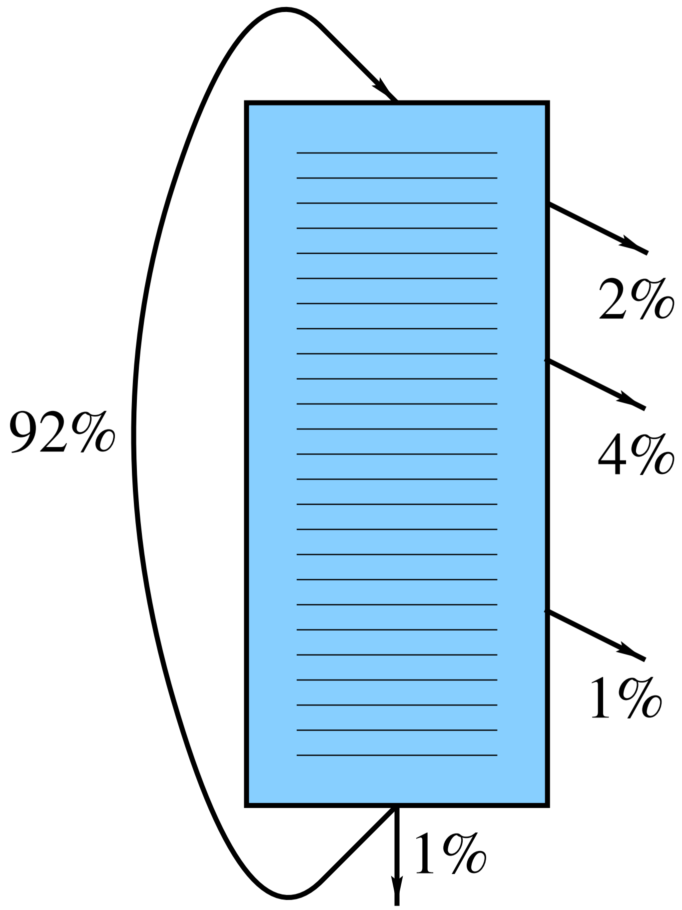
Predication

COS 597d

Fully Resolved Predicates: Motivation

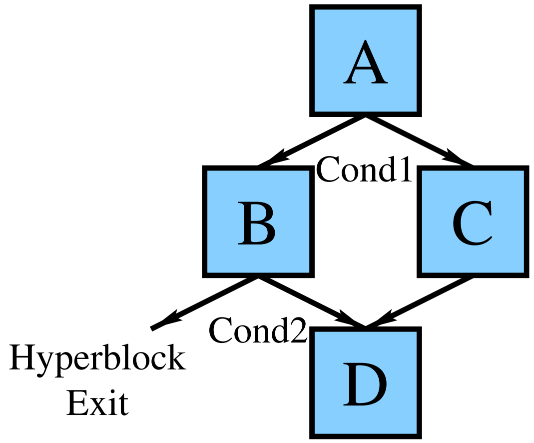
- Typical Hyperblocks and Superblocks have many infrequently taken exit branches.
- Infrequent exit branches
  - impede code motion
  - increase length of path to frequently taken branches
  - consume valuable branch resources
- Goal: Use predication to enhance performance in the presence of **easily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



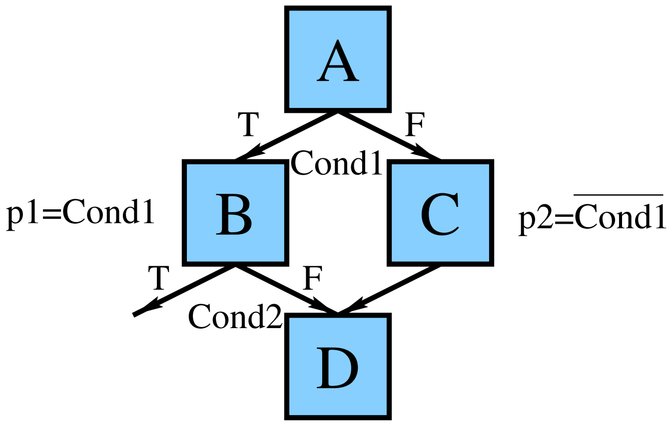
Fully Resolved Predicates: Concept

- 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.

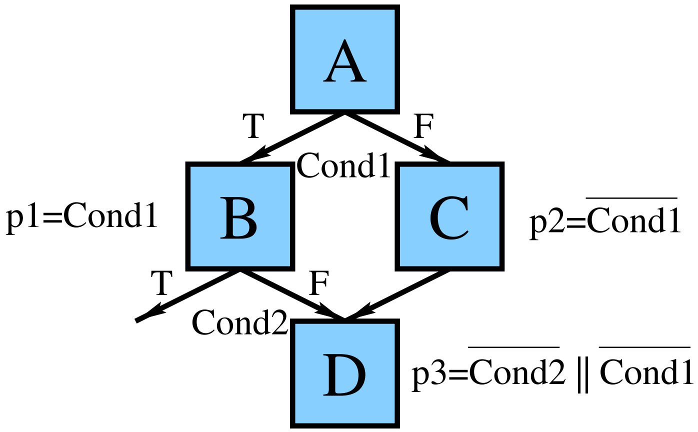
Fully Resolved Predicates: Computation



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:

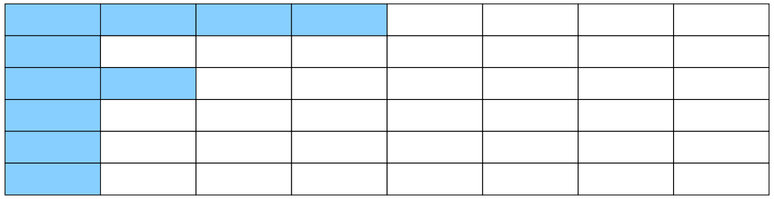
CB 6:

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

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:

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

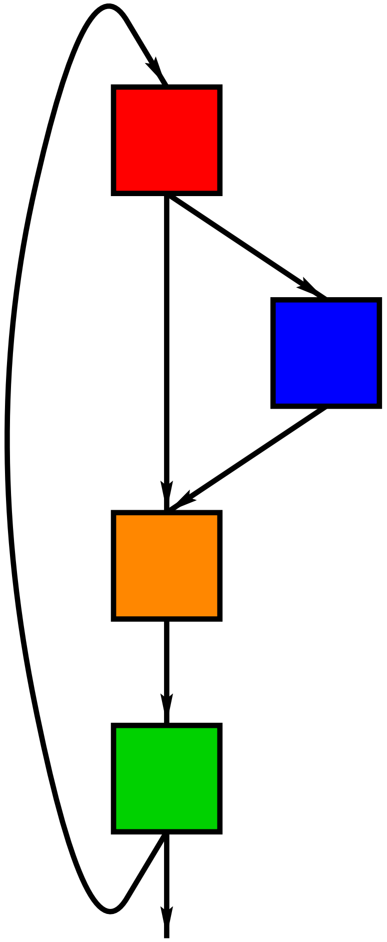
Cycle	Original HB	FRP Only	FRP w/ Height Red.
1	14		
2		14	14
3	4035		
4	0	4035	101148
5	0	0	4035
6	101148	101148	0
7		0	0
Cycles	619007	623056	424795
Speedup	1.00	0.99	1.46

- 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:

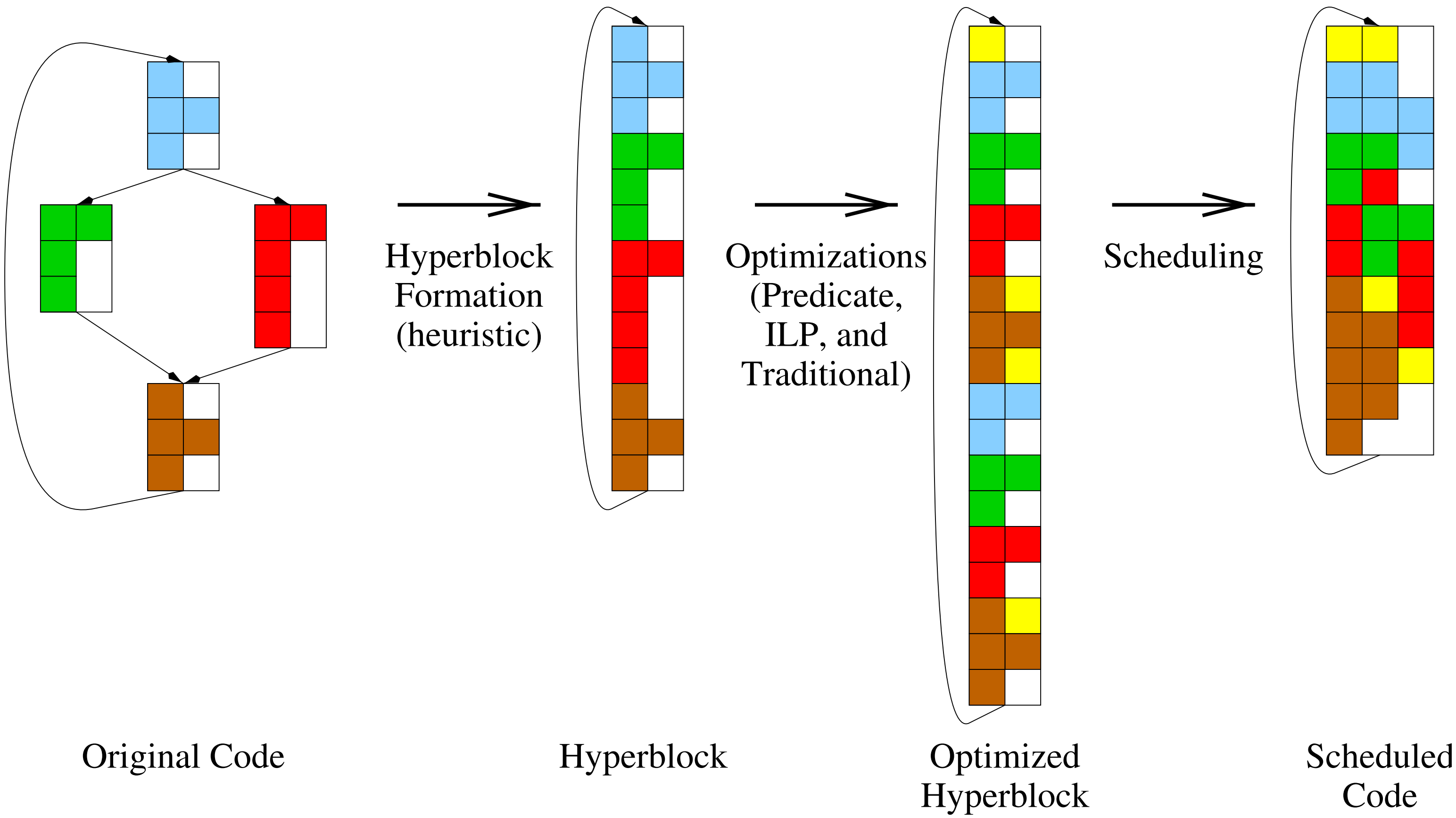
1		r9 = r9 - r12	r1009 = r9 + r1312
2	r110 = r9 << 2	($p1_{uf}$, $p2_{ut}$) = (r9 < 0)	r10 = r1009 << 2
3	r14 = MEM[r110] (p2)	r9 = r1009 (p1)	r14 = MEM[r10] (p1)
4			r10 = r9 << 2 (p1)
5	branch (r14 <> r8) CB 38		

