A Global Predication Compilation Framework

David I. August

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

Outline

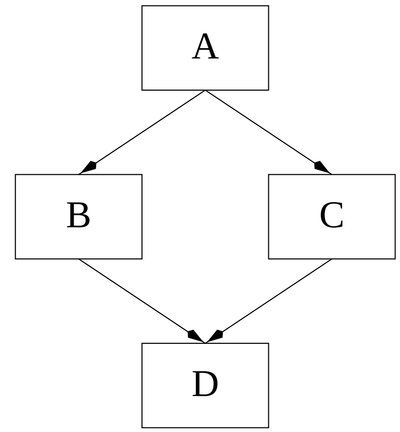
- Predication Background
- Predication Frameworks
- Predicate Optimization
  - Fully Resolved Predicates
  - Code Specialization
  - Control Logic Optimization
- Ultrablock Predication Framework
- Predicate Analysis
- Predicate Dataflow

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.



A if TRUE

B if P

C if $\overline{P}$

D if TRUE

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

$$P_{d0<type_0>}, P_{d1<type_1>} = (\ src_0\ \textbf{cond}\ src_1\ )\ \langle P_g \rangle$$

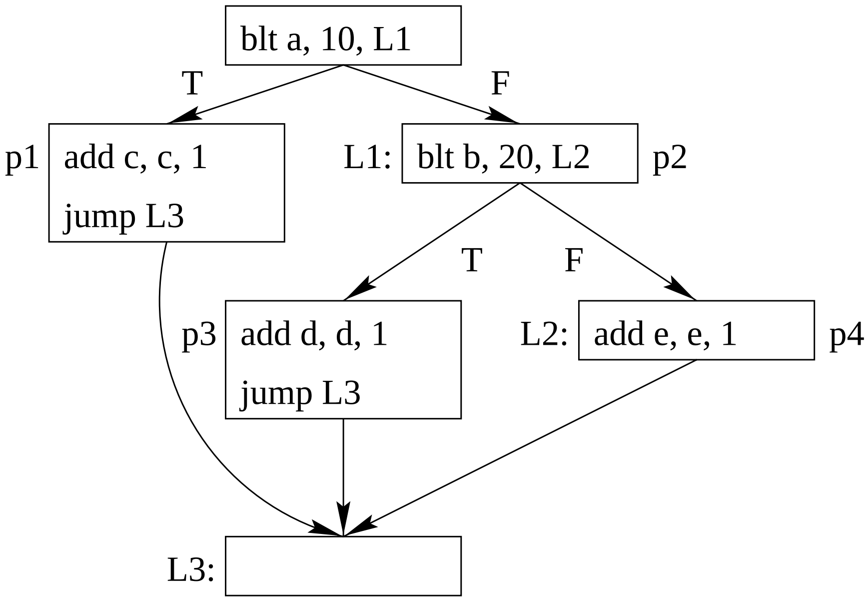
- $cond$ comparison: $=$, $<$, $\leq$, etc.
- $type_i$ assignment type:
  - UT/UF - Unconditional
  - OT/OF - Wired-or
  - AT/AF - Wired-and
  - CT/CF - Conditonal
  - $\vee$T/$\vee$F - Disjunctive
  - $\wedge$T/$\wedge$F - Conjunctive

Unconditional Predicate Define

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

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

$p1_{UT}$, $p2_{UF}$ = $(a < 10)$
c = c + 1 $\langle p1 \rangle$
$p3_{UT}$, $p4_{UF}$ = $(b < 20)$ $\langle p2 \rangle$
d = d + 1 $\langle p3 \rangle$
e = e + 1 $\langle p4 \rangle$



P_g	Comparison	P_d UT	P_d UF
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Wired-OR Predicate Define

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

```
if ( a && b )
   c = c + 1;
else
   d = d + 1;
```

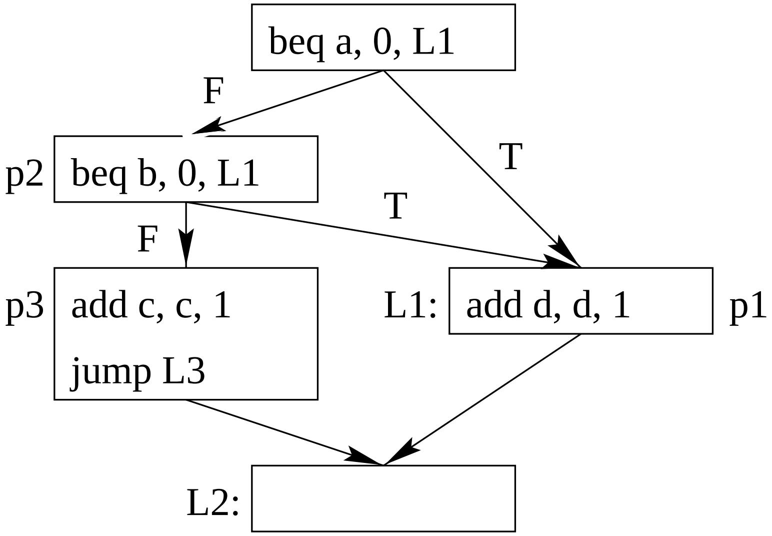
$p1$ = 0

$p1_{OT}, p2_{UF} = (a == 0)$

$p1_{OT}, p3_{UF} = (b == 0)\ \langle p2 \rangle$

c = c + 1 $\langle p3 \rangle$

d = d + 1 $\langle p1 \rangle$



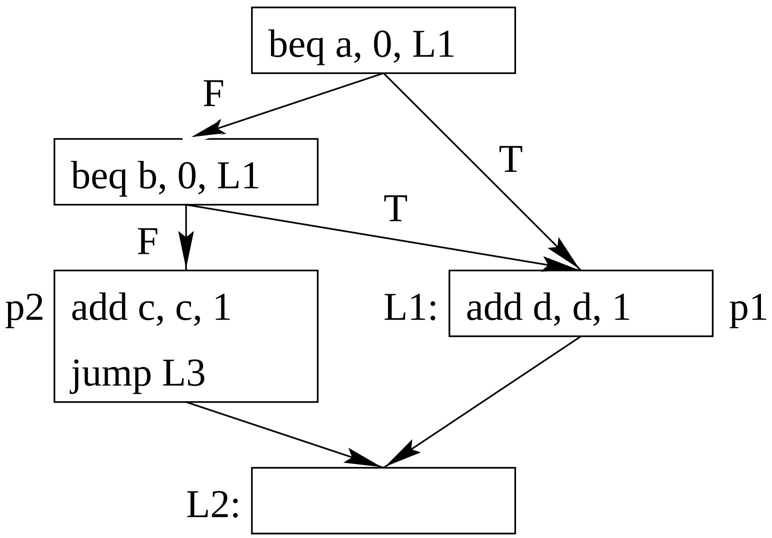
P_g	Comparison	P_d OT	P_d OF
0	0	–	–
0	1	–	–
1	0	–	1
1	1	1	–

Wired-AND Predicate Define

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

```
if ( a && b )
   c = c + 1;
else
   d = d + 1;
```

