Predication

COS 597d

Problem Statement

- VLIW and superscalar processors
  - Identify and execute independent instructions in parallel
  - Performance dependent on amount of ILP extracted from the application
- Branches limit the available ILP by imposing control flow dependences
- High frequency of branches in non-numerical applications
- ILP within basic blocks very small
  - 2-5 available instructions
  - Instructions have data dependences between one another

Dynamic Instruction Mix

Benchmark	Load	Store	IALU	FALU	Branch
008.espresso	0.228	0.027	0.520	0.000	0.225
022.li	0.323	0.137	0.226	0.000	0.313
023.eqntott	0.223	0.004	0.402	0.000	0.371
026.compress	0.201	0.098	0.450	0.000	0.252
072.sc	0.279	0.032	0.349	0.008	0.332
085.cc1	0.210	0.117	0.349	0.000	0.323
cccp	0.189	0.079	0.376	0.000	0.356
cmp	0.265	0.027	0.118	0.000	0.589
eqn	0.197	0.119	0.319	0.000	0.365
grep	0.202	0.092	0.146	0.000	0.560
lex	0.253	0.179	0.119	0.000	0.450
qsort	0.204	0.186	0.379	0.000	0.230
tbl	0.311	0.033	0.279	0.000	0.377
wc	0.123	0.131	0.331	0.000	0.415
yacc	0.240	0.035	0.349	0.000	0.376
Avg.	0.230	0.086	0.471	0.000	0.369

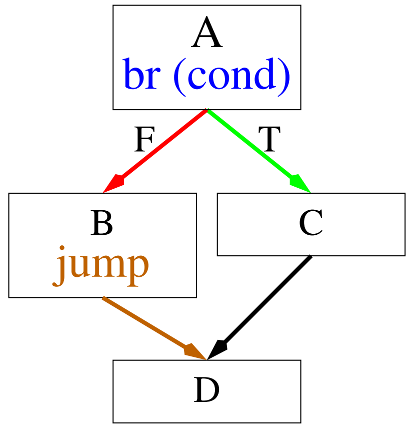
Like most user applications, these are integer, non-numeric programs.

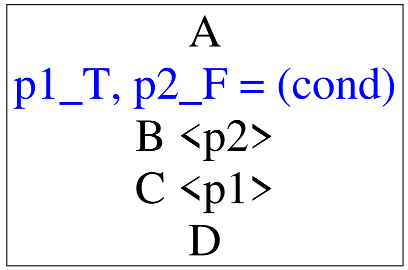
Problem Statement

- Look beyond basic block boundaries using branch prediction
  - Effective when continually predict correctly
  - Large penalty for mispredictions
  - Limited to extracting ILP along 1 path of control
- Limited branch handling capability per cycle puts ceiling on ILP
  - 1 branch per cycle $\Rightarrow$ 2-4 max IPC
  - Multiflow Trace - executed 4 branches per cycle
  - But . . .
    - Pipelined implementation is complex
    - Gate delay constraints of future ILP processors
    - Multiple branch predictions per cycle vs sacrifice BP accuracy
  - Duplicating arithmetic function units much simpler

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



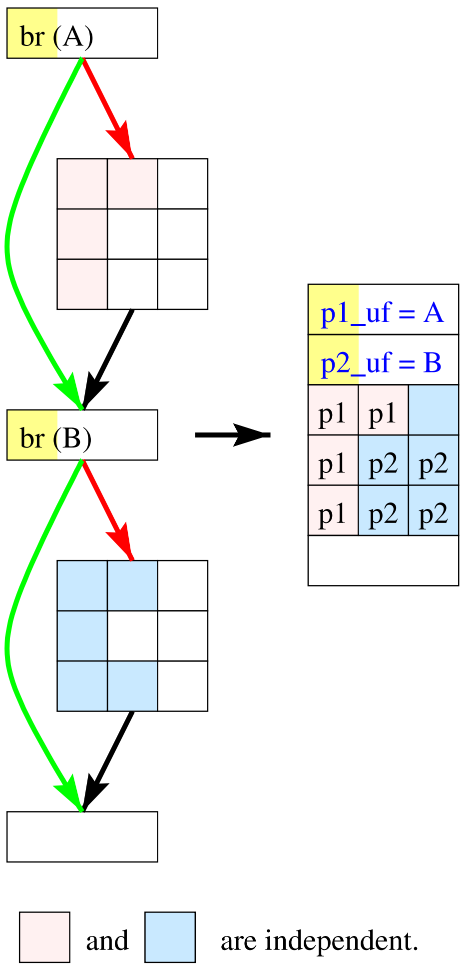


Predication Uses

- *Predicated Representation* - A program representation in which an instruction can be guarded by a Boolean source operand.
  - Larger code blocks enable an 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 and loop peeling.
  - Allows removal of branch mispredictions and resource consumption through elimination of branches.

Overlapping of Independent Paths

- ILP is increased when independent paths of computation can be executed simultaneously.
- Branching control flow does not allow this transformation to be performed cleanly.
- Predication allows independent paths to be overlapped efficiently.



Instruction Set Support for Predicated Execution

- Full Predication Support
  - Predicate definition instructions, full set of predicated instructions, separate register file.
  - Best performance gain
- Partial Predication Support
  - Some sub-set (or more limited variation) of full predication support
  - Brings some benefit of predication to existing ISA's
  - Modest performance gain
- Dynamic Predication Support
  - No change needed to existing ISA
  - Smallest performance gain

Full Predication Hardware Support...

Predicate Defining Instructions

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

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

Predicate Defining Instructions

$$P_{<type>} = (\,Comp\,)\,\langle P_g \rangle$$

		PlayDoh types						New types			
P_g	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.