Goal of This Work

- GOAL: To propose a flexible *predication framework* which takes full advantage of predication.
- A *Predication Framework* is a compilation framework which makes best use of predication.
  - Foundation for optimizations on predicated code
  - Defined by when and how predication is introduced or adjusted.
- When to Introduce Predication:
  - Early:
    - Allows aggressive optimization in the predicated representation
    - Hard to determine exactly how much and what to if-convert
  - Late:
    - Easy to determine exactly how much and what to if-convert
    - Predicate specific optimizations cannot be applied before if-conversion

Original IMPACT Predication Framework

- Early heuristic hyperblock formation estimates final code characteristics:

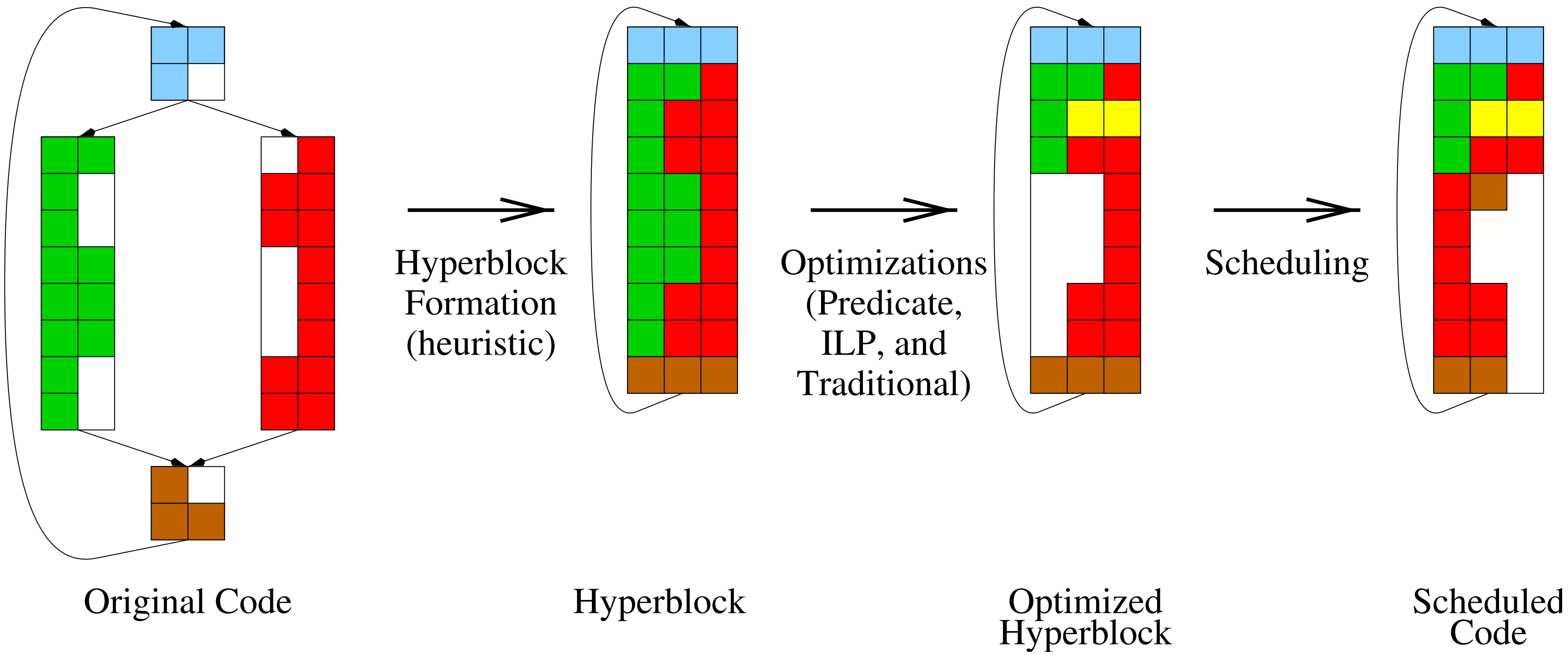


Problems with Original Predication Framework

- 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.

