Software Pipelining Loops Containing “IF” Statements:

Making more loops run faster with limited resources.

Mark Stoodley
November 1994

Today's Journey

1. **Introduction**
   - Context
   - Loop Example
2. Various Scheduling Algorithms
3. The New Algorithm
4. Preliminary Results
5. Summary

Context for this Work

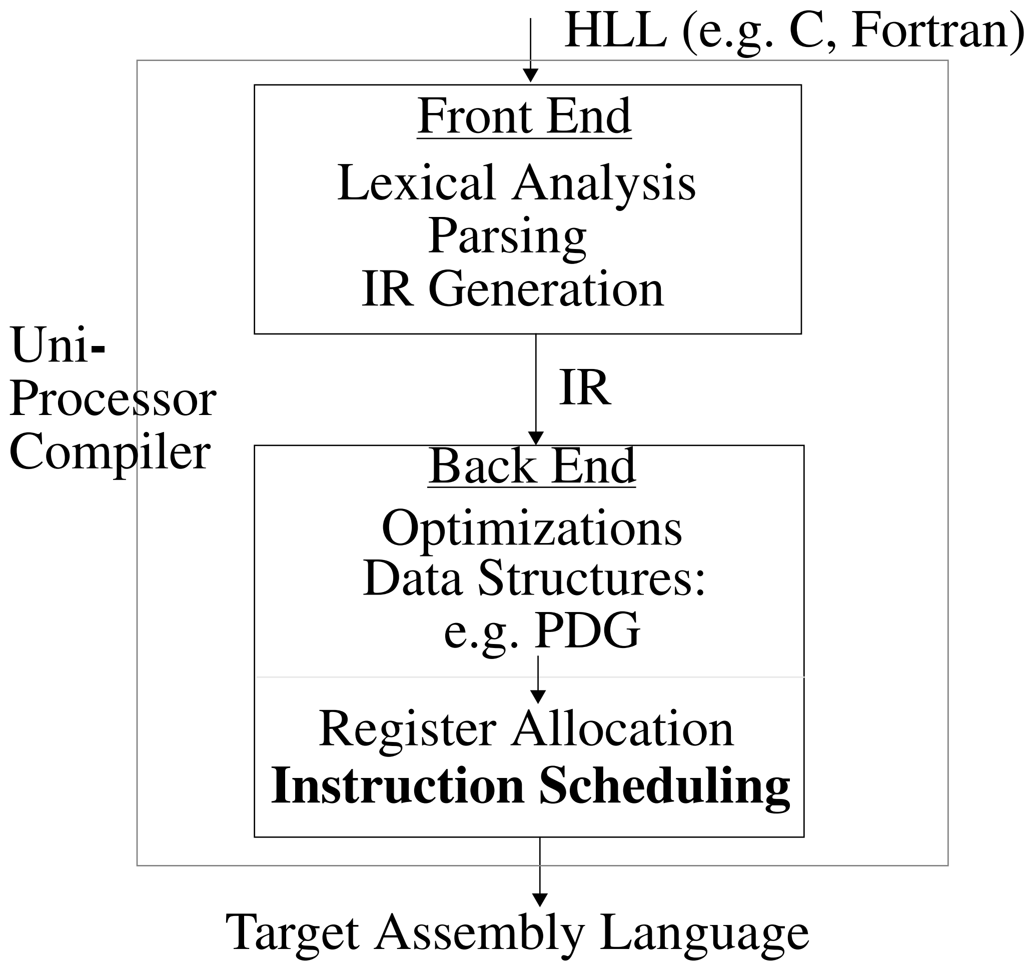
VLIW Architecture

- Allocation of operations to functional units is exposed to the compiler
- Compiler specifies either an operation or a nop for every functional unit for every instruction cycle in the program
- Examples in this talk: one integer unit, one floating point unit, and one control (branch) unit
- Will be assuming single-cycle latency for all operations

Why Software Pipeline?

- Want to fill instructions with as many operations as possible
- Limited parallelism in single basic blocks (usually 2 to 3)
- Software pipelining can schedule operations from many different iterations of a loop in one instruction -> “fuller” instructions
- Can increase performance of loops dramatically
  - loop speedups of 2 to 3 or even more are possible

Compiler Context for Instruction Scheduling



Loop Code Example

Example C Loop

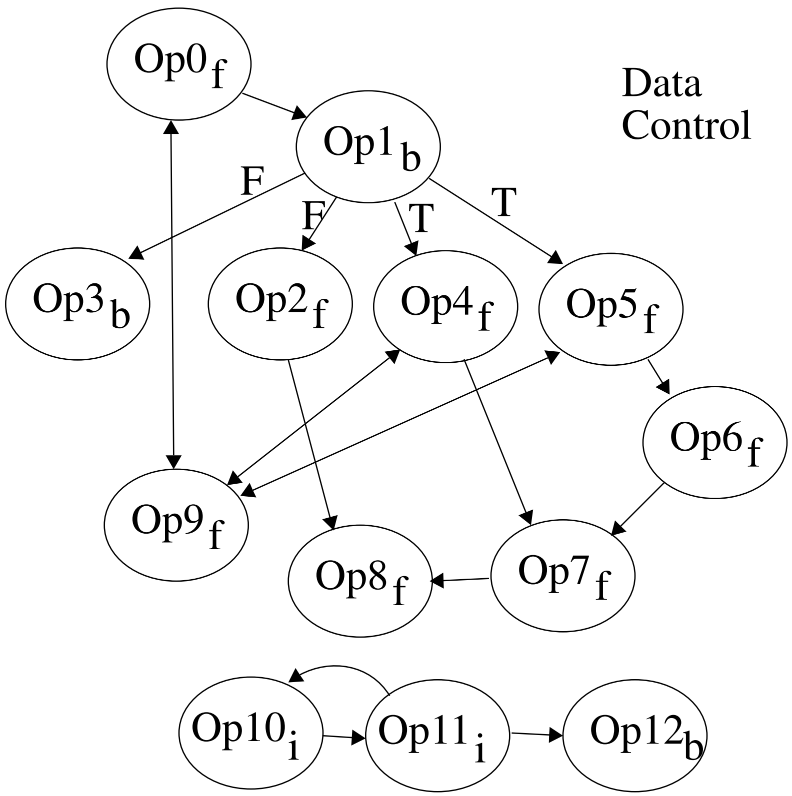
```
for (i = 0;i < N;i++) {
      if (x < threshold)
            z = threshold;
      else
            z = (x + 1.0) * (x/2 + 3.0);
      sum += z;
      x = x * x;
}
```