Predicate Defining Instructions in IA-64

Predicate defines are called *compares* in IA-64.

$$(qp)\ \mathrm{cmp}.crel.ctype\ p_1, p_2 = r_2, r_3$$

IA-64 crels:

eq, ne, lt, le, gt, ge, ltu, leu, gtu

IA-64 ctypes:

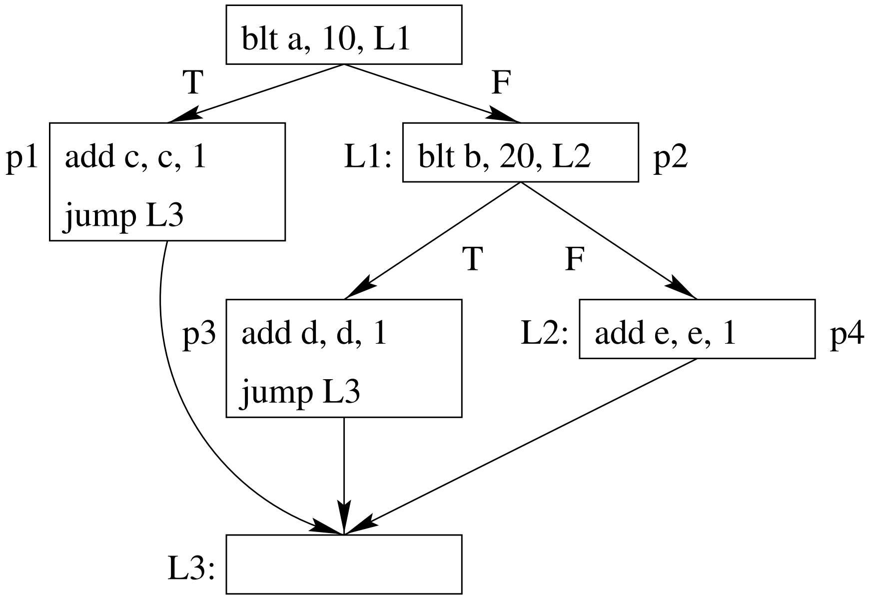
$ctype$	p_1	p_2
none	CT	CF
unc	UT	UF
or	OT	OT
and	AT	AT
or.andcm	OT	AF

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 $\langle p4 \rangle$



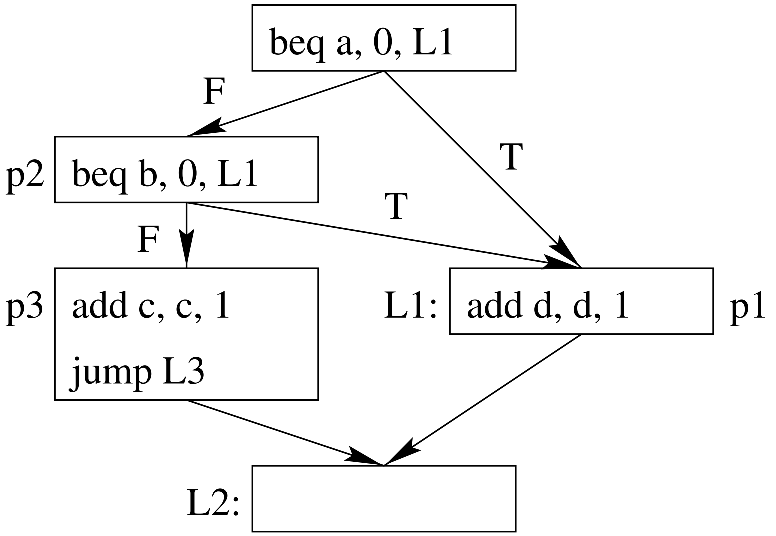
P_g	Comparison	P_d UT	P_d UF
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Wired-OR Predicate Define

Generate a predicate for a block which executes on the disjunction of multiple conditions.

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

$p1 = 0$
$p1_{OT}, p2_{UF} = (a == 0)$
$p1_{OT}, p3_{UF} = (b == 0)\ \langle p2 \rangle$
$c = c + 1\ \langle p3 \rangle$
$d = d + 1\ \langle p1 \rangle$



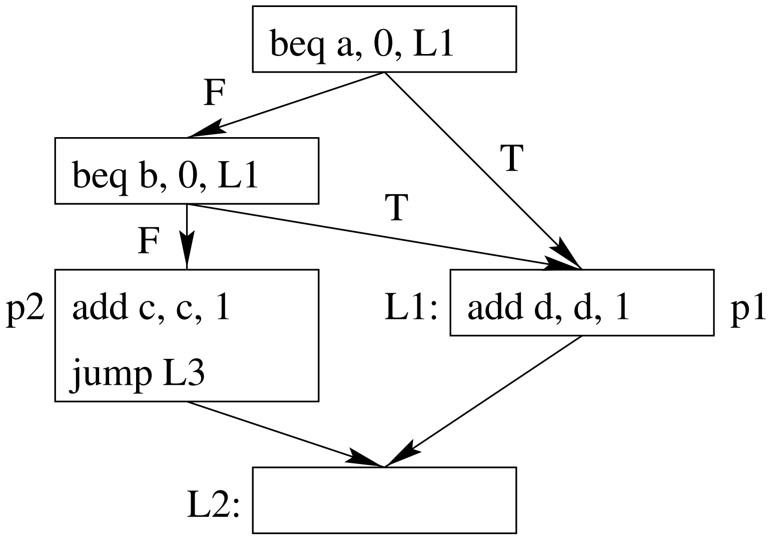
P_g	Comparison	P_d OT	P_d OF
0	0	–	–
0	1	–	–
1	0	–	1
1	1	1	–

Wired-AND Predicate Define

Generate a predicate for a block which executes on the conjunction of 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_d	
P_g	Comparison	AT	AF
0	0	-	-
0	1	-	-
1	0	0	-
1	1	-	0

Issue Semantics

- Conversion to NOP model
  - Instruction converted to NOP at issue if predicate is 0
  - Most efficient model, useful for in-order issue processors
- Data selection model
  - Facilitates dynamic register renaming and out-of-order issue
  - If (p==1) dest = op(src1, src2, ...)
  - Else dest = old_dest
  - Predicated instructions always produce a result