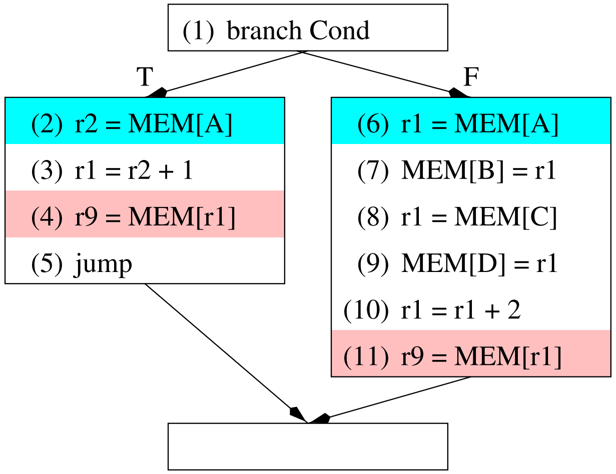
The Optimization Problem

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



Optimization

- Characteristics of formed hyperblock changes with further optimization:
  - Predicate specific optimizations can only be done after if-conversion
  - Optimizations may turn a good hyperblock into a bad one, or a bad hyperblock into a good one.
- Example:
  - Machine model: 3-issue.
  - Dependence height: $max\{3\ cycles, 6\ cycles\} = 6\ cycles.$
  - Issue resource height: $4\ inst + 6\ inst = 10\ inst \Rightarrow 4\ cycles.$



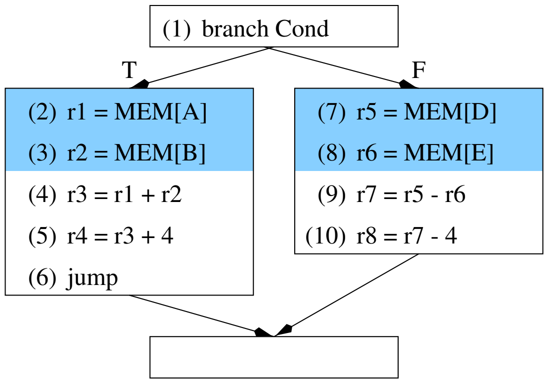
(a)

1	(1) p1, $\overline{p2}$ = Cond	(2,6) r2 = MEM[A]	(8) r3 = MEM[C]
2	(3) r1 = r2 + 1 <p1>	(7) MEM[B] = r2 <p2>	(10) r1 = r3 + 2 <p2>
3	(4,11) r9 = MEM[r1]	(9) MEM[D] = r3 <p2>	

(b)

Resource and Dependence Height

- Characteristics of formed hyperblock are dependent on its constituent paths:
  - Dependence height is at least the maximum of dependence heights.
  - Resource consumption is the sum of all resources consumed.
  - Resource height and dependence height interfere in a complex manner.
- Example:
  - Machine model: 3-issue, 1 memory unit.
  - Dependence height: $max\{3\ cycles, 3\ cycles\} = 3\ cycles$.
  - Memory resource height: $2\ mem + 2\ mem = 4\ mem \Rightarrow 4\ cycles$.
  - Issue resource height: $5\ inst + 4\ inst = 9\ inst \Rightarrow 3\ cycles$.



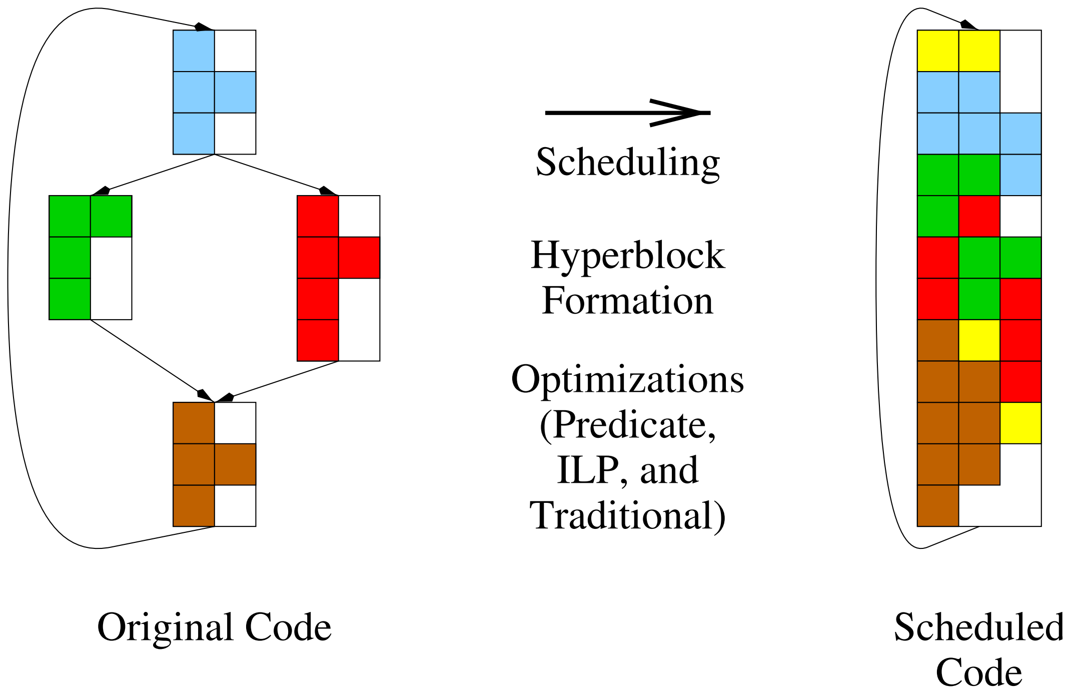
(a)