Name	Assembly Code	
Op0	L7: fslt.s x,threshold,f1	Test
Op1	beqz.s f1,L5	
Op2	fmov.s threshold,f10	Then
Op3	jmp L6	
Op4	L5: fadd.s x,f9,f2	Else
Op5	fdiv.s x,f8,f3	
Op6	fadd.s f3,f7,f11	
Op7	fmult.s f2,f11,f10	
Op8	L6: fadd.s sum,f10,sum	
Op9	fmult.s x,x,x	
Op10	add.d i,d4,i	Loop Control
Op11	slt.d i,N,d1	
Op12	bnez.d d1,L7	

Program Dependence Graph Example

Block	Name	Assembly Code
BB0	Op0	L7: fslt.s x,threshold,f1
	Op1	beqz.s f1,L5
BB1	Op2	fmov.s threshold,f10
	Op3	jmp L6
BB2	Op4	L5: fadd.s x,f9,f2
	Op5	fdiv.s x,f8,f3
	Op6	fadd.s f3,f7,f11
	Op7	fmult.s f2,f11,f10
BB3	Op8	L6: fadd.s sum,f10,sum
	Op9	fmult.s x,x,x
	Op10	add.d i,d4,i
	Op11	slt.d i,N,d1
	Op12	bnez.d d1,L7