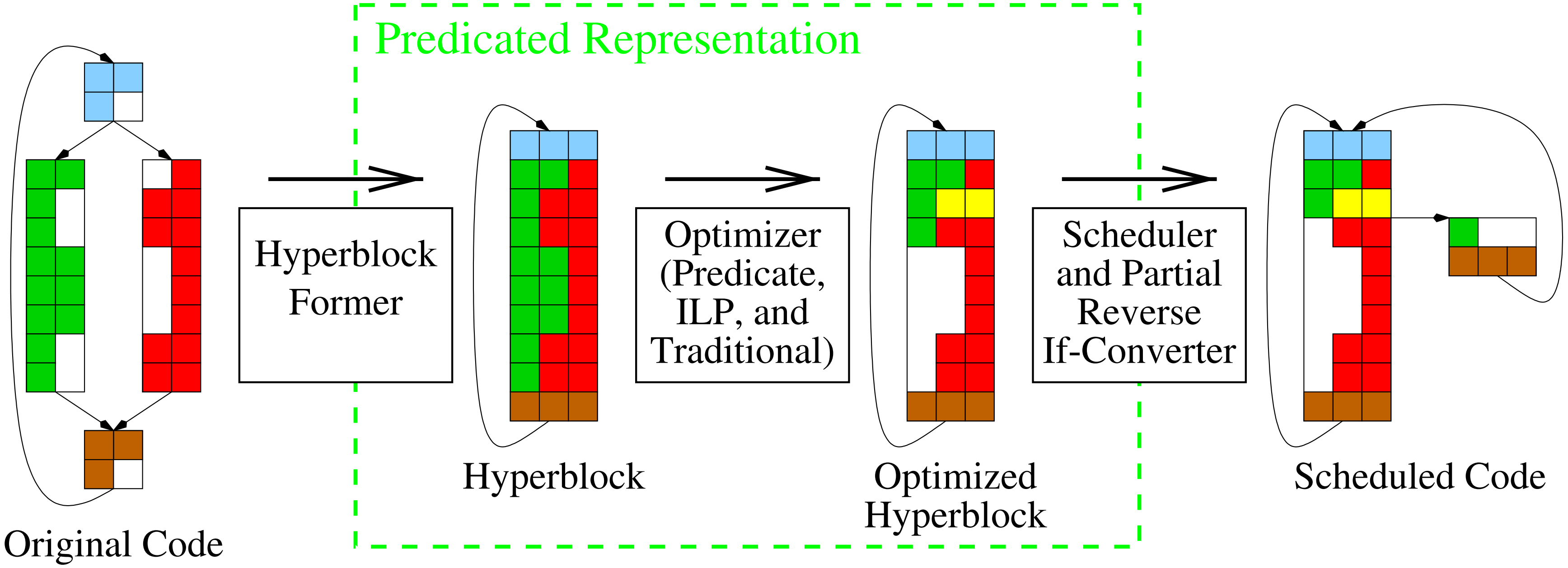
Data Selection Model

A: add r1, r4, r5 (p)
B: load r6, r1, 0

- A reserves its destination register (r1)
- B awaits A for r1 before proceeding
- Necessary when instructions reserve destination registers before their predicate is known or read
- Overhead
  - Destination operand read as additional source operand
  - Output dependent instructions converted to flow dependent instructions

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.



Hyperblocks

- A *hyperblock* is a set of predicated basic blocks in which control can enter only from the top, but may exit from one or more locations.
  - Collection of connected basic blocks
  - Single block dominates all blocks (entry block)
  - Control flow may only enter hyperblock at entry block
  - Control flow may exit hyperblock at any number of locations
  - Control flow between blocks in hyperblock is eliminated
  - Each hyperblock contains an implicit acyclic CFG.

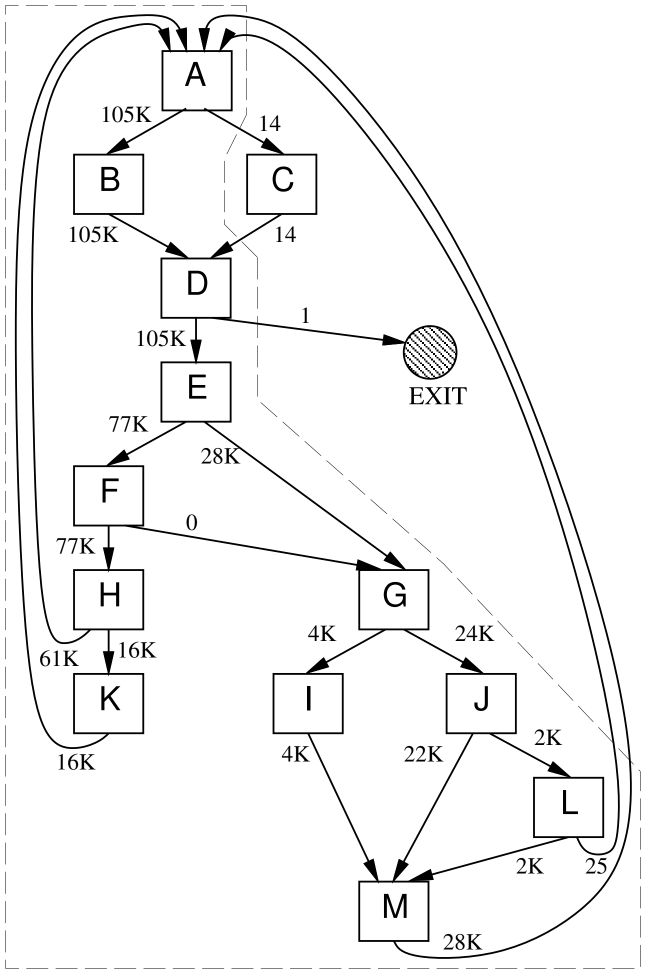
Hyperblocks

- A *hyperblock* is a set of predicated basic blocks in which control can enter only from the top, but may exit from one or more locations.
- Primary contributions:
  - Allows if-conversion of selected traces only
    - Machine resource constraints
    - Relative path height
    - Instructions which hinder optimization
  - Provides for speculation thorough predicate promotion
- In IMPACT, hyperblocks are formed heuristically early in the compilation process.
- Aggressive formation provides enhanced opportunities for optimization, can be adjusted at schedule time.