1	(1) p1, $\overline{p2}$ = Cond	(2) r1 = MEM[A]	
2	(3) r2 = MEM[B] <p1>		
3	(4) r3 = r1 + r2 <p1>	(7) r5 = MEM[D] <p2>	
4	(5) r4 = r3 + 4 <p1>	(8) r6 = MEM[E] <p2>	
5	(9) r7 = r5 - r6 <p2>		
6	(10) r8 = r7 - 4 <p2>		

(b)

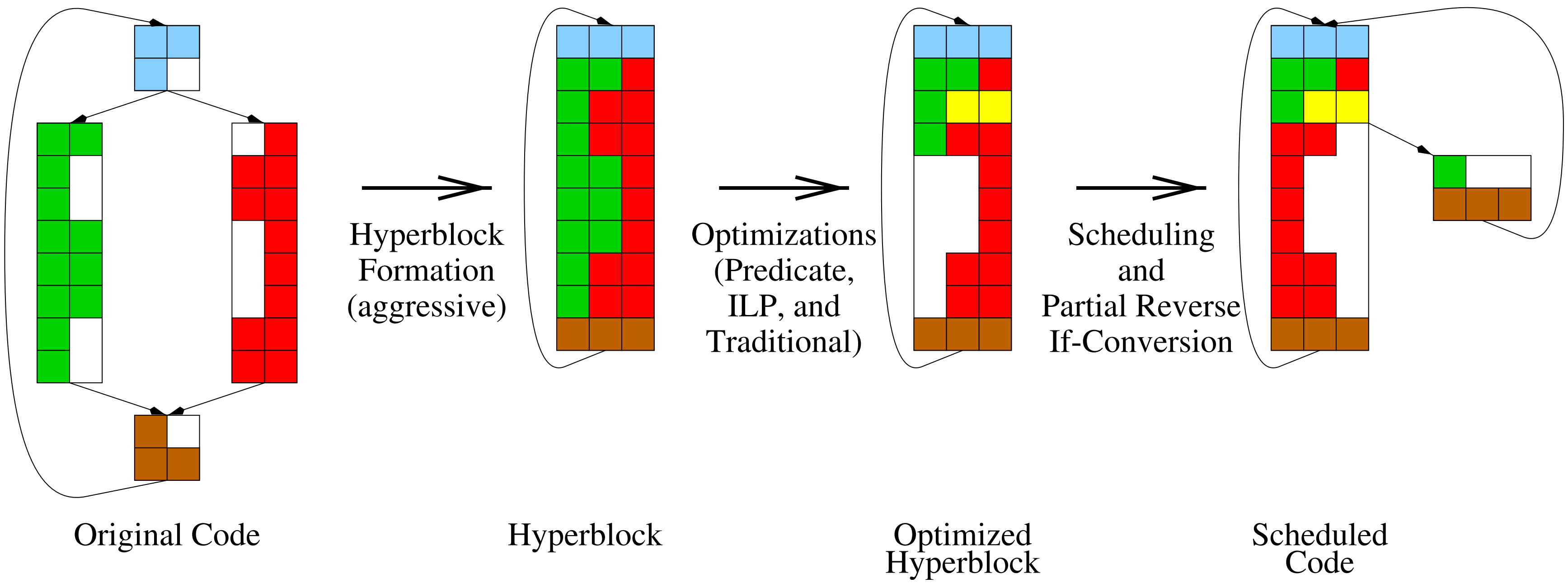
The If-Conversion During Scheduling Framework

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



The Proposed Predication Framework

- The Partial Reverse If-Conversion Framework
  - Early aggressive formation
  - Partial Reverse If-Conversion for scheduler adjustment of predication