Program Dependence Graph



Today's Journey

1. Intro & Context

2. Various Scheduling Algorithms
   - List Scheduling
   - SP Without IF
   - SP With IF

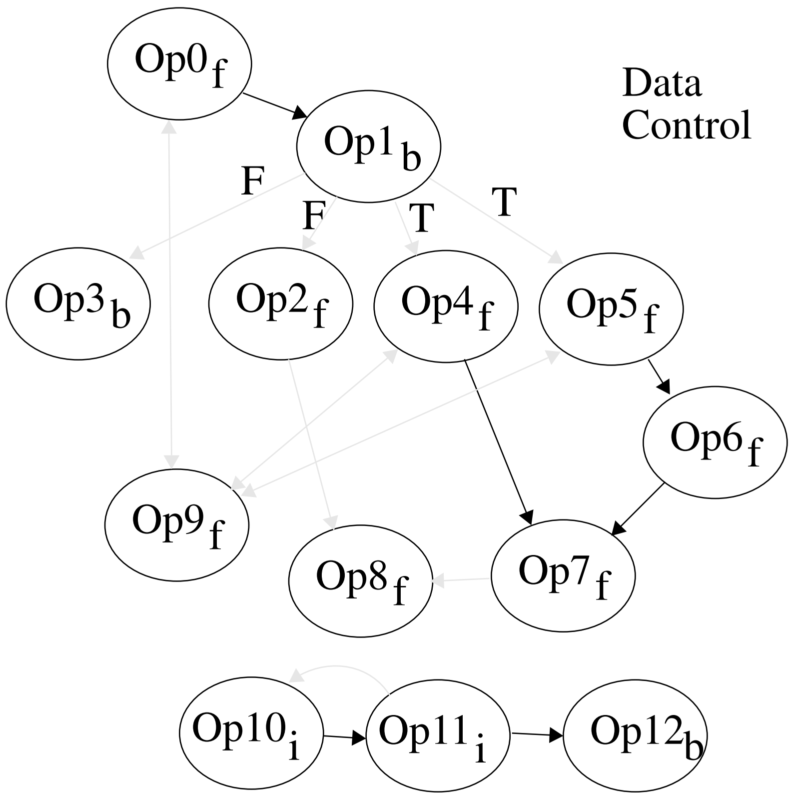
3. The New Algorithm

4. Preliminary Results

5. Summary

List Scheduling

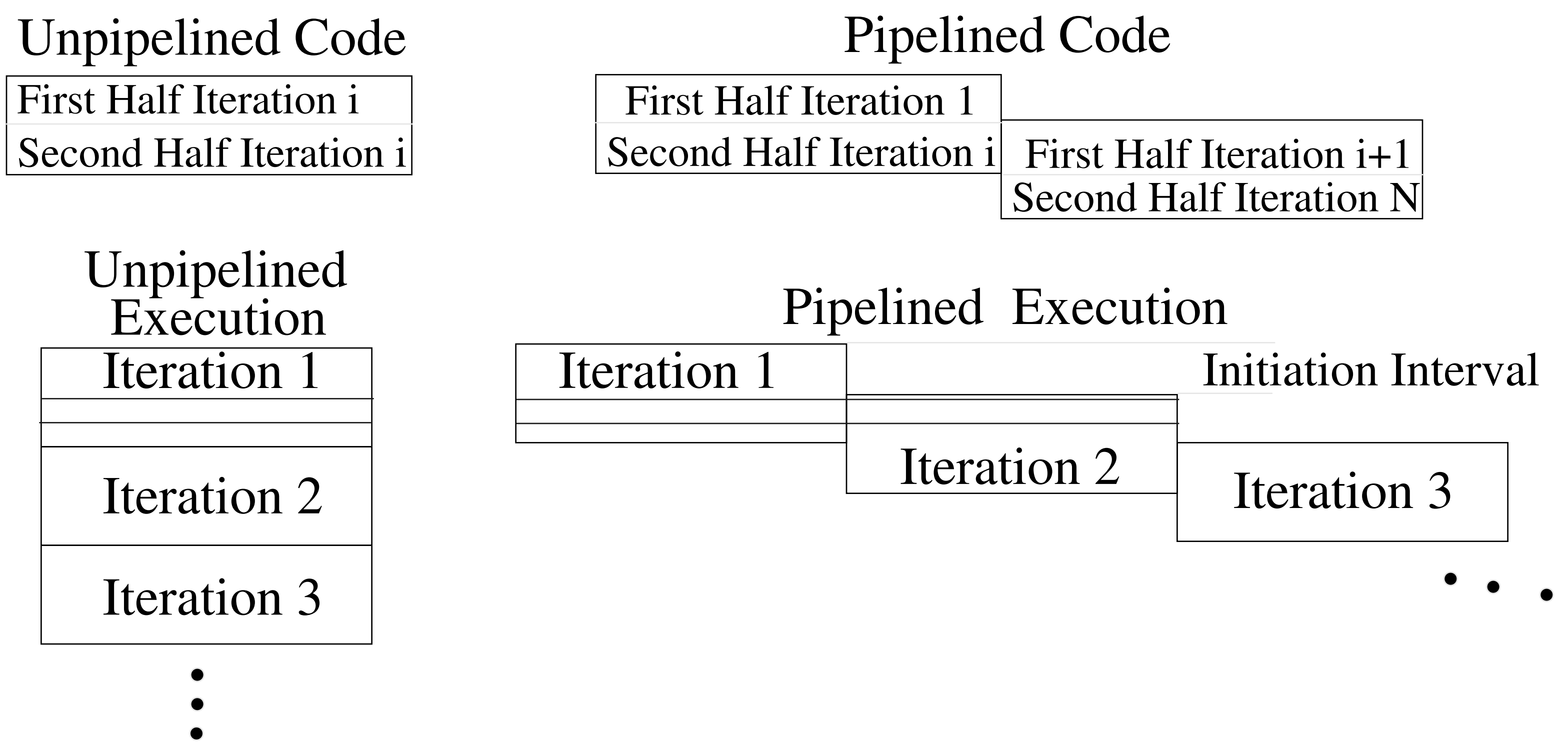
Program Dependence Graph



List Schedule

	Int	FP	Branch
L0		Op0	
L1			Op1:L3
L2		*Op2*	Op3:L7
L3		**Op5**	
L4		**Op6**	
L5		**Op4**	
L6		**Op7**	
L7	Op10	Op8	
L8	Op11	Op9	
L9			Op12:L0

Software Pipelining

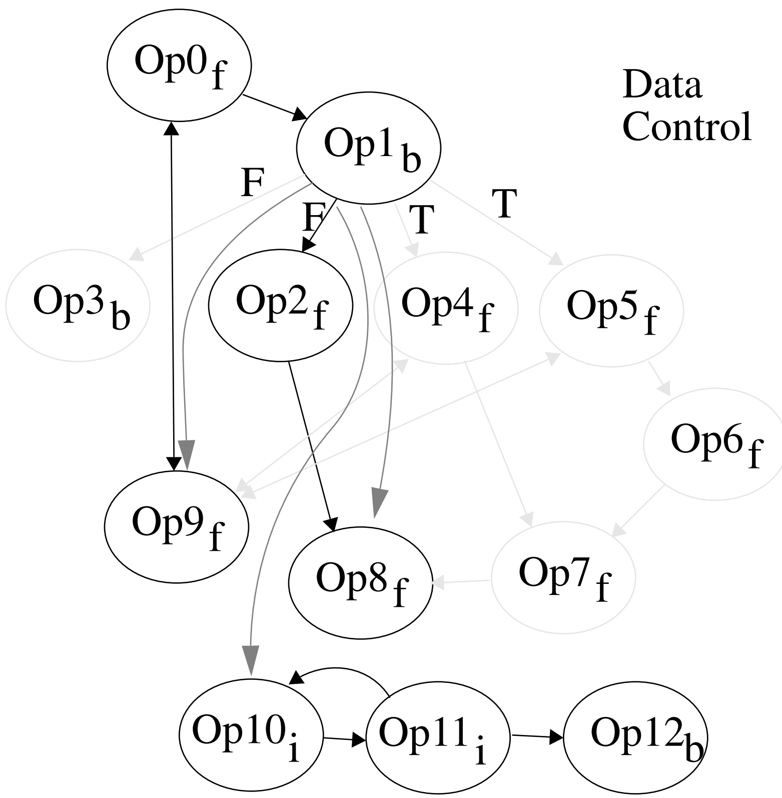


Initiation Interval (II) determines amount of overlap and loop speedup

-> want it small, ideally one cycle

Software Pipelining Example (No IF Statements)

Program Dependence Graph



Schedule

Label	Int	FP	Branch
L0		Op0	
L1			Op1
L2	Op10	Op9	
L3	Op11	Op2	
L4			
L5		Op8	
L6			Op12