wc Example

1	A:	add	r6,r6,−1
2		blt	r6,0,C
3	B:	ld_c	r4,mem(r7+0)
4		add	r7,r7,1
5	D:	beq	r4,−1,EXIT
6	E:	add	r1,r1,1
7		bge	32,r4,G
8	F:	bge	r4,127,G
9	H:	bne	0,r8,A
10	K:	add	r2,r2,1
11		add	r8,r8,1
12		jmp	A
13	G:	beq	r4,10,I
14	J:	bne	r4,32,L
15	M:	mov	r8,0
16		jmp	A
17	I:	add	r3,r3,1
18		jmp	M
19	L:	bne	r4,9,A
20		jmp	M
21	C:	st	mem(r5+0),r6
22		st	mem(r5+4),r7
23		mov	Parm0,r5
24		jsr	filbuf
25		mov	r4,Ret0
26		ld	r6,mem(r5+0)
27		ld	r7,mem(r5+4)
28		jmp	D

(a)



(b)

0	A:	pclr	p4,p6	
1		add	r6,r6,−1	
2		blt	r6,0,C	
3		ld_c	r4,mem(r7 + 0)	
4		add	r7,r7,1	
5		beq	r4,−1,EXIT	
6		add	r1,r1,1	
7		pge	p4(OR),p1($\overline{\text{U}}$),32,r4	
8		pge	p4(OR),p2($\overline{\text{U}}$),r4,127	(p1)
9		pne	p3($\overline{\text{U}}$),−,0,r8	(p2)
10		add	r2,r2,1	(p3)
11		add	r8,r8,1	(p3)
13		peq	p7(U),−,r4,10	(p4)
17		add	r3,r3,1	(p7)
13’		peq	p6(OR),p5($\overline{\text{U}}$),r4,10	(p4)
14		pne	p6($\overline{\text{OR}}$),p8(U),r4,32	(p5)
19		peq	p6(OR),−,r4,9	(p8)
15		mov	r8,0	(p6)
16		jmp	A	

Register contents:

r1 = charct	r5 = fp
r2 = wordct	r6 = fp−>cnt
r3 = linect	r7 = fp−>ptr
r4 = c	r8 = token

Predicate association:

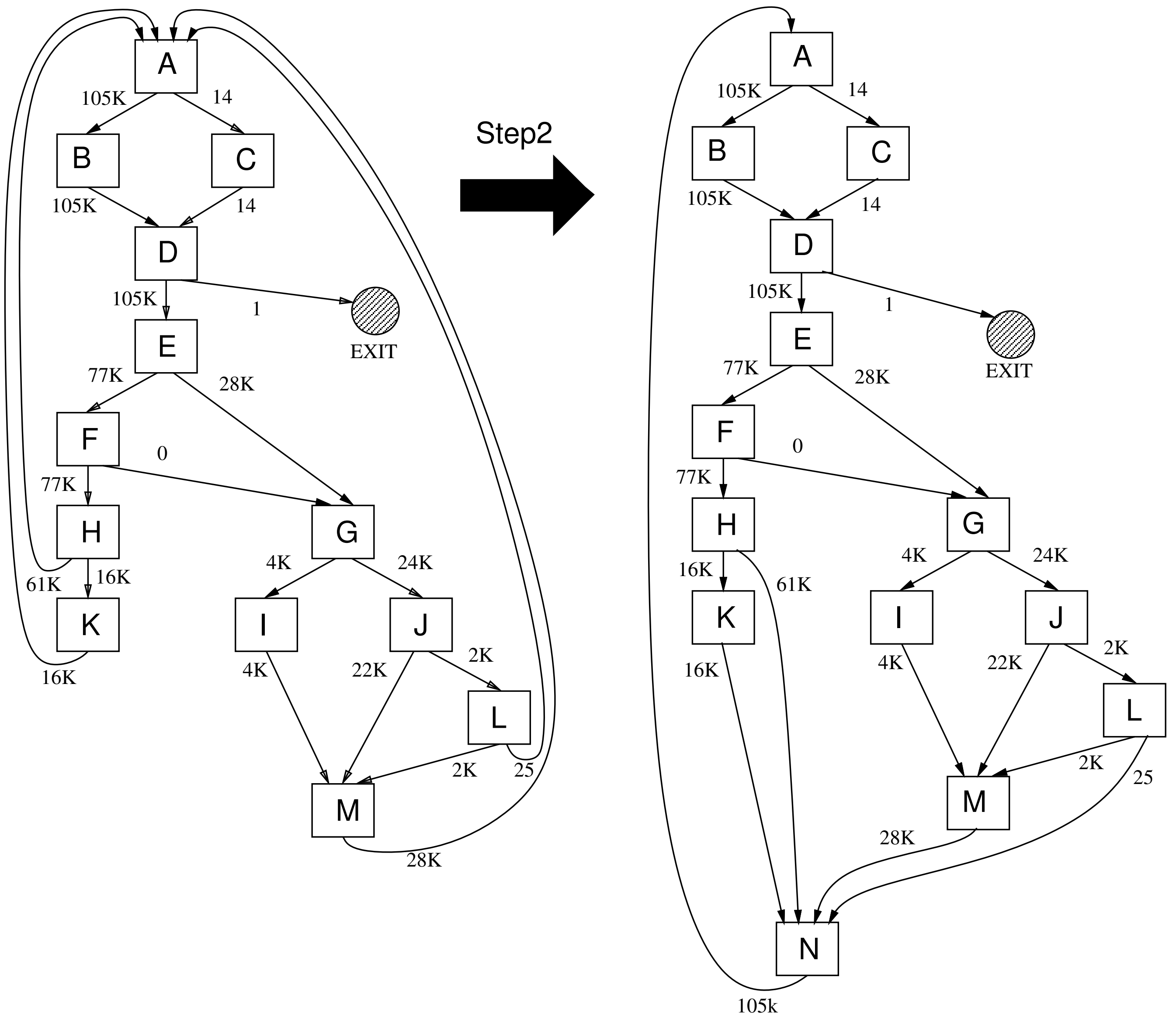
p1 $\sim$ F	p5 $\sim$ J
p2 $\sim$ H	p6 $\sim$ M
p3 $\sim$ K	p7 $\sim$ I
p4 $\sim$ G	p8 $\sim$ L