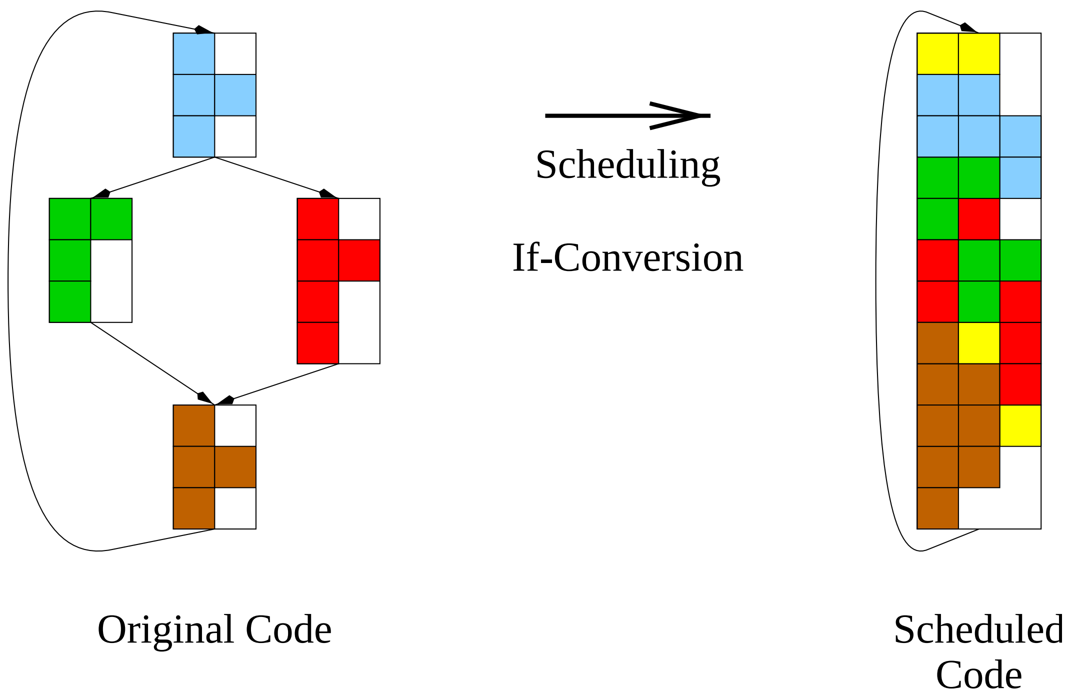
$p1$ = 0
$p2$ = 1
$p1_{OT}, p2_{AF}$ = $(a == 0)$
$p1_{OT}, p2_{AF}$ = $(b == 0)$
c = c + 1 $\langle p2 \rangle$
d = d + 1 $\langle p1 \rangle$



P_g	Comparison	P_d AT	P_d AF
0	0	-	-
0	1	-	-
1	0	0	-
1	1	-	0

The If-Conversion During Scheduling Framework

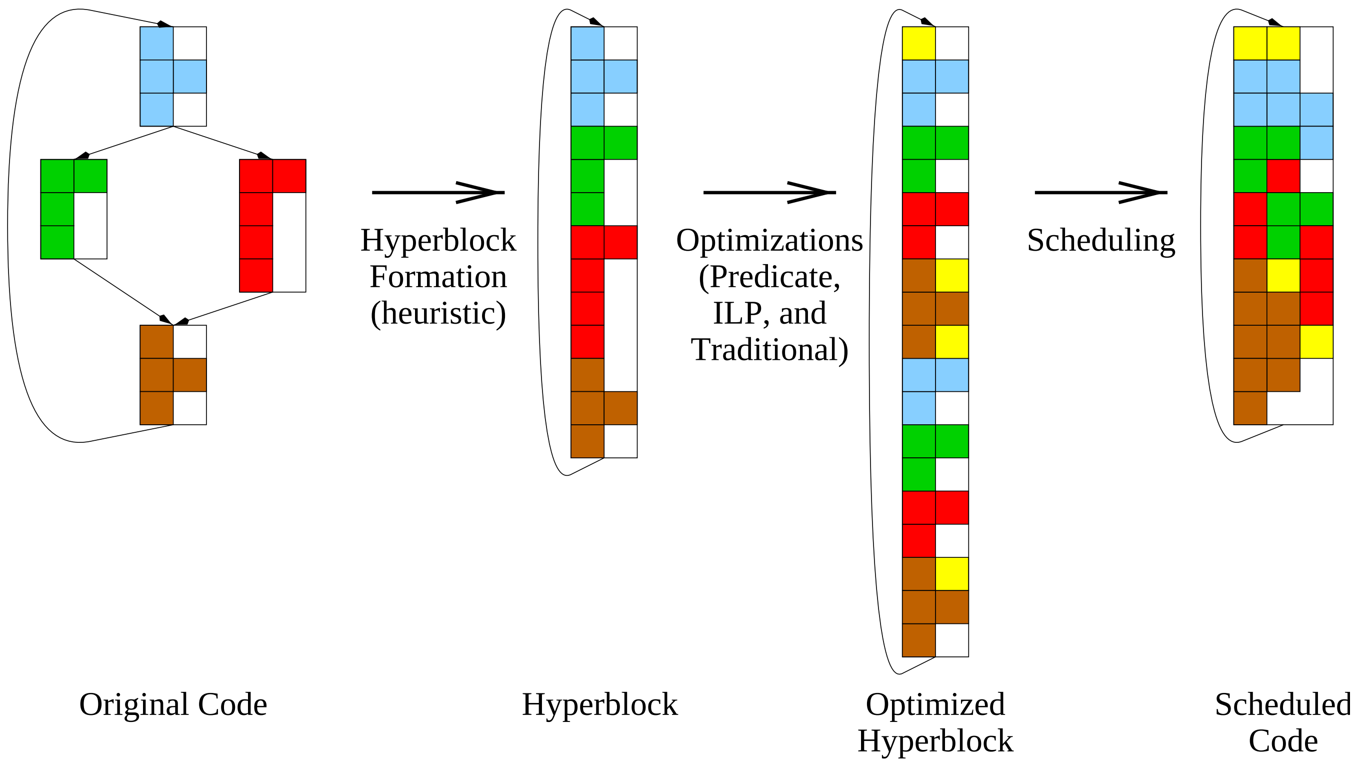
- Best time to balance control flow and predication
- Minimizes effect on existing compiler
- Naive - doesn't use predicated representation



David I. August
IMPACT Research Group

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:

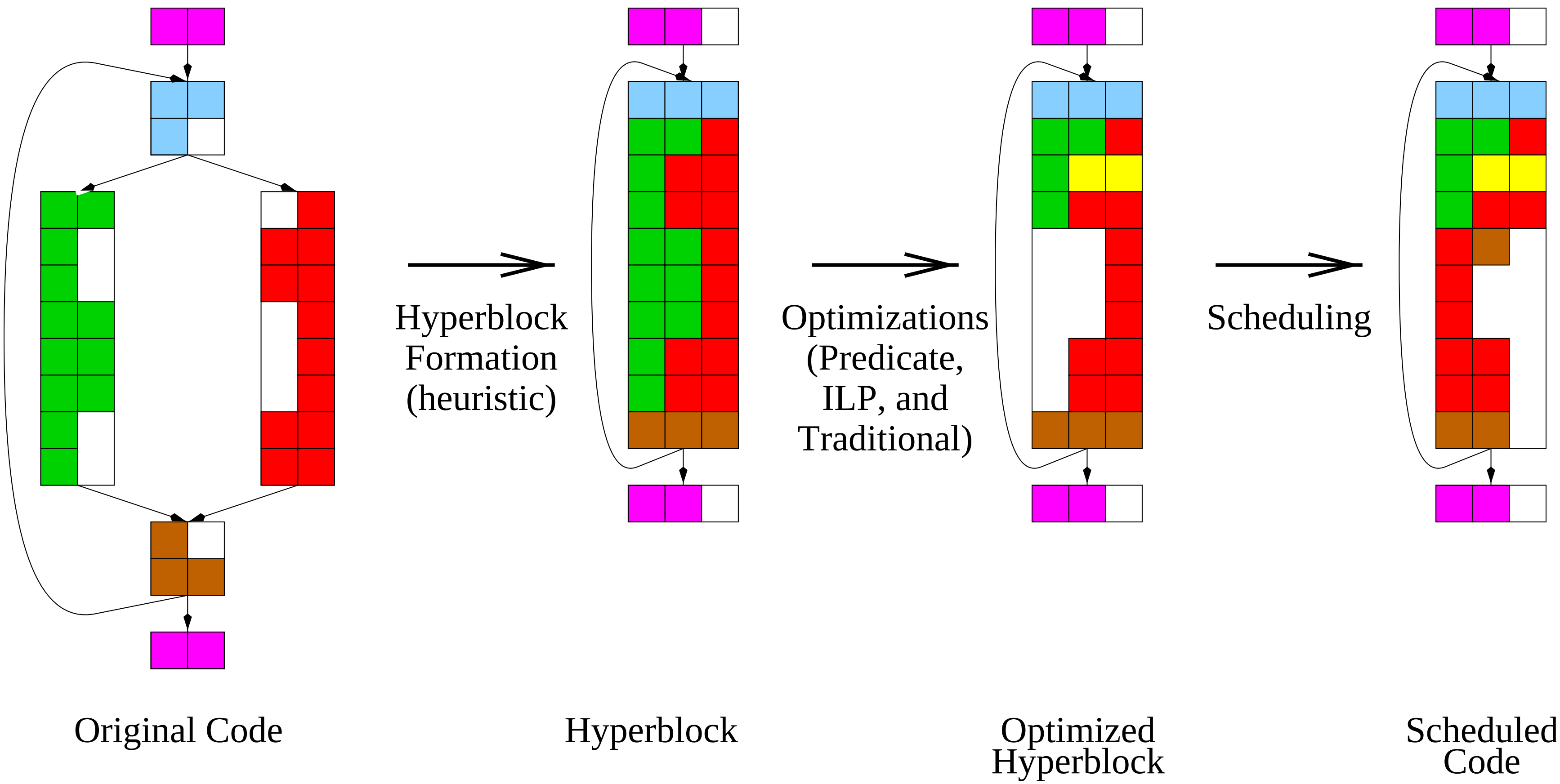


Problems with Hyperblock Compilation Framework

- 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.

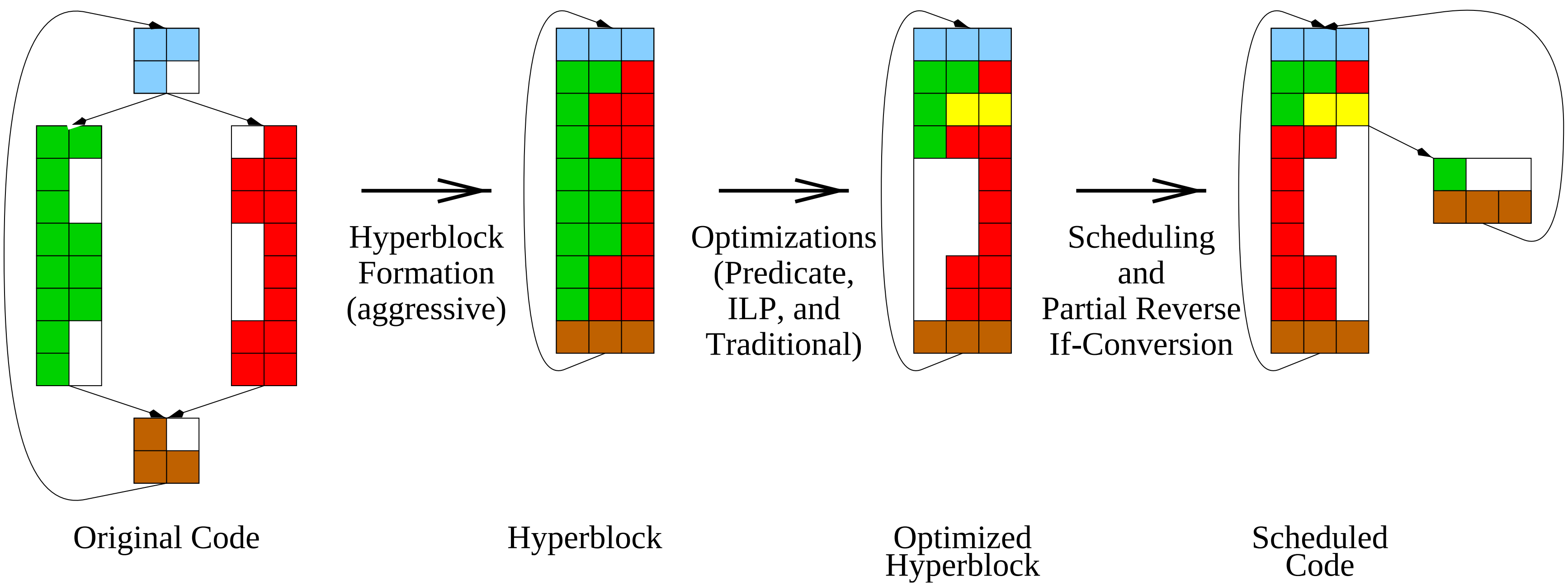
Phase Ordering - The Optimization Problem

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



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:

<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>

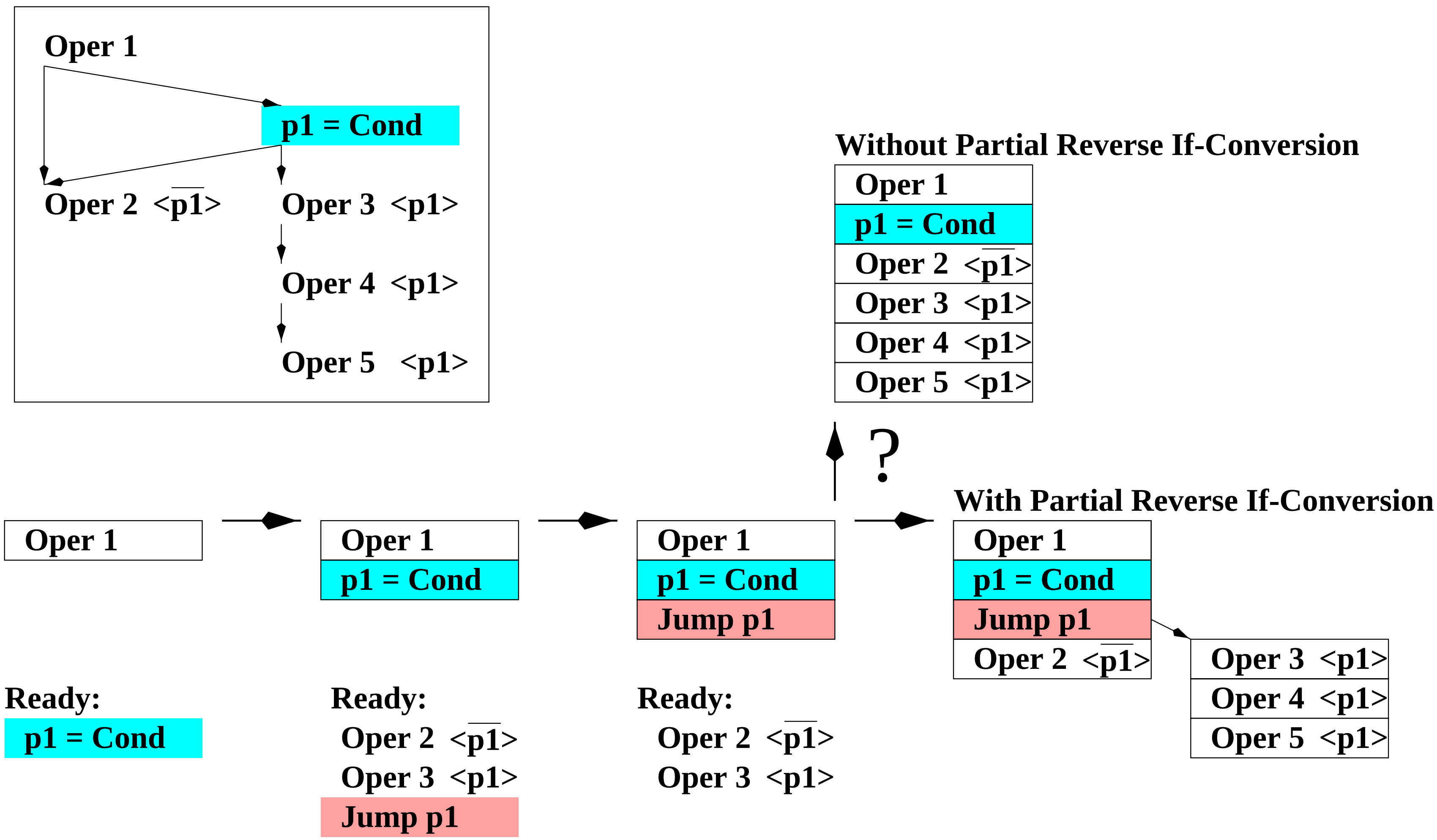
AFTER

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

Jump target:

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

Partial Reverse If-Conversion Algorithm

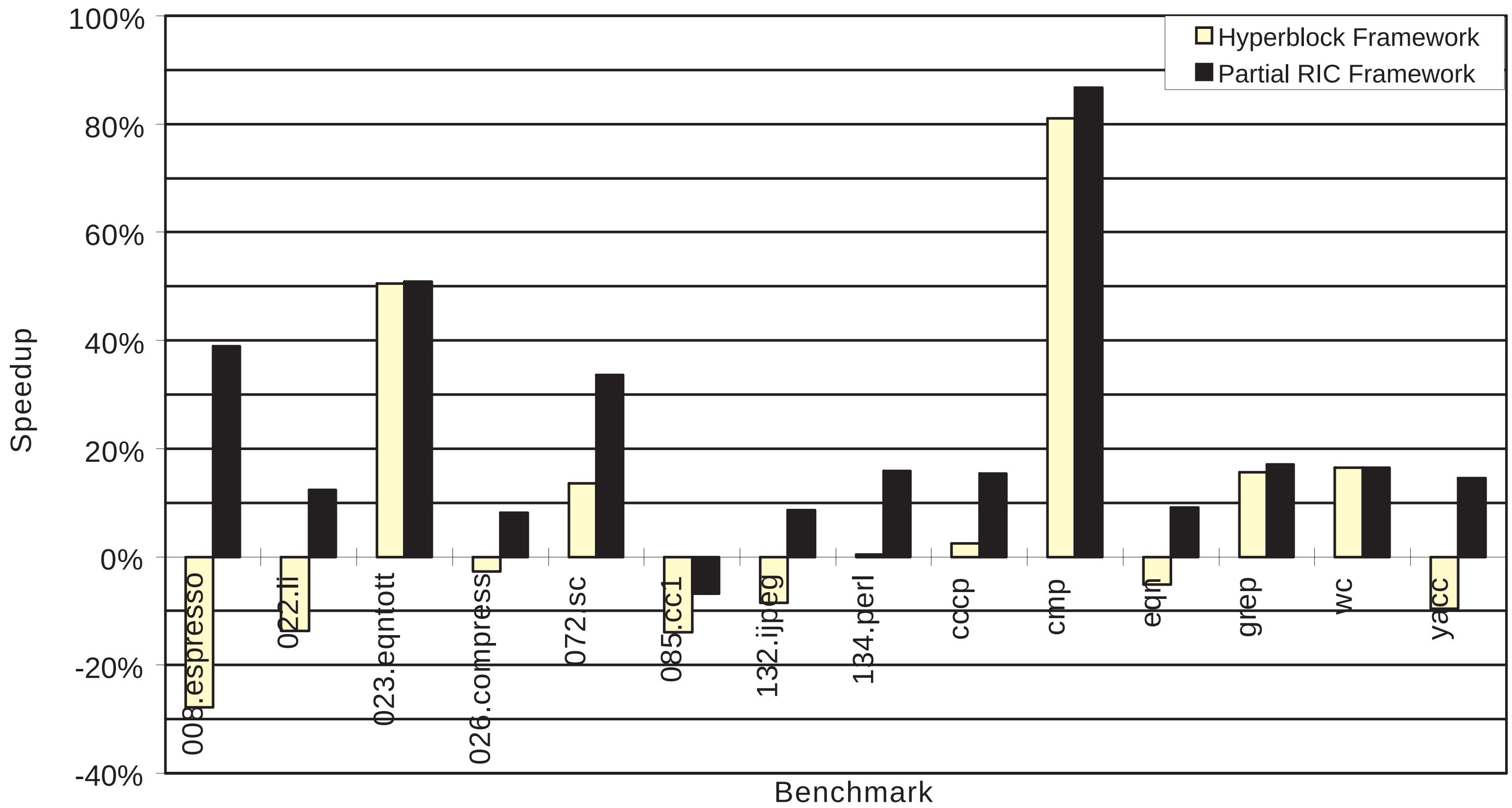


Code Example

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

David I. August
IMPACT Research Group

Performance Improvement



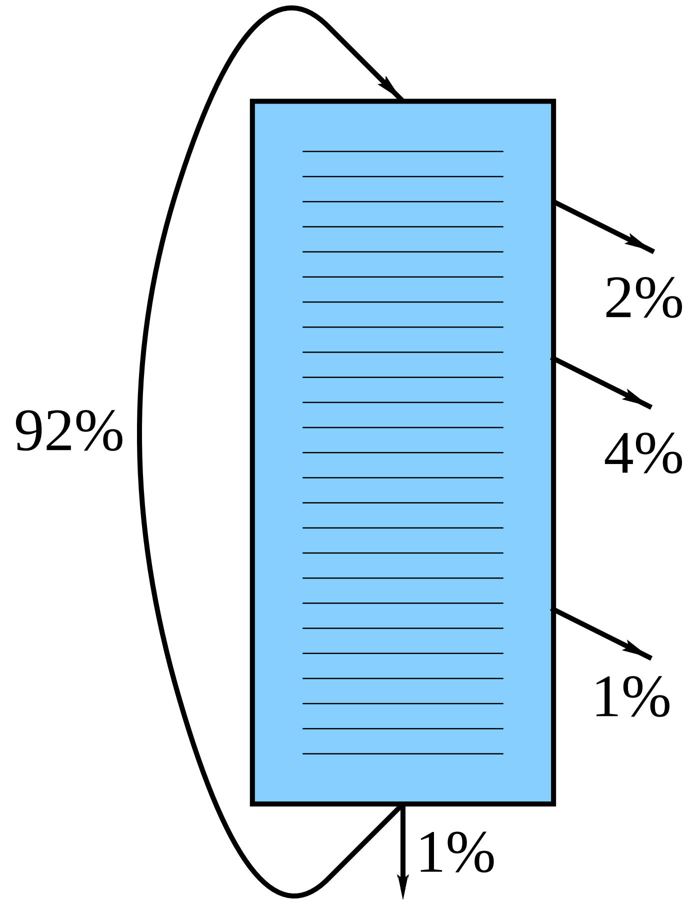
- No branch prediction penalty
- 4-issue: 1 branch, 2 integer, 2 memory, and 1 float

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

Fully Resolved Predicates: Motivation

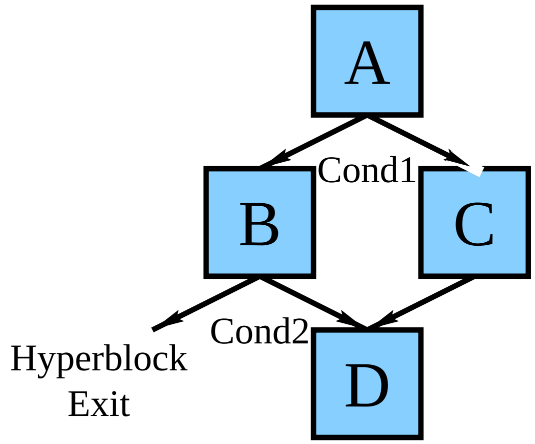
- Typical Hyperblocks and Superblocks have many infrequently taken exit branches.
- Infrequent exit branches
  - impede code motion
  - increase length of path to frequently taken branches
  - consume valuable branch resources
- Goal: Use predication to enhance performance in the presence of **easily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