Op0

Op1

Op10 Op9

L7 Op11 Op2

L8

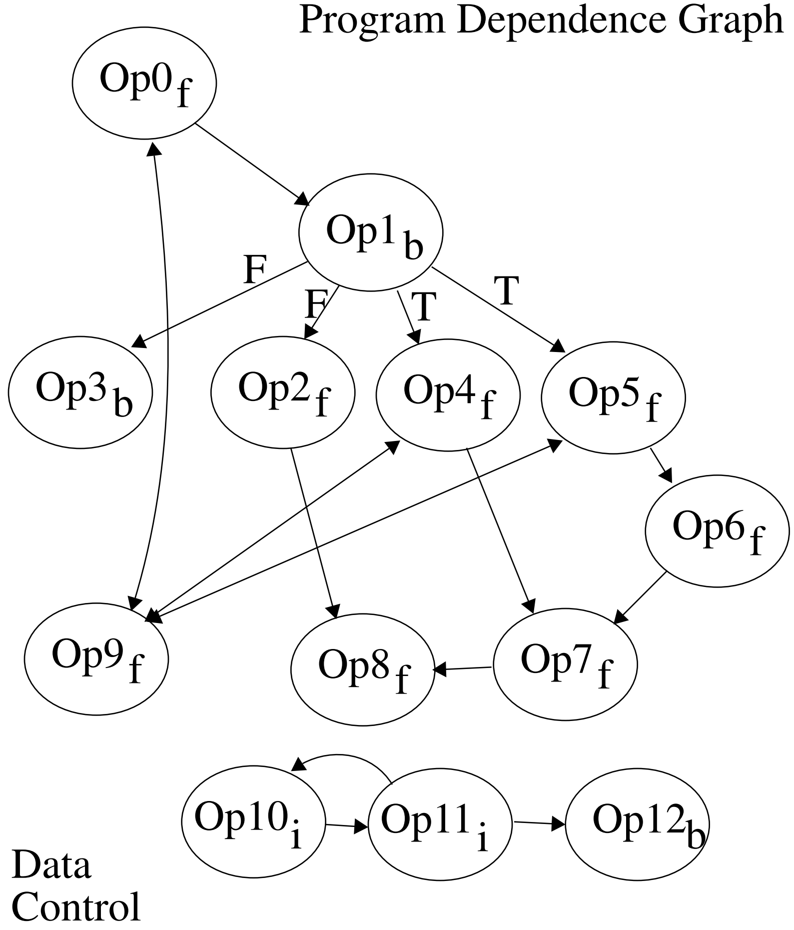
L9 Op8

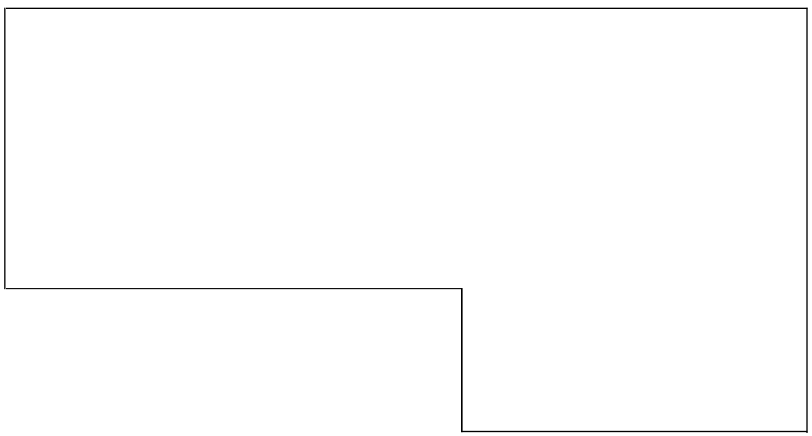
L10 Op12

		Op0	
			Op1
	Op10	Op9	
L11	Op11	Op2	
L12			
L13		Op8	
L14			Op12

Software Pipelining Example (IF Statements) Using Hierarchical Reduction

Program Dependence Graph





Label	**Int**	**FP**	**Branch**
L0			Op1:L6
L1	*nop*	*nop*	*nop*
L2	*nop*	*nop*	*nop*
L3	*nop*	*nop*	*nop*
L4	*nop*	*Op2*	*Op3:L9*
L5		**Op5**	
L6		**Op6**	
L7		**Op4**	
L8		**Op7**	
L9			

Software Pipelining with IF Statements

Is there an advantage to having different II through different paths of execution??

Consider:

i) loop with 2 possible paths of execution (one ***k*** cycles, the other ***a*** cycles less -> ***k-a*** cycles long)

ii) the loop is executed ***N*** times

iii) a fraction ***f*** of the iterations go through the short branch

$$ExecutionTime|_{Worst} = k \times N$$

$$ExecutionTime|_{Best} = (1-f) \times k \times N + f \times (k-a) \times N = kN - faN$$

$$Speedup = \frac{ExecutionTime|_{Worst}}{ExecutionTime|_{Best}} = \frac{kN}{kN - faN} = \frac{1}{1 - \frac{fa}{k}}$$

Some Example Loop Speedups

Assuming equal branch probabilities (i.e. $\boldsymbol{f} = 0.5$):

i) $\boldsymbol{k} = 5$, $\boldsymbol{a} = 1$ (save one cycle on one path through the loop)

$$Speedup = \frac{1}{1 - \frac{a}{2k}} = \frac{1}{1 - \frac{1}{10}} = \frac{10}{9} \cong 1.11$$

ii) $\boldsymbol{k} = 7$, $\boldsymbol{a} = 3$ (save 3 cycles on one path through the loop)

$$Speedup = \frac{1}{1 - \frac{a}{2k}} = \frac{1}{1 - \frac{3}{14}} = \frac{14}{11} \cong 1.27$$