(c)

Hyperblock Formation

- 5 Step Procedure
  1. Identify region (loop, hammock, dominating block)
  2. Backedge/exit coalescing
  3. Block selection
  4. Tail duplication
  5. If-conversion

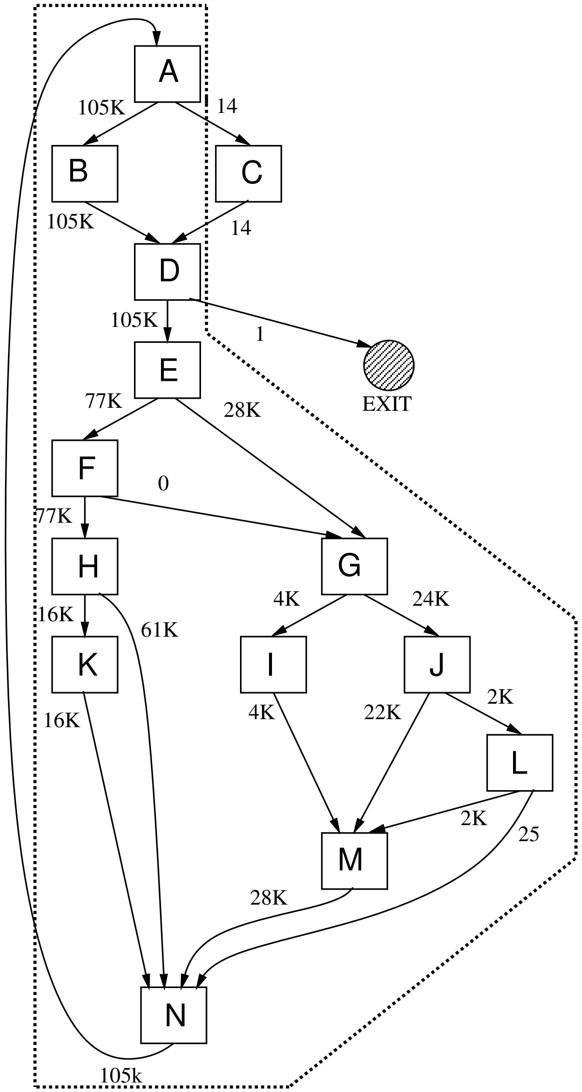
Step 1: Identification and Step 2: Backedge Coalescing



Step 3: Block Selection Heuristic

- Key parameters: Block size, execution frequency, instruction characteristics
- One heuristic: Block Selection Value [Mahlke92]
  $BSV_i = \left(K \times \frac{weight\_bb_i}{size\_bb_i} \times \frac{size\_main\_path}{weight\_main\_path} \times bb\_char_i\right)$
  - $weight$: Normalized against "main path"
  - $size$: Normalized against "main path"
  - K: Machine-dependent issue rate constant
  - bb_char: $0 < bb\_char \leq 1$, desirability of instructions