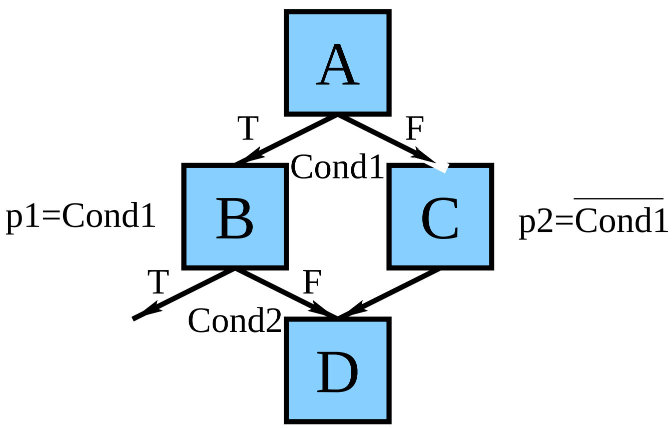
Fully Resolved Predicates: Concept

- 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.

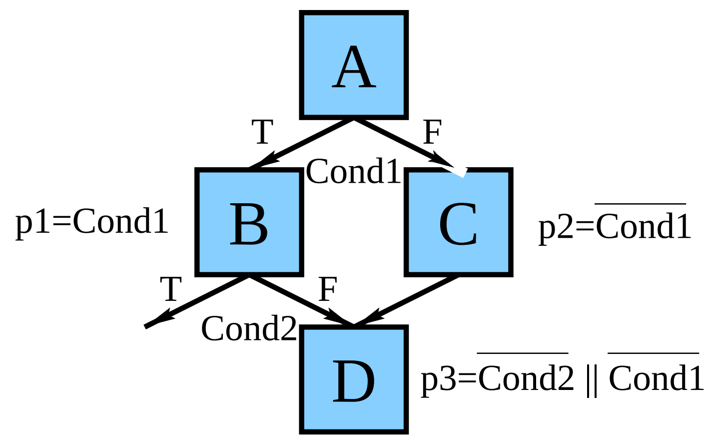
Fully Resolved Predicates: Computation



Partially Resolved Predicates



Fully Resolved Predicates



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:

```
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:

CB 6:

				Taken Frequency
1	r35 = MEM[r34]		branch r34 >= r37, CB 95	14
2	r34 = r34 + 1			
3			branch r35 == 10, CB 11	4035
4			branch r35 == 0, CB 11	0
5			branch r33 >= r57, CB 11	0
6	MEM[r33] = r35	r33 = r33 + 1	jump CB 6	101148

FRP Predicated Code Segment:

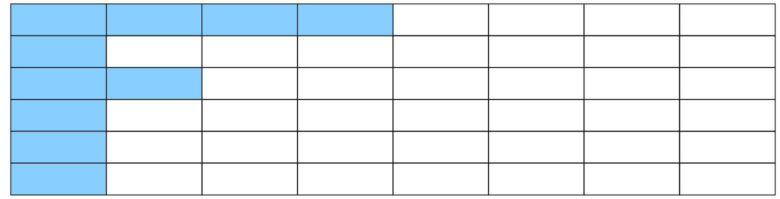
CB 6:

				Taken Frequency
1	r35 = MEM[r34]	$p0_{ut}$, $p1_{uf}$ = (r34 >= r37)		
2	r34 = r34 + 1 <p1>		jump CB 95 <p0>	14
3	$p2_{ut}$, $p3_{uf}$ = (r35 == 10) <p1>			
4	$p4_{ut}$, $p5_{uf}$ = (r35 == 0) <p3>		jump CB 11 <p2>	4035
5	$p6_{ut}$, $p7_{uf}$ = (r33 >= r57) <p5>		jump CB 11 <p4>	0
6	MEM[r33] = r35 <p7>	r33 = r33 + 1 <p7>	jump CB 6 <p7>	101148
7			jump CB 11 <p6>	0

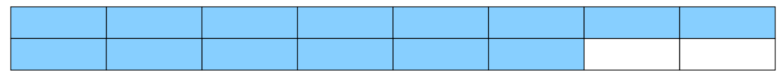
Path Height Reduction: Concept

- 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:
$T1 = A \circ B \quad T2 = C \circ D$
$E = T1 \circ T2$

Parallel Semantics:
$E \circ\!= A \circ B \quad E \circ\!= C \circ D$

"$\circ$" represents the universal associative operator.

FRP/PHR: Code Example

FRP Predicated Code Segment:

CB 6:

				Taken Frequency
1	r35 = MEM[r34]	$p0_{ut}$, $p1_{uf}$ = (r34 >= r37)		
2	r34 = r34 + 1 <p1>		jump CB 95 <p0>	14
3	$p2_{ut}$, $p3_{uf}$ = (r35 == 10) <p1>			
4	$p4_{ut}$, $p5_{uf}$ = (r35 == 0) <p3>		jump CB 11 <p2>	4035
5	$p6_{ut}$, $p7_{uf}$ = (r33 >= r57) <p5>		jump CB 11 <p4>	0
6	MEM[r33] = r35 <p7>	r33 = r33 + 1 <p7>	jump CB 6 <p7>	101148
7			jump CB 11 <p6>	0

FRP Predicated Code Segment with Height Reduction:

CB 6:

				Taken Frequency
1	r35 = MEM[r34]	$p0_{ut}$, $p1_{uf}$ = (r34 >= r37)	$p7_{af}$ = (r34 >= r37)	
2	r34 = r34 + 1 <p1>	$p7_{af}$ = (r33 >= r57)	jump CB 95 <p0>	14
3	$p2_{ut}$, $p3_{uf}$ = (r35 == 10) <p1>	$p7_{af}$ = (r35 == 10)	$p7_{af}$ = (r35 == 0)	
4	MEM[r33] = r35 <p7>	r33 = r33 + 1 <p7>	jump CB 6 <p7>	101148
5	$p4_{ut}$, $p5_{uf}$ = (r35 == 0) <p3>		jump CB 11 <p2>	4035
6	$p6_{ut}$ = (r33 >= r57) <p5>		jump CB 11 <p4>	0
7			jump CB 11 <p6>	0

FRP/PHR: *grep* Code Example Performance