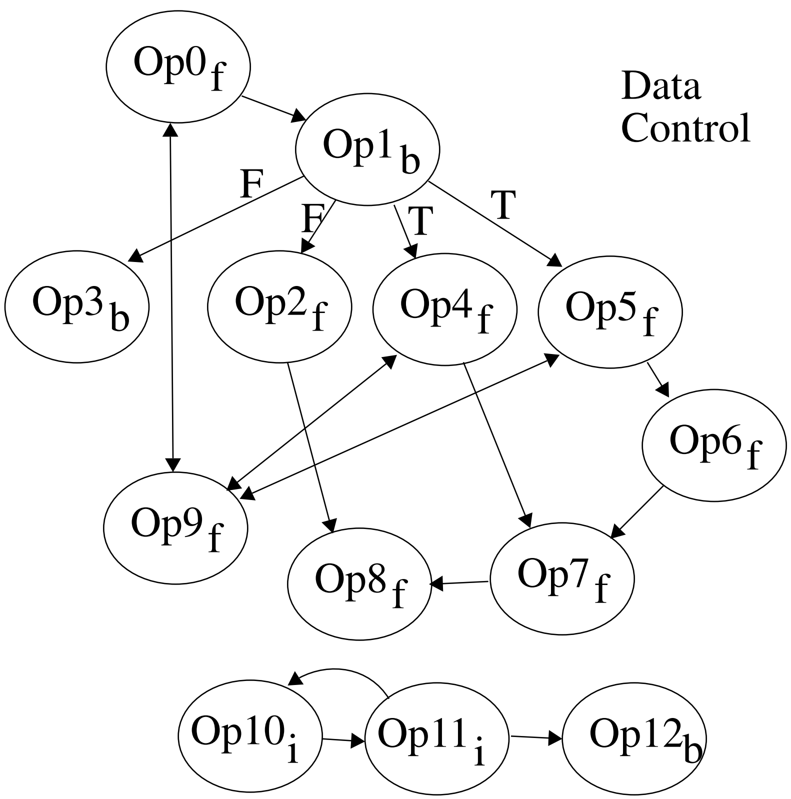
Best Case for $\boldsymbol{k} = 7$, $\boldsymbol{a} = 3$: $\boldsymbol{f} = 1.0$

$$Speedup = \frac{1}{1 - \frac{a}{k}} = \frac{1}{1 - \frac{3}{7}} = \frac{7}{4} \cong 1.75$$

Software Pipelining Example Using Enhanced Percolation Scheduling

Minimum II for short path is 4 cycles, for long path is 7 cycles.

Program Dependence Graph



Label	Int	FP	Branch
L0	Op10	Op0	
L1	Op11		Op1

		Op2	
L3		**Op5**	
L4		**Op6**	
L5		**Op4**	
L6		**Op7**	
L7		**Op9**	
L8		**Op8**	
L9	Op10	Op0	Op12

		Op5	
L2		*Op2*	*Op3:L6*
L3		**Op6**	
L4		**Op4**	
L5		**Op7**	
L6		**Op9**	
L7		**Op8**	
L8	Op10	**Op0**	**Op12**