Step 3: Path Selection Heuristic

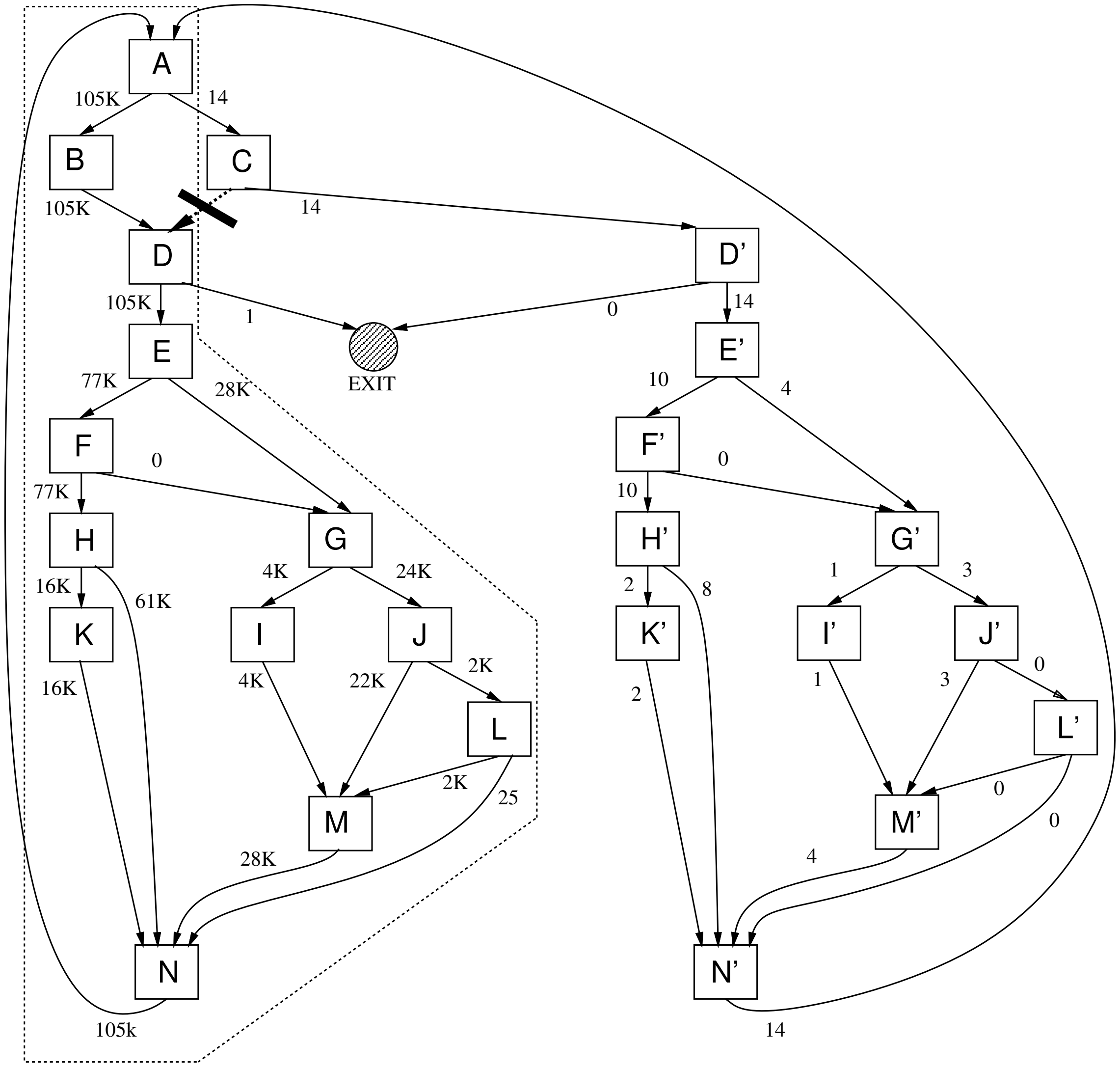


1. A-B-D-E-F-H-N
2. A-B-D-E-F-H-K-N
3. A-B-D-E-G-J-M-N
4. A-B-D-E-G-J-L-M-N
5. A-B-D-E-G-I-M-N
6. A-B-D-E-G-J-L-N
7. A-B-D
8. A-C-D-E-F-H-N
9. A-C-D-E-F-H-K-N
10. A-C-D-E-G-J-M-N
11. A-C-D-E-G-J-L-M-N
12. A-C-D-E-G-I-M-N
13. A-C-D-E-G-J-L-N
14. A-C-D
15. A-B-D-E-F-G-I-M-N
16. A-B-D-E-F-G-J-M-N
17. A-B-D-E-F-G-J-L-M-N
18. A-B-D-E-F-G-J-L-N
19. A-C-D-E-F-G-I-M-N
20. A-C-D-E-F-G-J-M-N
21. A-C-D-E-F-G-J-L-M-N
22. A-C-D-E-F-G-J-L-N

Challenges of Static Selection

- Hyperblock formation can be hazardous
  - Machine resource constraints
  - Path height mismatch—hope to penalize only infrequent paths
- Mechanisms that form more optimization-prone hyperblocks can blow up code size if overapplied, or may be unapplicable.
  - Tail duplication
  - Loop peeling
- Creation of hyperblocks to minimize code size rather than to maximize performance may be possible w/o profile info [Connors99]

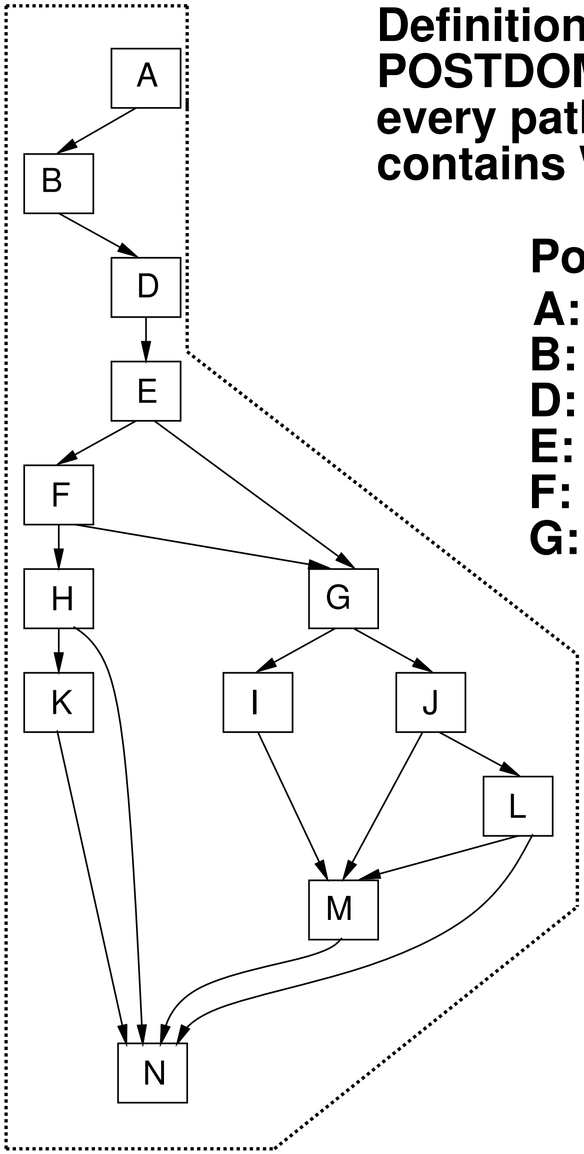
Step 4: Tail Duplication



Step 5: If-Conversion Using RK Algorithms [Park91]

- Developed from Program Dependence Graph [Ferrante87] and Cydrome Compiler [Dehnert93] algorithms.
- Begins with finding Control Dependence Sets (CD).
- Partitions if-conversion into two subproblems:
  - R Assignment of guard predicates to blocks
  - K Placement of predicate defining instructions
- RK minimizes both the number of predicates and the number of predicate defining instructions used.
- IMPACT uses adapted RK techniques.

Step 5: If-Conversion - Post Domination



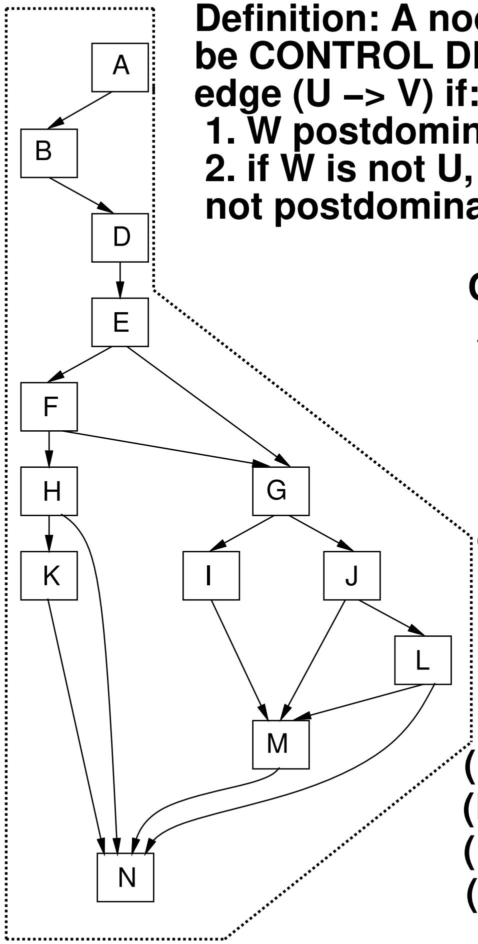
Definition: A node W is said to POSTDOMINATE a node V if every path from V to END contains W.

