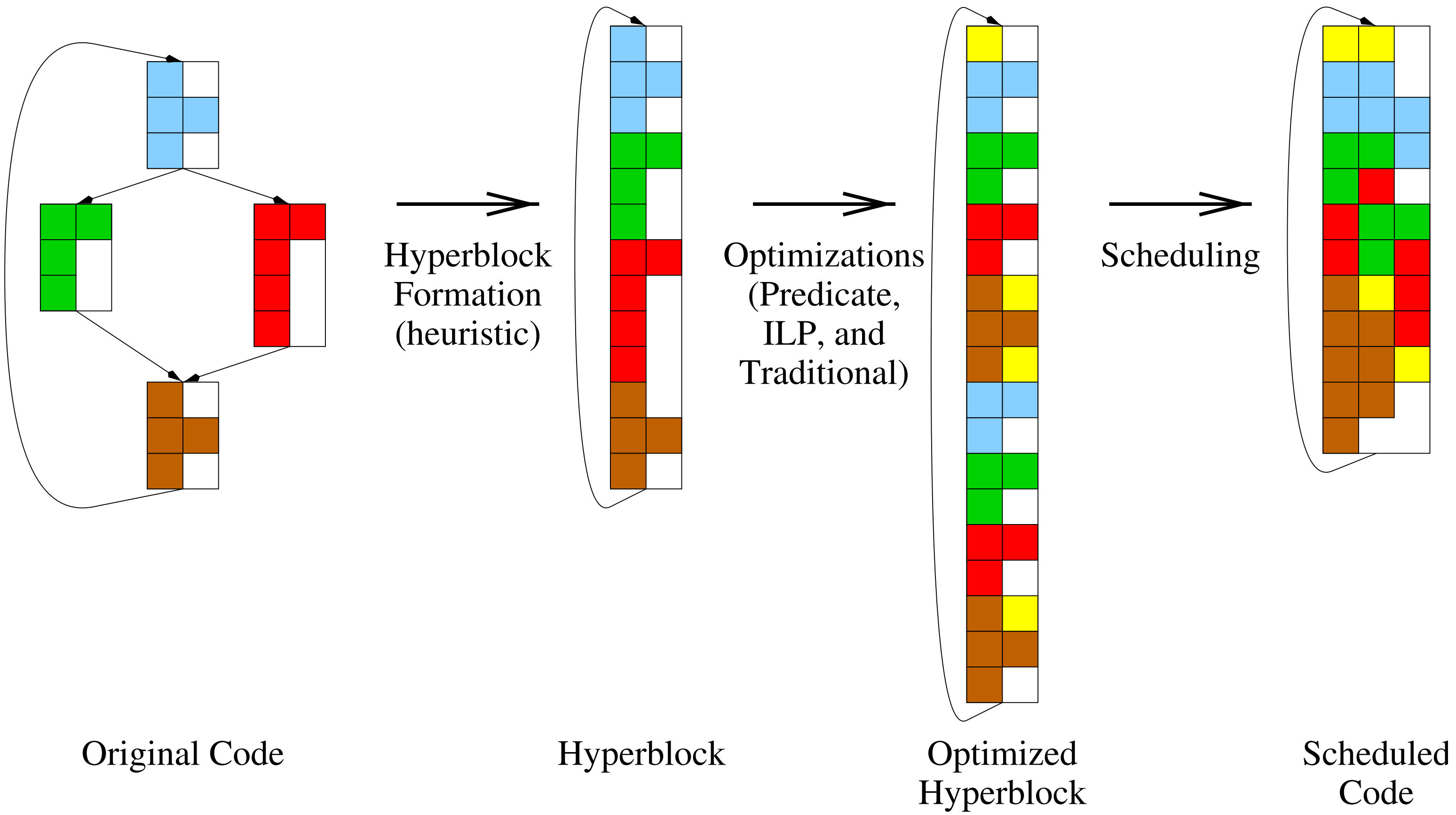


4 Cycle Branch Predication Penalty:



Original IMPACT Predication Framework

- Early heuristic hyperblock formation estimates final code characteristics:

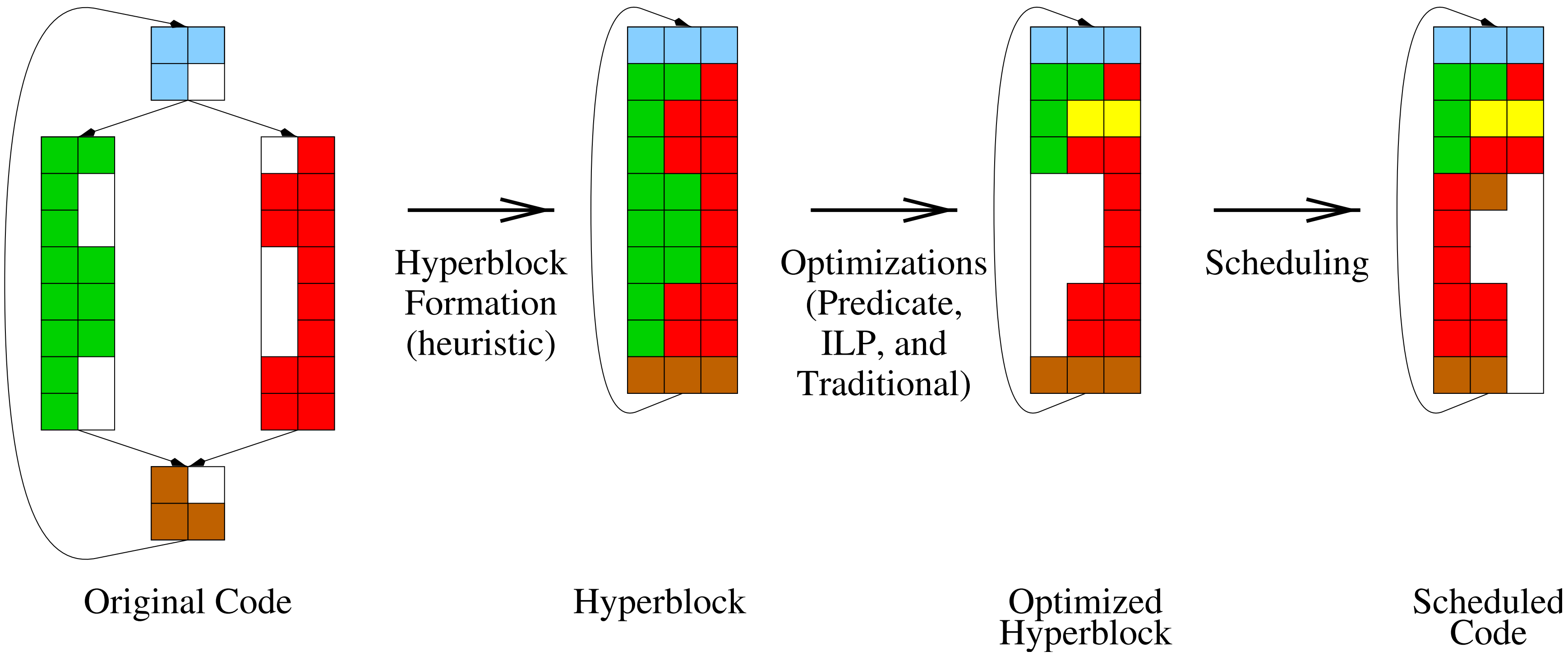


Problems with Original Predication Framework

- 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.