Today's Journey

1. Introduction

2. Various Scheduling Algorithms

3. The New Algorithm
   All-Paths Pipelining

4. Preliminary Results

5. Summary

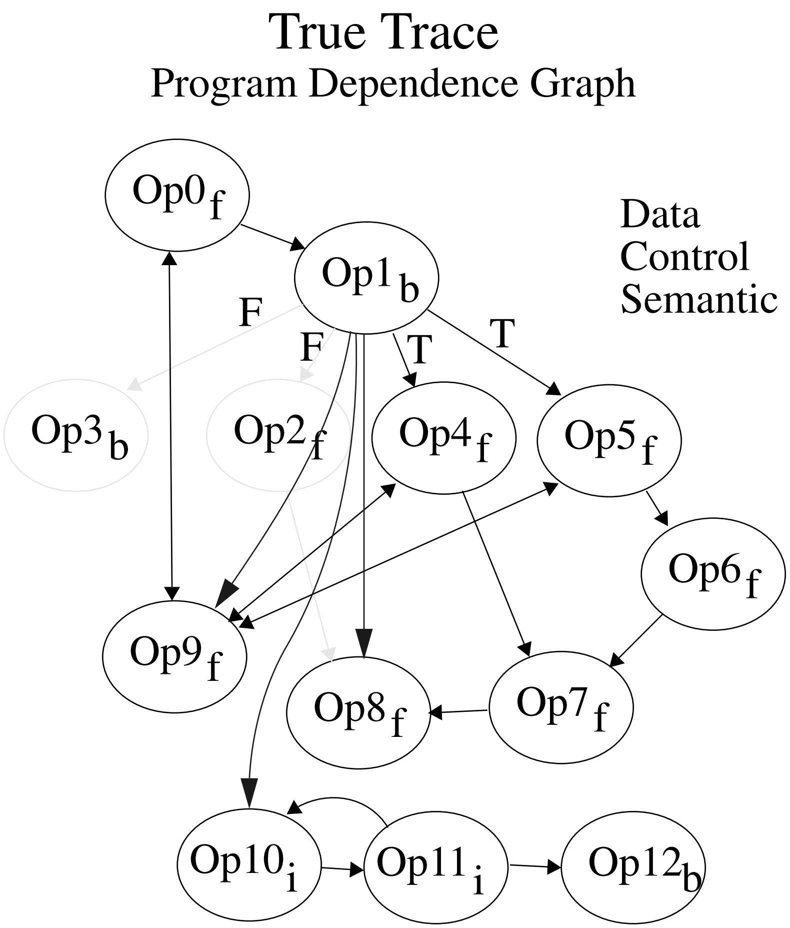
Algorithm Overview

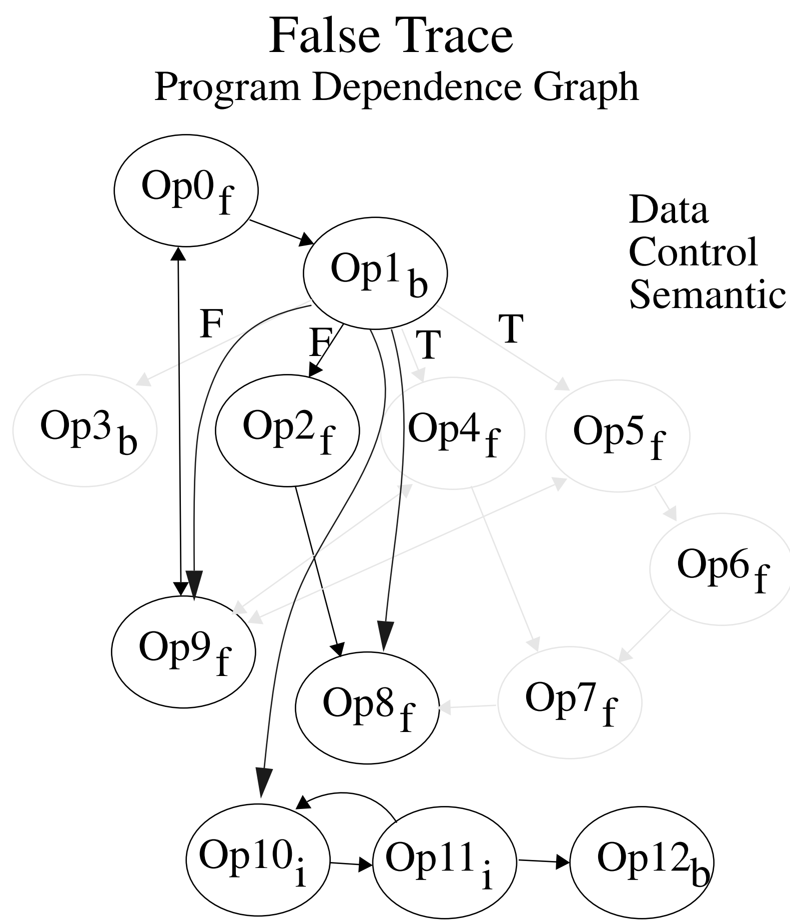
1. Create traces for each path of execution through the loop (some extra semantics-preserving edges are added to the PDG for each trace).

2. Software pipeline each trace in isolation (potentially getting different II for each trace).

3. Create the introductory and intermediate instructions necessary to allow any execution path to be executed by the new code.