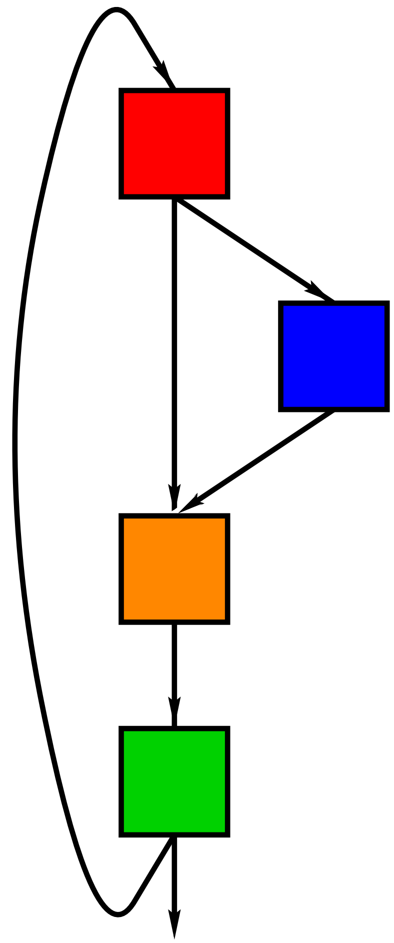
Cycle	Original HB	FRP Only	FRP w/ Height Red.
1	14		
2		14	14
3	4035		
4	0	4035	101148
5	0	0	4035
6	101148	101148	0
7		0	0
Cycles	619007	623056	424795
Speedup	1.00	0.99	1.46

- 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

CB 38:

1	r9 = r9 - r12	
2	($p1_{uf}$) = (r9 < 0)	
3		r9 = r9 + r13 (p1)
4	r10 = r9 << 2	
5	r114 = MEM[r10]	
6		
7	branch (r14 <> r8) CB 38	

Specialized Code Segment

CB 38:

1	r9 = r9 - r12	
2	($p1_{uf}$, $p2_{ut}$) = (r9 < 0)	
3	r110 = r9 << 2 (p2)	r9 = r9 + r13 (p1)
4	r114 = MEM[r110] (p2)	r10 = r9 << 2 (p1)
5		r14 = MEM[r10] (p1)
6	branch (r114 <> r8) CB 38 (p2)	
7		branch (r14 <> r8) CB 38 (p1)

Specialized Code Segment After Optimization

r1312 = r13 - r12

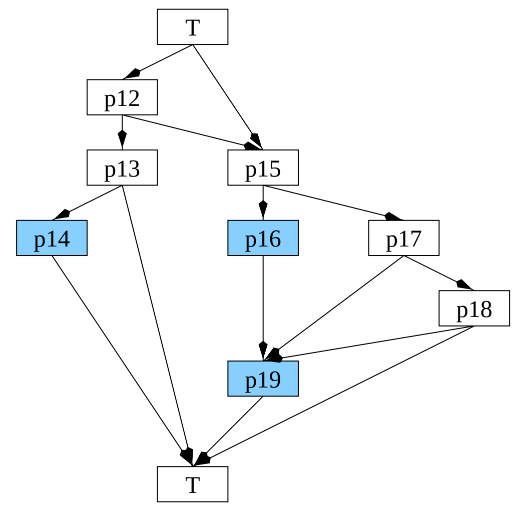
CB 38:

1		r9 = r9 - r12	r1009 = r9 + r1312
2	r110 = r9 << 2	($p1_{uf}$, $p2_{ut}$) = (r9 < 0)	r10 = r1009 << 2
3	r14 = MEM[r110] (p2)	r9 = r1009 (p1)	r14 = MEM[r10] (p1)
4			r10 = r9 << 2 (p1)
5	branch (r14 <> r8) CB 38		

Advanced Control Flow Transformation

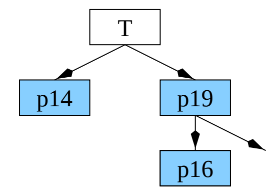
Original predicate definiton schedule

p15_of, p12_ut = (r4 > 32) <T>		
p15_of, p13_ut = (r4 < 127) <p12>		
p14_ut = (0 == r2) <p13>	p16_ut, p17_uf = (r4 == 10) <p15>	p19_ot = (r4 == 10) <p15>
p19_ot, p18_uf = (r4 == 32) <p17>		
p19_ot = (r4 == 9) <p18>		



c1 = (r4 > 32)
c2 = (r4 < 127)
c3 = (r2 == 0)
c4 = (r4 == 10)
c5 = (r4 == 32)
c6 = (r4 == 9)

	Original predicate expressions	Expressed in terms of conditions	Minimized
p12	c1	c1	
p13	p12 & c2	c1 & c2	
p14	p13 & c3	c1 & c2 & c3	c1 & c2 & c3
p15	!c1 | p12 & !c2	!c1 | c1 & !c2	
p16	p15 & c4	(!c1 | c1 & !c2) & c4	c4
p17	p15 & !c4	(!c1 | c1 & !c2) & !c4	
p18	p17 & !c5	((!c1 | c1 & !c2) & !c4) & !c5	
p19	p15 & c4 | p17 & c5 | p18 & c6	(!c1 | c1 & !c2) & c4 | ((!c1 | c1 & !c2) & !c4) & c5 | (((!c1 | c1 & !c2) & !c4) & !c5) & c6	c4 | c5 | c6



Predicate definition schedule after range analysis and and-type parallelization

p14_at = (r4 > 32) <T>	p14_at = (r4 < 127) <T>	p14_at = (r2 == 0) <T>	...

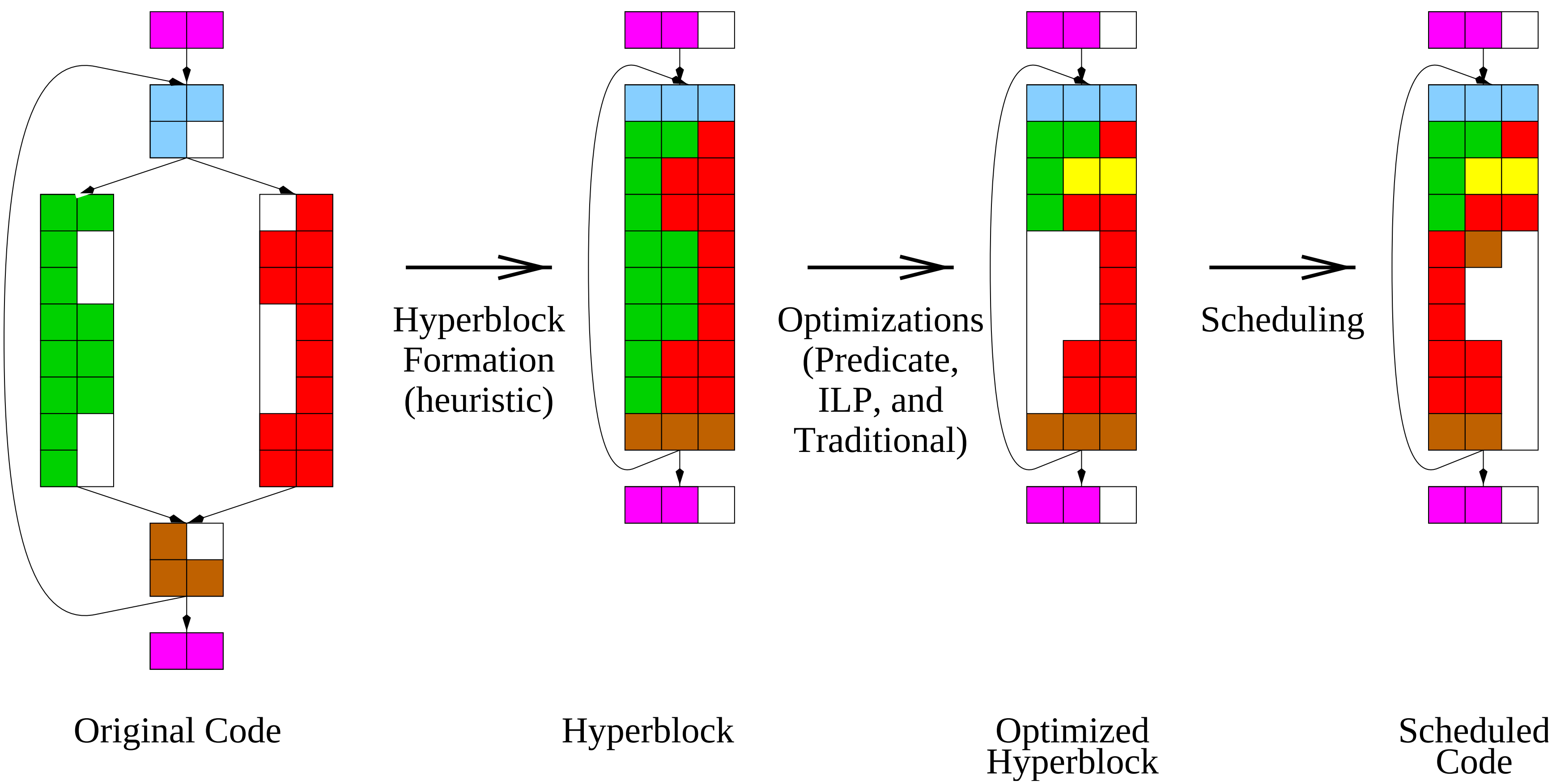
...	p19_ot, p16_ut = (r4 == 10) <T>	p19_ot = (r4 == 32) <T>	p19_ot = (r4 == 9) <T>

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.

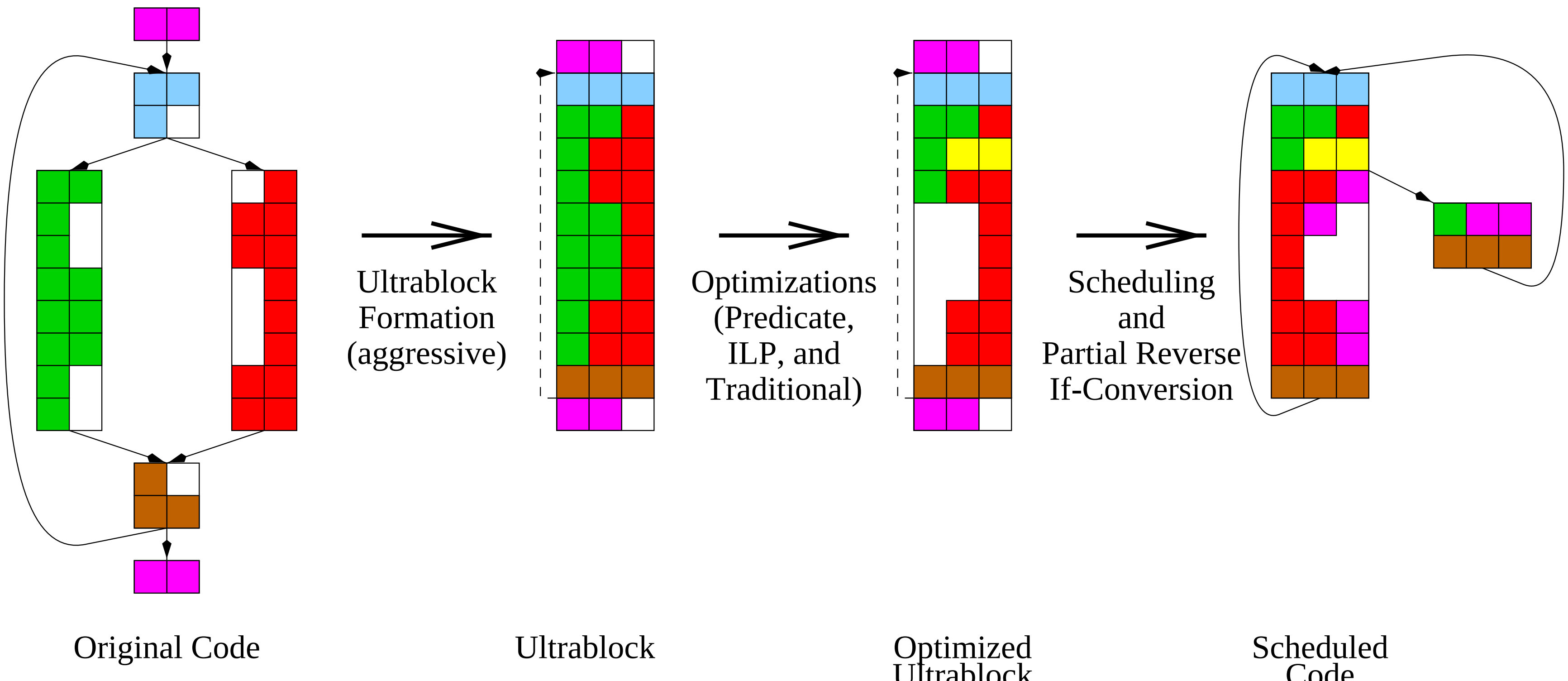
Compilation Block Scope - The Loop Boundary Problem

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



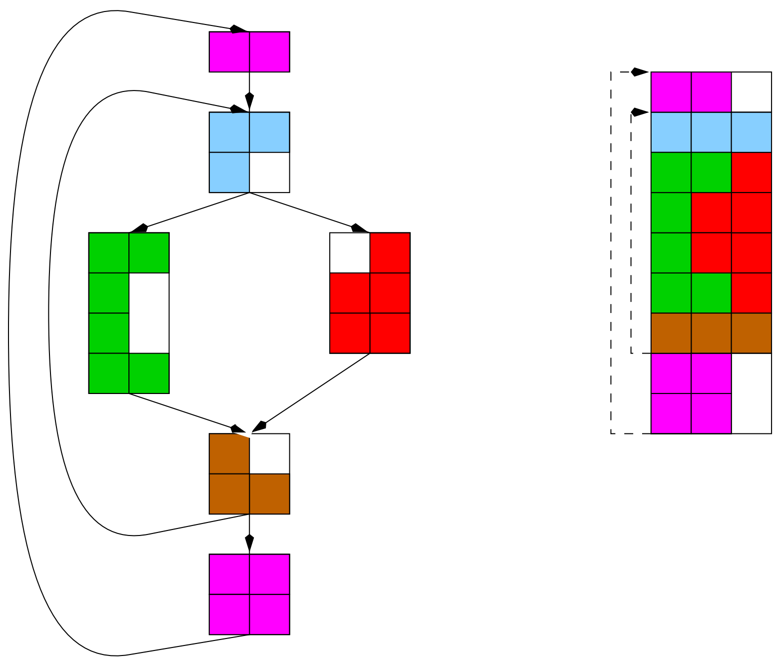