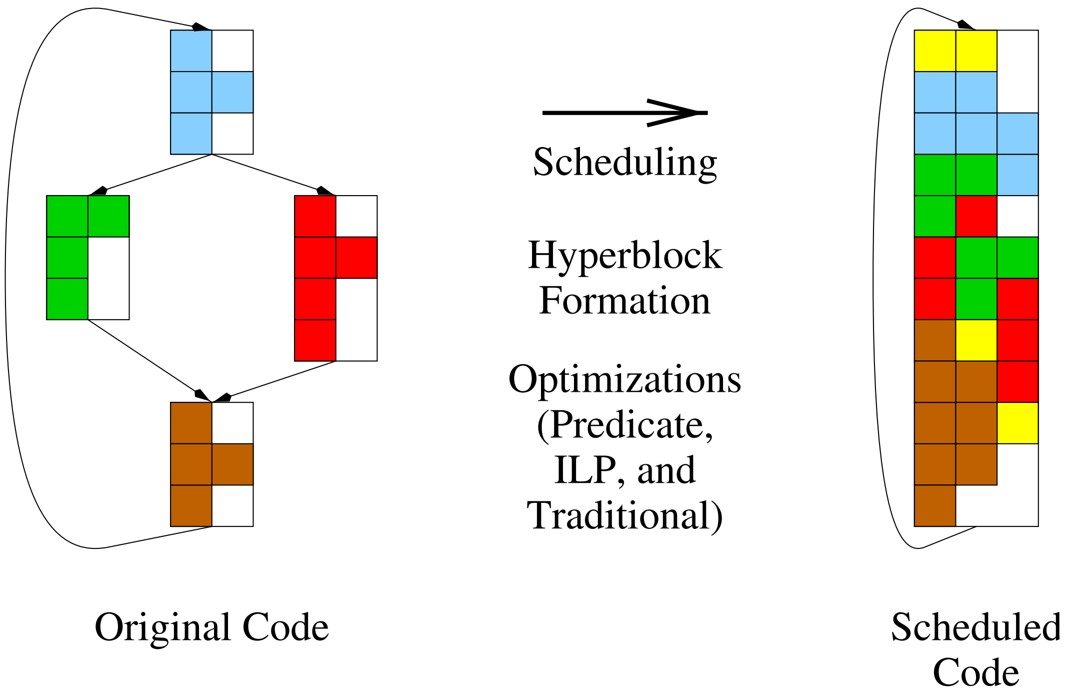
The Optimization Problem

- 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



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>

p2

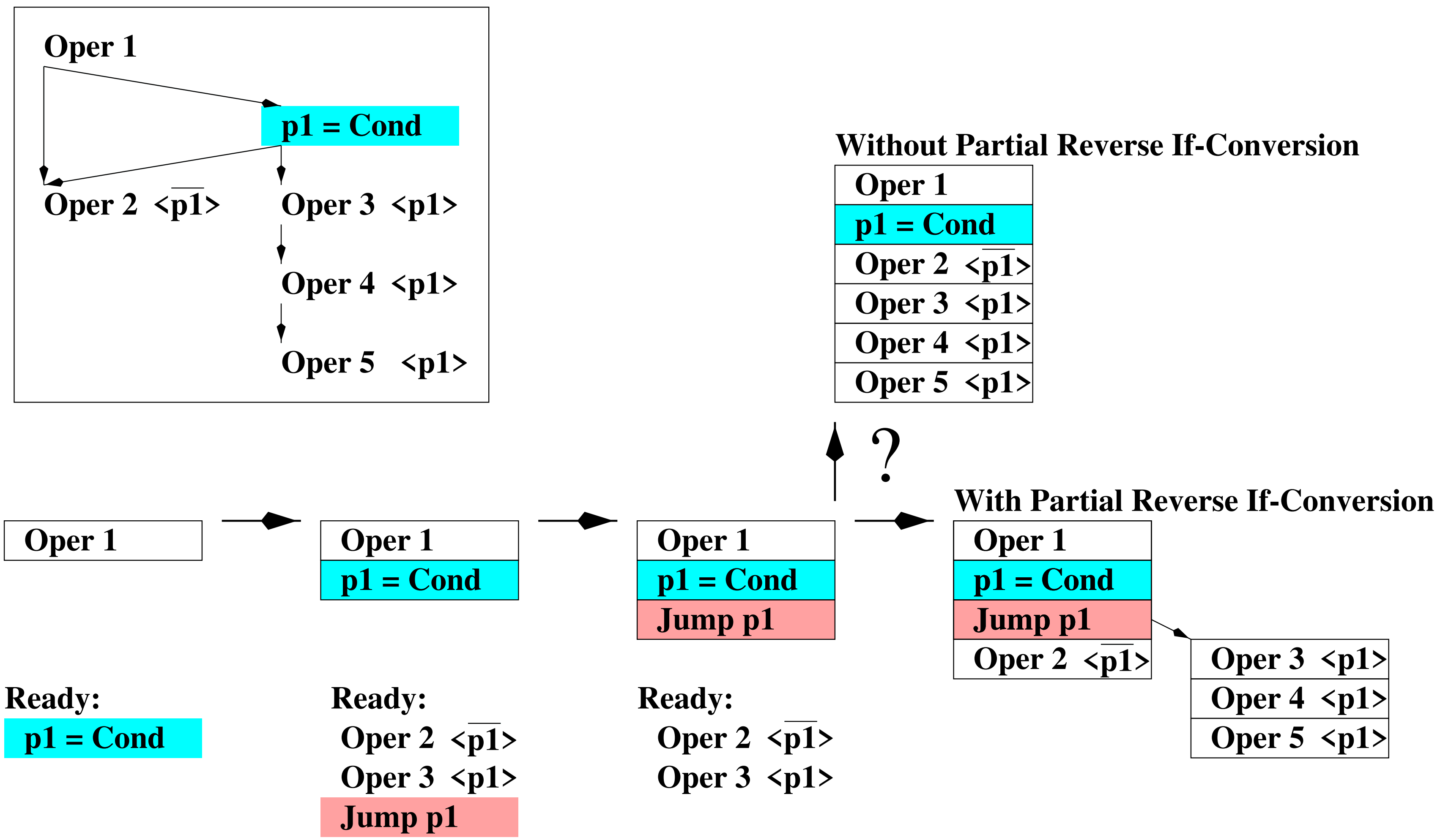
4	r1 =	<p2>
5	T3 = T1	<TRUE>
6	r2 =	<TRUE>
7	r3 = T3	<p1>