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

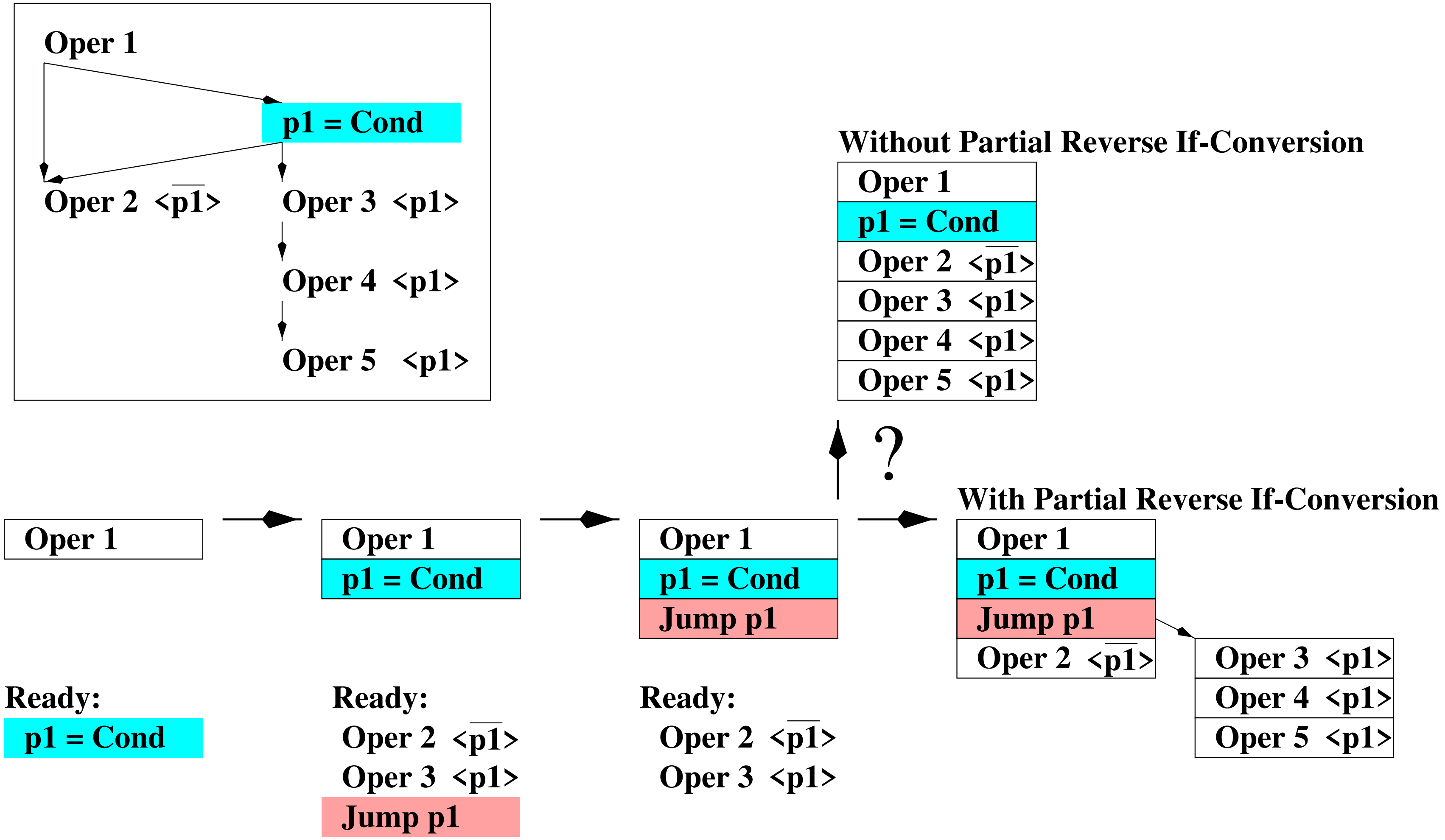
1	T2 =	<TRUE>
2	T1 = T2	<p1>
3		<p2>
p2		
4	r1 =	<p2>
5	T3 = T1	<TRUE>
6	r2 =	<TRUE>
7	r3 = T3	<p1>