The Ultrablock Compilation Framework

- 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 reinstantiation of control flow



Intermediate Representation

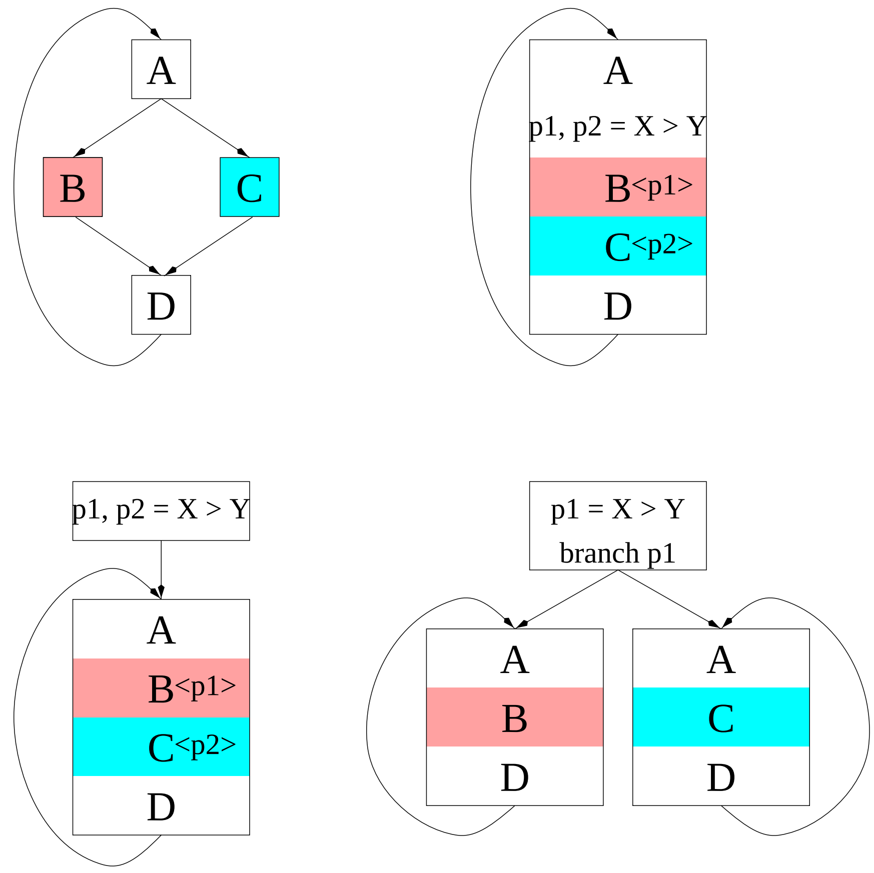
- IR needs to be extended to represent *ultrablocks* which can represent internal cycles to support compilation of general regions.
- Special purpose control flow and loop transformations can be replaced by data flow optimizations.
- A few techniques possible with current data flow optimizations are: loop versioning, loop fusion, if-then-else fusion, if-then-else interchange.



Ultrablock Example: Loop Versioning

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:

$p1_{ut} = cond1$ $p2_{ut} = cond2 < p1 >$	$p2_{ut}, p1_{ot} = cond1$ $p1_{ot} = cond2$	$p1_{ut}, p2_{at} = cond1$ $p2_{at} = cond2$

If $p1$ is a superset of $p2$:

$$\begin{aligned} r1 &= 10 \quad < p1 > \\ r2 &= r1 + 2 < p2 > \end{aligned} \Longrightarrow r2 = 12 < p2 >$$

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.

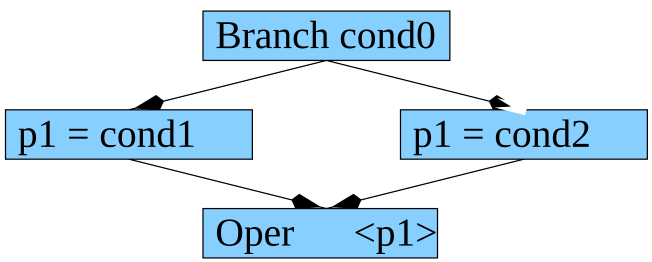
$$\begin{aligned} p2_{ut}, p1_{ot} &= cond1 \\ p1_{ot} &= cond2 \\ p3_{ut} &= cond3 \ < p2 > \end{aligned}$$

$p1$ 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.

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 = cond0 & cond1 | !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

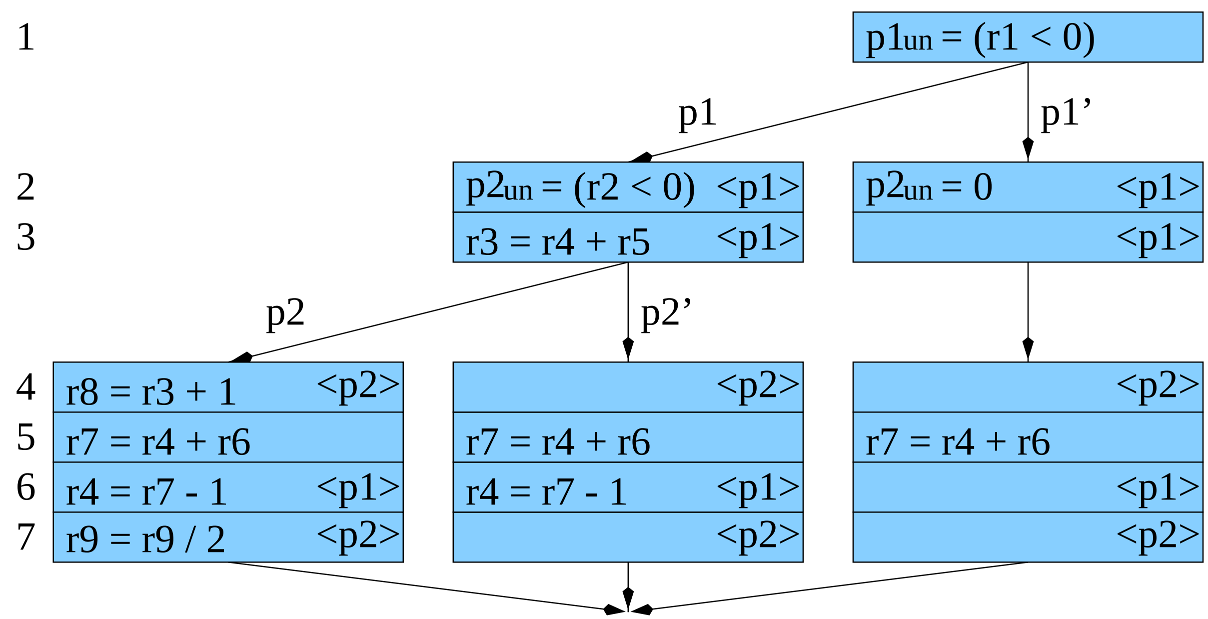
- Dataflow can be performed without regard to predicates; results are conservative.
- Conservative results make optimizations, scheduling, and register allocation less effective.
- Conservative dataflow
  - Only instructions on TRUE can KILL.
  - r3 is not killed by instruction 3 because it is predicated.
  - The live range of r3 = {1, 2, 3, 4}.
- Predicate-aware dataflow
  - Instructions on a predicate KILL on that predicate.
  - The live range of r3 = {3, 4}.

1	$p1_{un}$ = (r1 < 0)	
2	$p2_{un}$ = (r2 < 0)	<p1>
3	r3 = r4 + r5	<p1>
4	r8 = r3 + 1	<p2>
5	r7 = r4 + r6	
6	r4 = r7 - 1	<p1>
7	r9 = r9 / 2	<p2>

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.

1	$p1_{un}$ = (r1 < 0)	
2	$p2_{un}$ = (r2 < 0)	<p1>
3	r3 = r4 + r5	<p1>
4	r8 = r3 + 1	<p2>
5	r7 = r4 + r6	
6	r4 = r7 - 1	<p1>
7	r9 = r9 / 2	<p2>



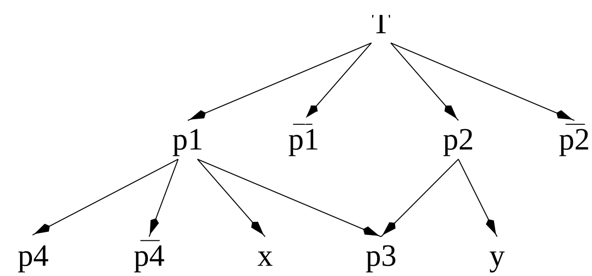
Dataflow Analysis Path Explosion Problem

- 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 p1, p2, and p3 are independent we have $2^3 = 8$ paths.