AFTER

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

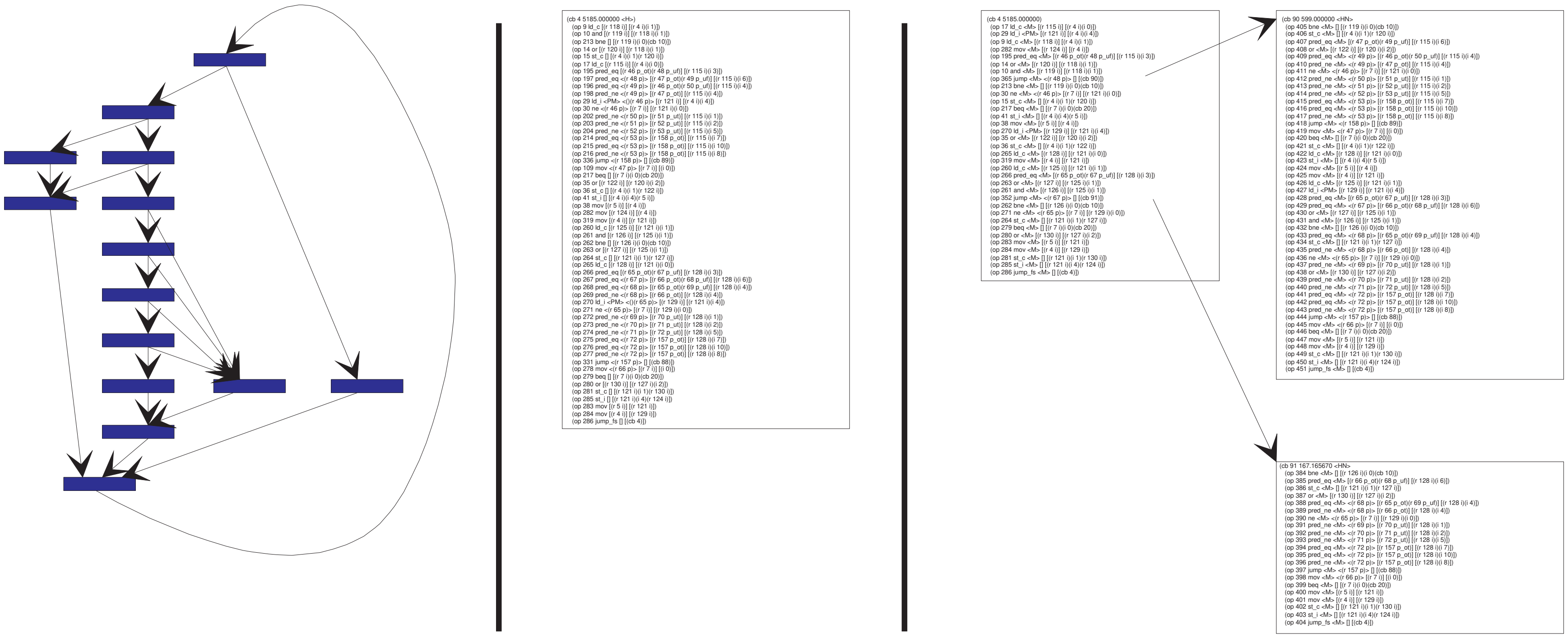
4	r1 =	<p2>
6	r2 =	<TRUE>

Partial Reverse If-Conversion Algorithm



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



David I. August
IMPACT Research Group

Application Statistics

Benchmark	Reverse If-Conversions	Opportunities
008.espresso	204	1552
022.li	50	393
023.eqntott	43	443
026.compress	11	56
072.sc	33	724
085.cc1	479	3827
132.ijpeg	134	1021
134.perl	42	401
cccp	77	1046
cmp	4	49
eqn	33	326
grep	3	103
wc	0	88
yacc	247	1976

Predicate Define Instructions

$$pD_0\_type_0,\ pD_1\_type_1 = (src_0\ \mathbf{cond}\ src_1)\ \langle pSRC\rangle$$

		PlayDoh types						New types			
pSRC	Comp	UT	UF	OT	OF	AT	AF	$\vee$T	$\vee$F	$\wedge$T	$\wedge$F
0	0	0	0	-	-	-	-	-	1	0	0
0	1	0	0	-	-	-	-	1	-	0	0
1	0	0	1	-	1	0	-	1	1	0	-
1	1	1	0	1	-	-	0	1	1	-	0

- 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.
  - $\vee T$ and $\vee F$ compute disjunctions.
  - $\wedge T$ and $\wedge F$ compute conjunctions.