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

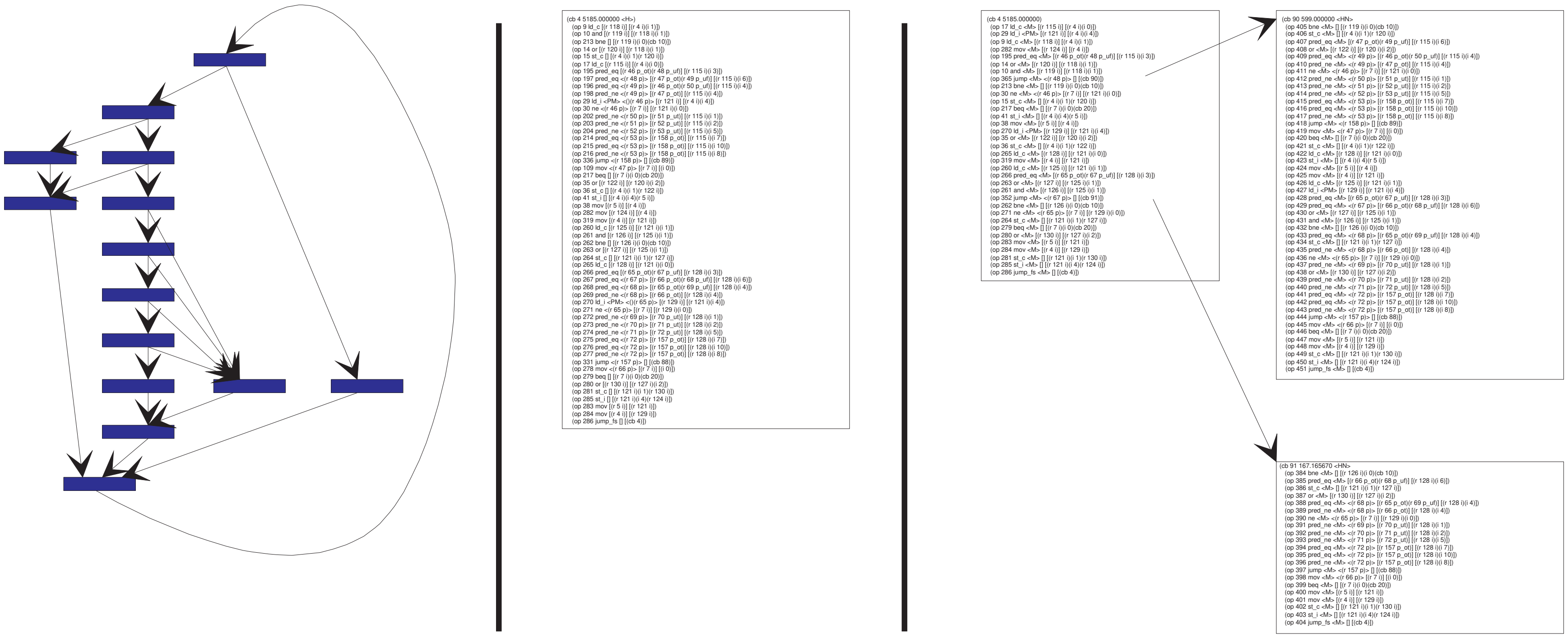


Partial Reverse If-Conversion Algorithm

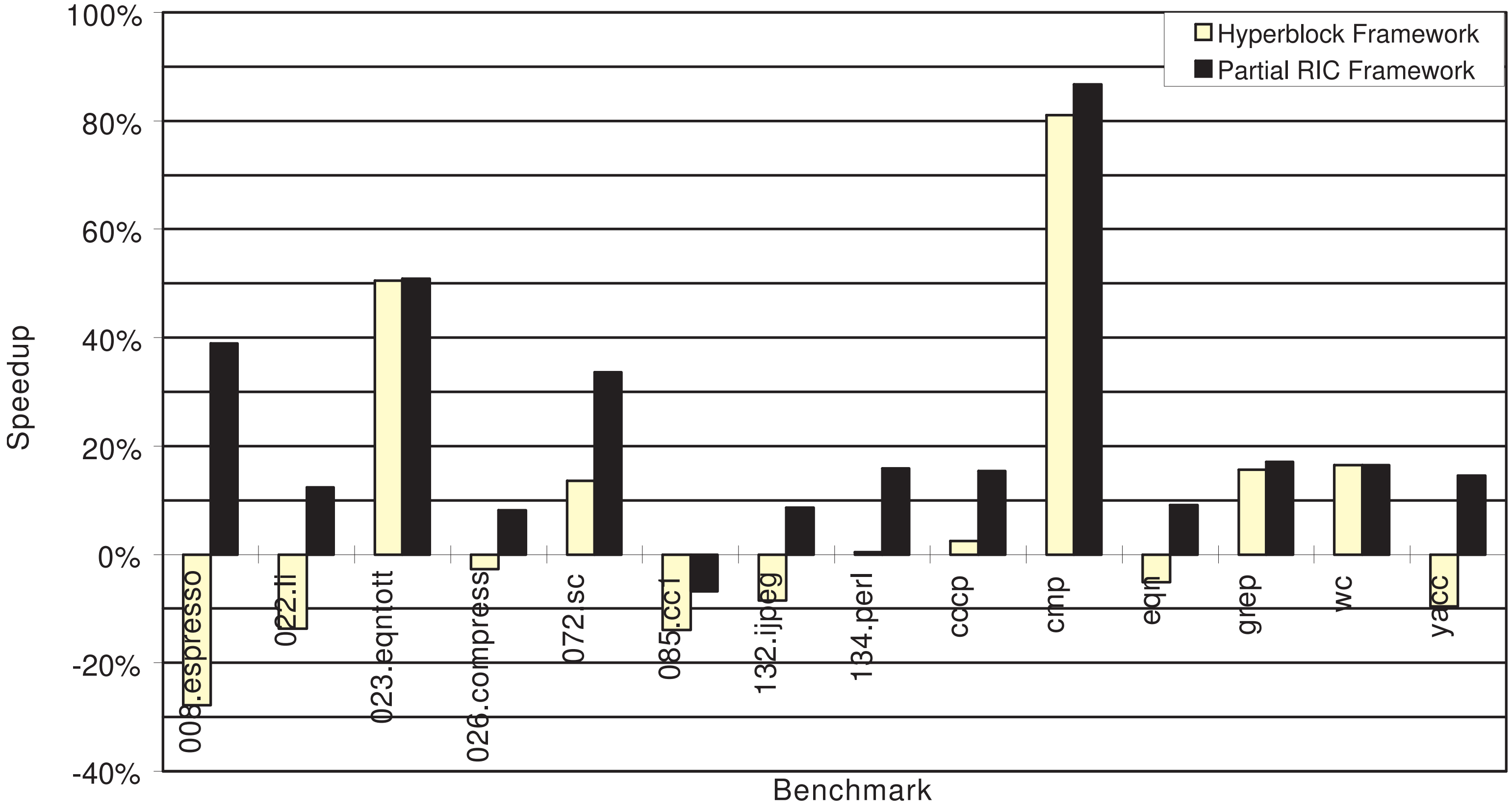
```
WHILE num_ops_unsched != 0 DO
    // Handle reverse if-converting branches first
    FOREACH ric_op IN ric_queue DO
        IF Schedule_Op(ric_op, cycle) THEN
            Compute location and schedule for each unscheduled op;
            IF (sched_no_ric > ric_cycles) THEN
                Place all ops in their no_ric schedule location;
            ELSE
                Unschedule_OP(ric_op);
            Remove ric_op from ric_queue;
    // Then handle regular operations
    FOREACH regular_op IN ready_priority_queue DO
        IF Schedule_Op(regular_op, cycle) THEN
            Remove regular_op from ready_priority_queue;
            IF Is_Predicate_Define(regular_op) THEN
                Add reverse if-converting branch to ric_queue;
    cycle = cycle + 1;
```