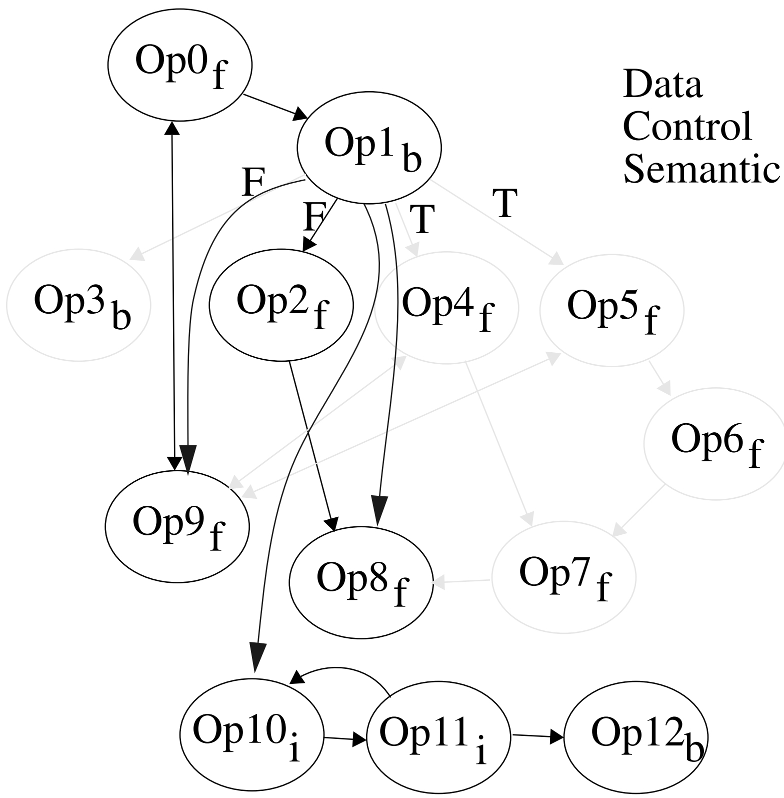
Step One: Extract Trace PDGs





Step Two: Software Pipeline Each Trace--Schedule for False Trace

False Trace
Program Dependence Graph



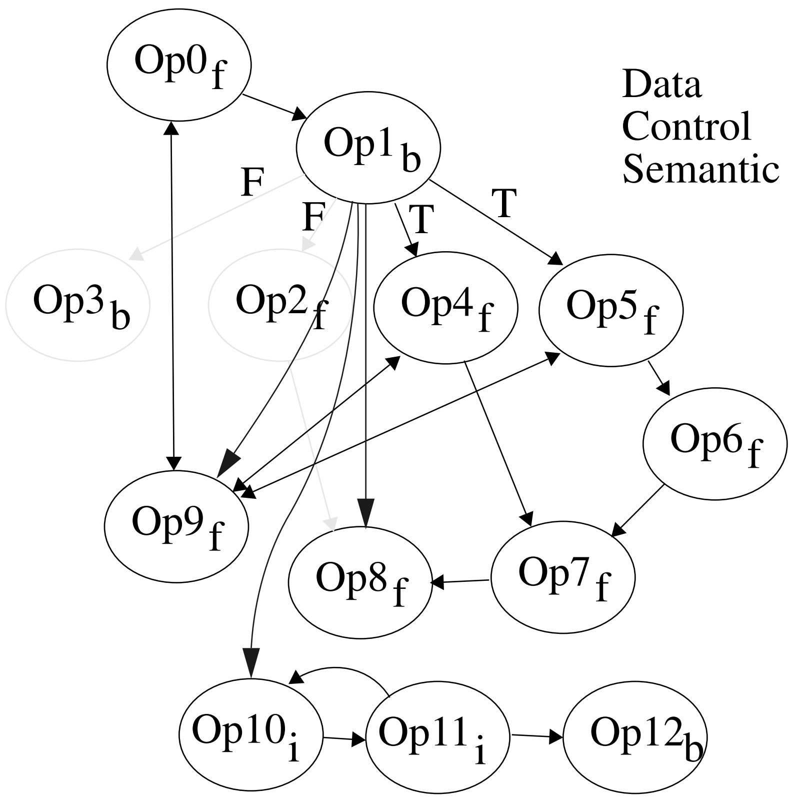
Label	Int	FP	Branch
L0		Op0	
L1			Op1
L2		Op9	

	Op10
L3	Op11

	Op2	
L4		
L5	Op8	
L6		Op12

Software Pipeline Schedule for True Trace

True Trace
Program Dependence Graph



Label	Int	FP	Branch
L0		Op0	
L1			Op1
L2	Op10	**Op5**	
L3	Op11	**Op4**	
L4		Op9	
L5		**Op6**	
L6		**Op7**	
L7			
L8		Op8	
L9			Op12