Post Dominator Sets

A: A
B: AB
D: ABD
E: ABDE
F: F
G: G
H: H
I: I
J: J
L: L
K: K
M: IM

N: ABDEFGHIJKLMN

Step 5: If-Conversion - Control Dependence



Definition: A node W is said to be CONTROL DEPENDENT on edge (U –> V) if:
1. **W postdominates V, and**
2. **if W is not U, then W does not postdominate U.**

Control Dependence Sets

A:	I: (G,I)
B:	J: (G,J)
D:	K: (H,K)
E:	L: (J,L)
F: (E,F)	M: (G, I)(J,M)(L,M)
G: (E,G)(F,G)	N:
H: (F,H)	

Each Unique Set Gets A Predicate

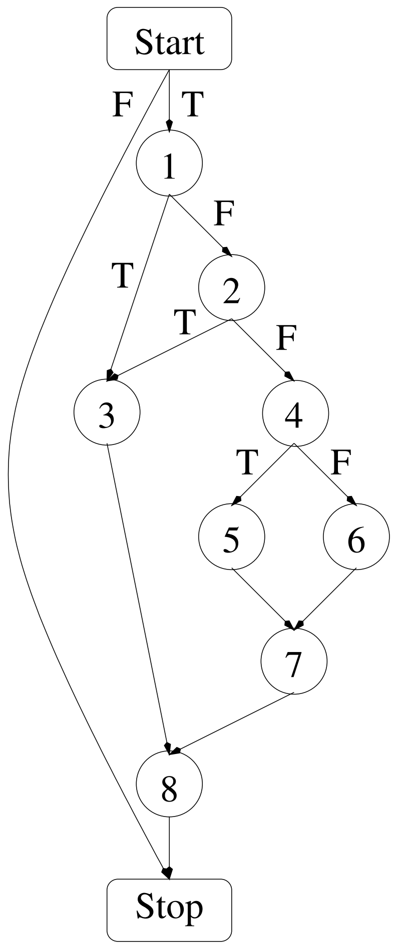
(E,F) = P1	(H,K) = P6
(E,G)(F,G) = P2	(G,J) = P5
(F,H) = P3	(J,L) = P7
(G,I) = P4	

(G,I)(J,M)(L,M) = P8

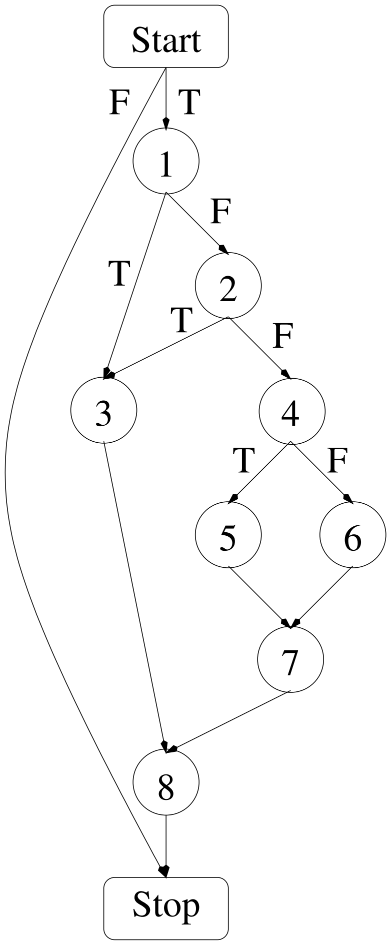
Control Dependence Sets

- Described by Ferrante, et al. in PDG work
- Post-dominator on the Control Flow Graph
- Control Dependence: Given a control flow graph G with nodes X and Y, Y is *control dependent* on X iff:
  - There exists a path P from X to Y with any Z in P post-dominated by Y
  - X is not post-dominated by Y

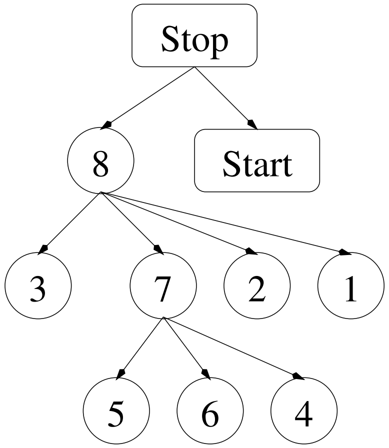
```
if( x < 3 || x > 7 )(1,2)
  l = l + 1;  (3)
else
  if ( x == 5 )(4)
    m = m + 1;(5)
  else
    n = n + 1;(6)
  o = o + 1;(7)
c = c + 1;(8)
```



Extracting Control Dependence Sets



Control Flow Graph



Post-dominator Tree

1	(Start,T)
2	(1,F)
3	(1,T),(2,T)
4	(2,F)
5	(4,T)
6	(4,F)
7	(2,F)
8	(Start,T)

CD(N)