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



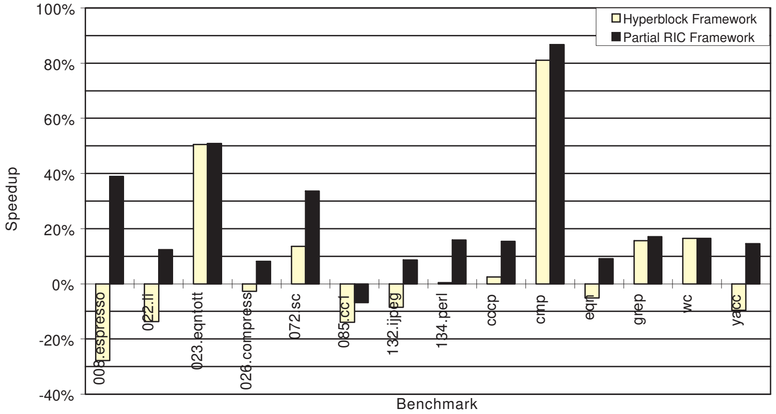
Performance Improvement



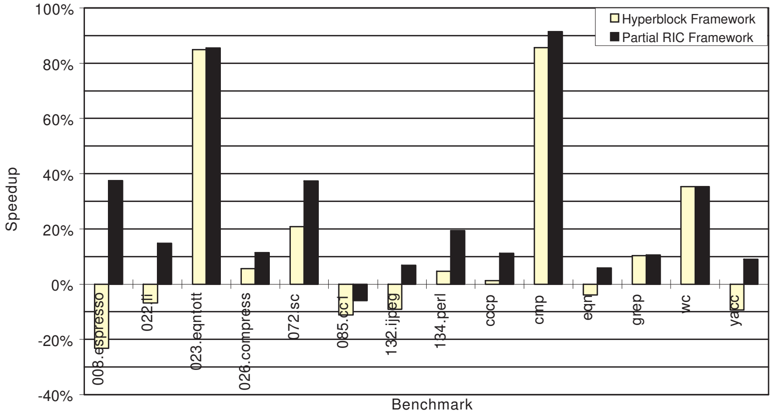
- No branch prediction penalty
- 4-issue: 1 branch, 2 integer, 2 memory, and 1 float

Experimental Results

No Branch Prediction Penalty:



4 Cycle Branch Predication Penalty:

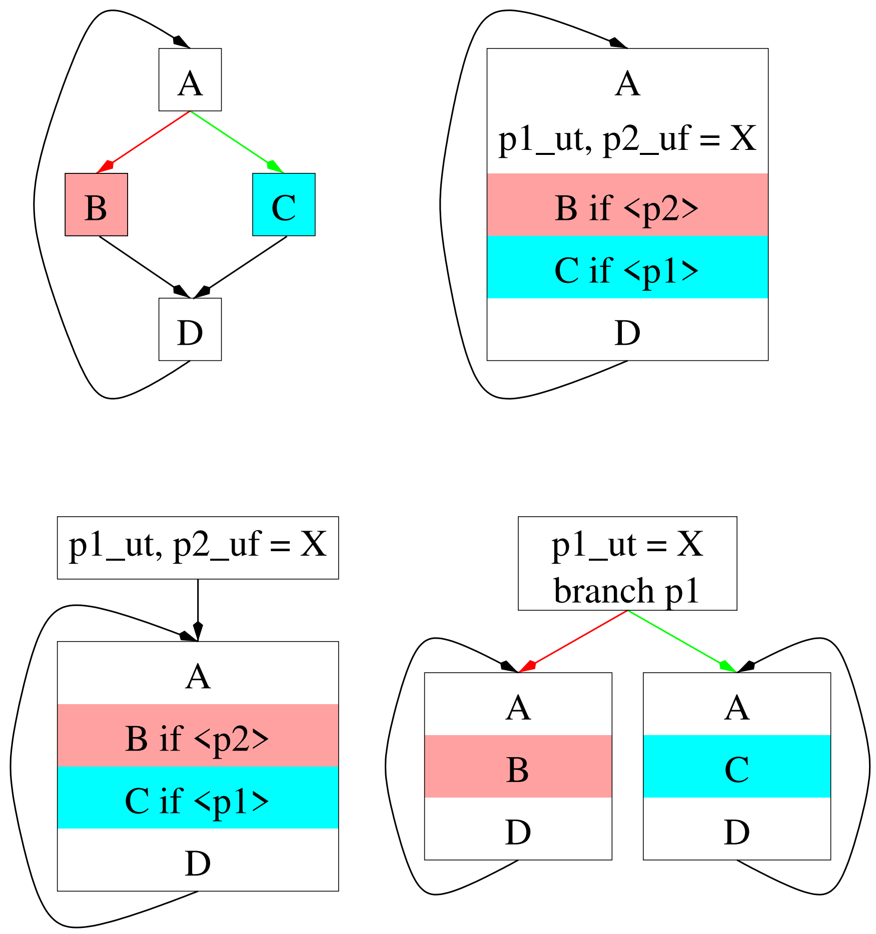


Application Statistics

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

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.



Conclusion

- Effective framework for predicated compilation:
  - Early aggressive if-conversion
  - Partial reverse if-conversion during scheduling
- Results show improvement over original framework:
  - Original framework too aggressive → improve performance by removing paths
  - Original framework too conservative → improve performance by including partial paths
  - Provides performance stability for changing processor resources
- Future work:
  - Make reverse if-conversion algorithm more elegant
  - Use a path-based scheduling approach