```
p1_un = (r1 < 0)
p2_un = (r2 < 0)
p3_un = (r3 < 0)
r5 = X            <p1>
r6 = Y            <p2>
Z = r5            <p3>
```

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.



SSA pred. def.	Resulting disjunctive expressions
$p_{i,j}$ =ut C $\langle p_g \rangle$	$p_{i,j} = p_g C$
$p_{i,j}$ =uf C $\langle p_g \rangle$	$p_{i,j} = p_g C'$
$p_{i,j} = [p_{i,j-1}]$ ot C $\langle p_g \rangle$	$p_{i,j} = p'_g p_{i,j-1} \vee p_g p_{i,j-1} C' \vee p_g C$
$p_{i,j} = [p_{i,j-1}]$ of C $\langle p_g \rangle$	$p_{i,j} = p'_g p_{i,j-1} \vee p_g p_{i,j-1} C' \vee p_g C$
$p_{i,j} = [p_{i,j-1}]$ ct C $\langle p_g \rangle$	$p_{i,j} = p'_g p_{i,j-1} \vee p_g C$
$p_{i,j} = [p_{i,j-1}]$ cf C $\langle p_g \rangle$	$p_{i,j} = p'_g p_{i,j-1} \vee p_g C'$
$p_{i,j} = [p_{i,j-1}]$ at C $\langle p_g \rangle$	$p_{i,j} = p'_g p_{i,j-1} \vee p_g p_{i,j-1} C$
$p_{i,j} = [p_{i,j-1}]$ af C $\langle p_g \rangle$	$p_{i,j} = p'_g p_{i,j-1} \vee p_g p_{i,j-1} C'$

A Global Predication Compilation Framework

David I. August

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