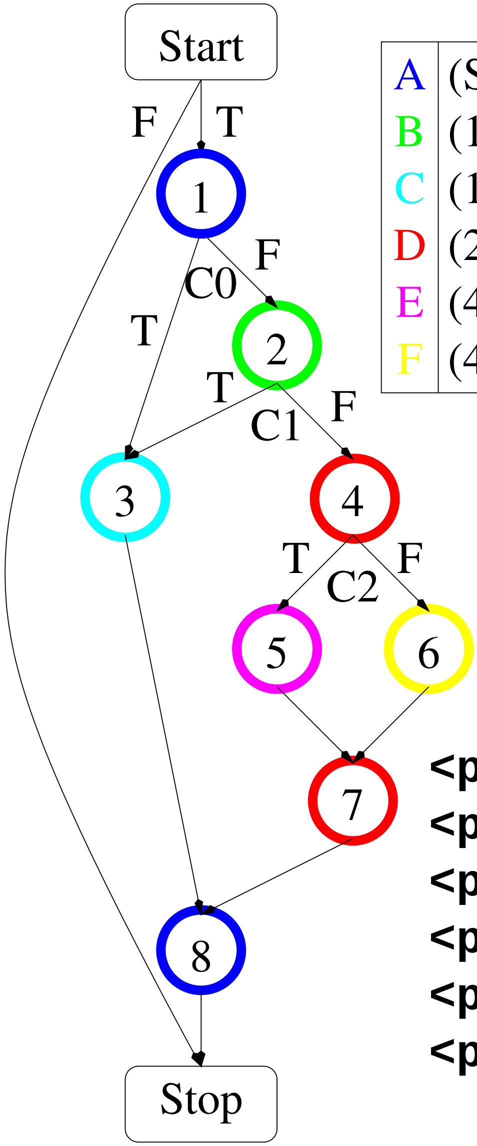
- Consider the set S of edges (A, B) in the CFG such that B does not post-dominate A.
- For each edge $s \in S$ find the least-common ancestor L of A and B in the post-dominator tree. Two cases:
  - L is a parent of A: All nodes in the pdom tree on the path from L to B, including B but not L, are control dependent on A.
  - $L = A$: All nodes in the pdom tree on the path from A to B, inclusive, are control dependent on A.

[Ferrante87]

If-Conversion using RK [Park91]

- RK partitions nodes into equivalence classes using CD information:
  - Every node x belongs to one and only one class $p = R(x)$
  - Nodes with identical set $K(p)$ of control dependences belong to the same class p.
- RK can be summarized:
  - Assign an unique predicate to each unique set of control dependences.
  - Assign each block exactly one guarding predicate according to its control dependence set.
  - Assign a predicate defining instruction for each control dependence arc of each predicate at the head of the arc.
  - Add predicate initializing instructions as required.
- If-conversion is completed by assembling the included blocks into a single block in a topologically compatible order.

If-Conversion Example



A	(Start, T)		T
B	(1,F)	p1	C0’
C	(1,T), (2,T)	p2	C0 + p2 C1
D	(2,F)	p3	p2 C1’
E	(4,T)	p4	p4 C2
F	(4,F)	p5	p4 C2’

```
      pred_clear(p2)
      p1_uf,p2_ot=(C0)
<p1> p2_ot, p3_uf=(C1)
<p2> l=l+1
<p3> p4_ut, p5_uf=(C2)
<p4> m=m+1
<p5> n=n+1
<p3> o=o+1
      c=c+1
```

Completed Hyperblock

```
Loop: pred_clr  p4, p6
      ld  r98, mem(r3 + 0)
      add  r27, r98, -1
      st  mem(r3 + 0), r27
      blt  r98, 1, LC
      ld  r30, mem(r3 + 4)
      add  r29, r30, 1
      st  mem(r3 + 4), r29
      ld  r4, mem(r30 + 0)
      beq  r4, -1, EXIT
      ld  r33, mem(r73 + 0)
      add  r32, r33, 1
      st  mem(r73+0), r32
      pred_ge  p4(OR), p1(Ū), 32, r4
      pred_ge  p4(OR), p2(Ū), r4, 127  (p1)
      pred_eq  p3(U), -, 0, r2 (p2)
      pred_eq  p6(OR), p5(Ū), r4, 10  (p4)
      pred_eq  p7(U), -, r4, 10 (p4)
      pred_eq  p6(OR), p8(Ū), r4, 32  (p5)
      ld  r36, mem(r72+0)  (p3)
      add  r35, r36, 1  (p3)
      st  mem(r72+0), r35  (p3)
      add  r2, r2, 1  (p3)
      ld  r39, mem(r71 + 0)  (p7)
      add  r38, r39, 1  (p7)
      st  mem(r71 + 0), r38  (p7)
      pred_eq  p6(OR), -, r4, 9  (p8)
      mov  r2, 0  (p6)
      jump  Loop
```

Peformance Summary for Wc

8-issue, 1BR per cycle
Perfect I/D caches

	Base	Pred
Dynamic Branches	572K	315K
Mispredictions	52K	56
MPR	9.0%	0.0%
Distance betw Brs	2.0	6.8
Distance betw MPs	46	56K
Performance (8 issue)	637K	289K

Size and Completion Rate

		HB completion ratio			
Benchmark	size	$\geq$0.90	$\geq$0.70	$\geq$0.50	$\geq$0.30
052.alvinn	115	0.97	0.99	0.99	1.00
056.ear	113	0.92	0.93	0.95	0.98
022.li	24	0.62	0.73	0.81	0.94
023.eqntott	50	0.85	0.88	0.92	0.96
026.compress	39	0.61	0.84	0.84	0.92
cmp	148	0.99	0.99	1.00	1.00
grep	45	0.68	0.86	0.91	0.96
lex	46	0.80	0.89	0.96	0.98
qsort	48	0.74	0.75	0.85	0.89
wc	131	0.99	1.00	1.00	1.00

Comparing to Superblock

		SB completion ratio			
Benchmark	size	$\geq$0.90	$\geq$0.70	$\geq$0.50	$\geq$0.30
052.alvinn	115	0.97	0.99	0.99	1.00
056.ear	81	0.59	0.66	0.79	0.97
022.li	22	0.63	0.74	0.82	0.94
023.eqntott	21	0.46	0.46	0.54	0.63
026.compress	29	0.34	0.42	0.47	0.68
cmp	113	0.81	0.85	0.88	0.93
grep	42	0.75	0.82	0.90	0.96
lex	37	0.66	0.74	0.91	0.96
qsort	33	0.42	0.52	0.82	0.89
wc	24	0.45	0.72	0.80	0.96