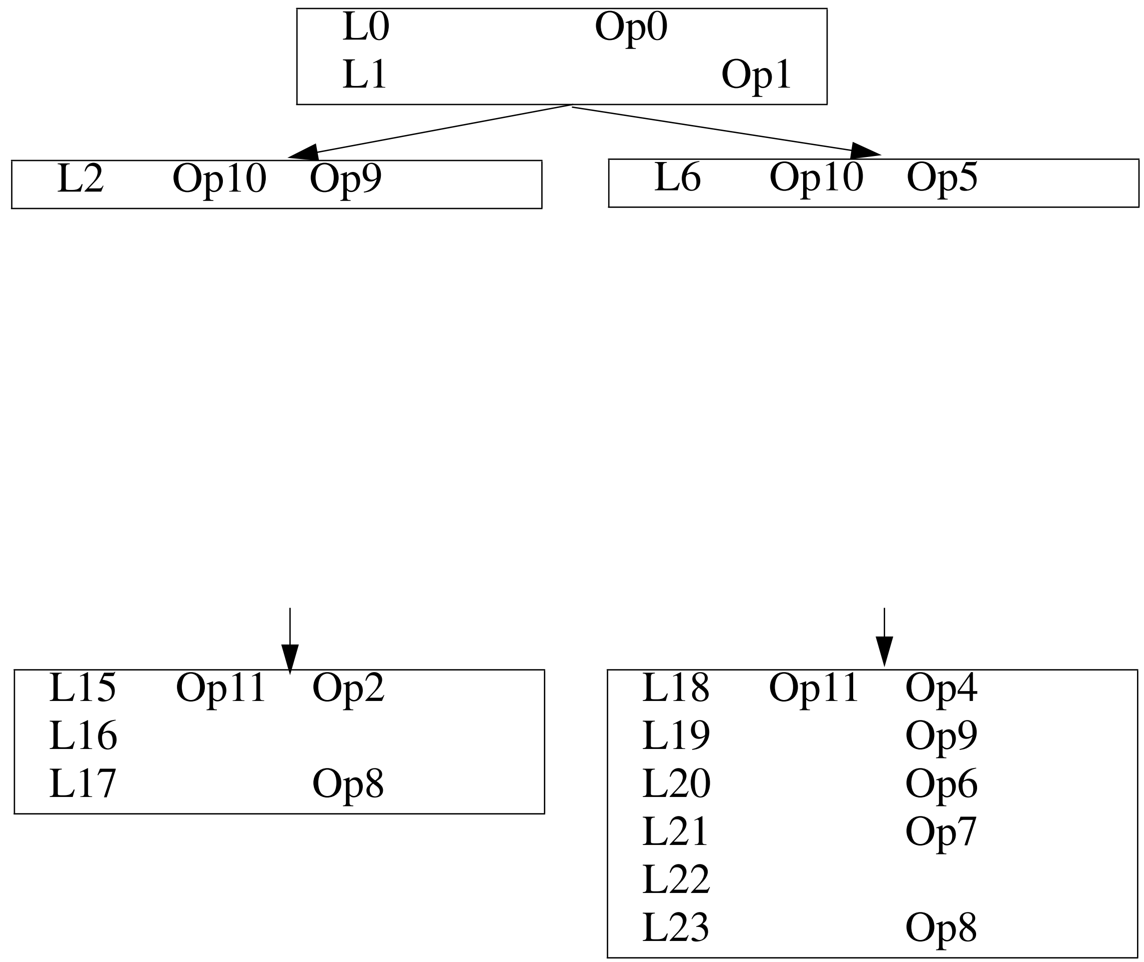
Step Three: Combine Traces

L3	*Op11*	*Op2*	
L4		*Op0*	
L5		*Op8*	*Op1*

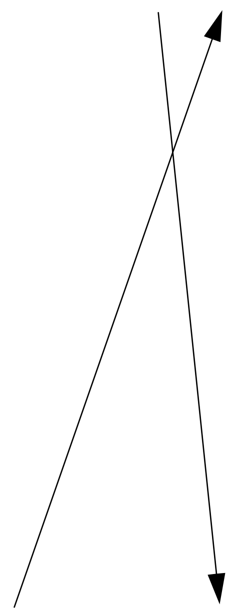
L13	*Op10*	*Op9*	*Op12*

L7	**Op11**	**Op4**	
L8		**Op9**	
L9		**Op6**	
L10		**Op7**	
L11		**Op0**	
L12		**Op8**	**Op1**

L14	**Op10**	**Op5**	**Op12**



L0 Op0
L1 Op1
L2 Op10 Op9
L6 Op10 Op5
L15 Op11 Op2
L16
L17 Op8
L18 Op11 Op4
L19 Op9
L20 Op6
L21 Op7
L22
L23 Op8



Summary of Loop Execution

Technique	II for False Path	II for True Path	Average Cycles per Iteration	Loop Speedup
List Scheduling	6	9	7.5	1.0
SP: Hierarchical Reduction	7	7	7	1.07
SP: Enhanced Percolation Scheduling	4	8	6	1.25
SP: All Paths	4	7	5.5	1.36

Preliminary Results

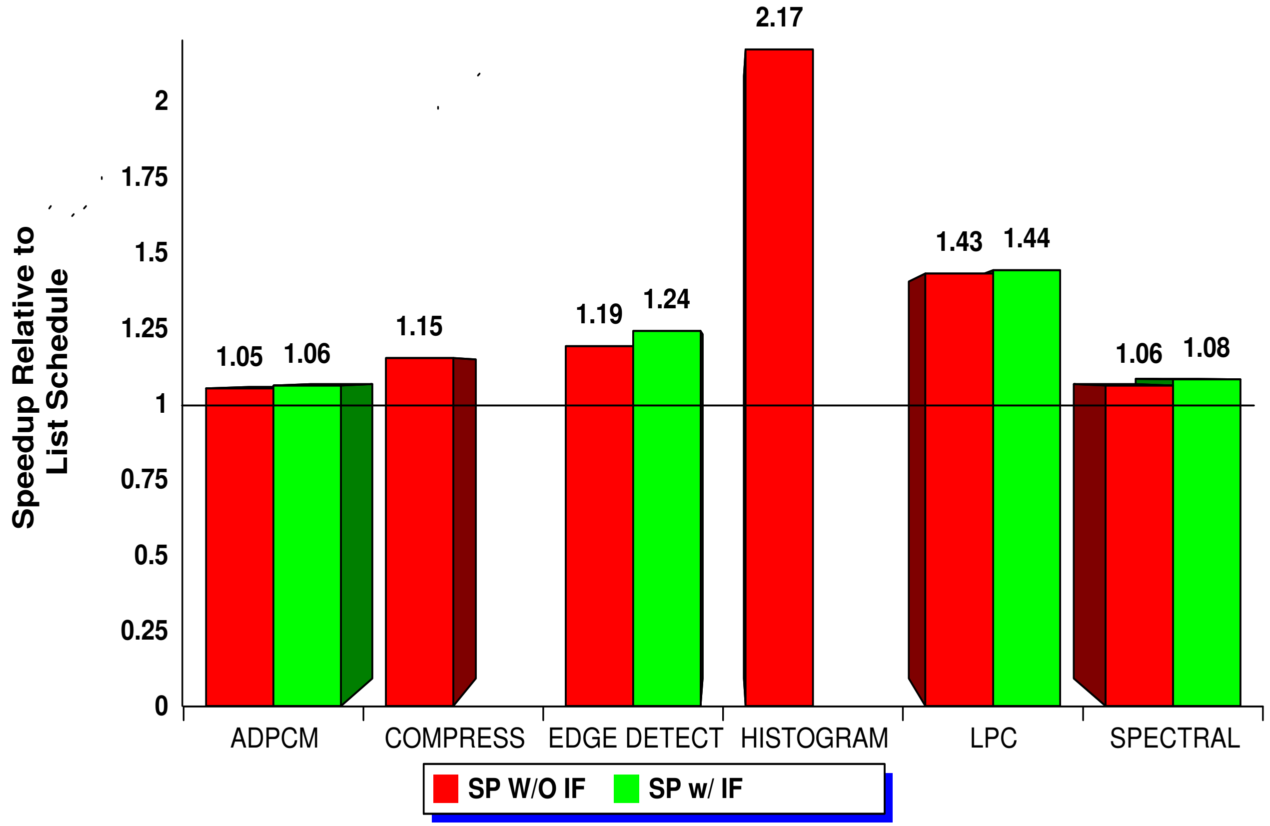
Benchmark Programs: DSP Applications

- ADPCM
- COMPRESS
- EDGE DETECT
- HISTOGRAM
- LPC
- SPECTRAL ESTIMATION

Machine Configuration:

- VLIW Architecture
- Three Register Sets:
  - Integer, Address, and Floating Point
- Five Functional Units:
  - Memory, Addressing, Integer, Floating Point, and Control

Estimated Execution Time Speedup



Summary

1. Software Pipelining using Hierarchical Reduction must assume worst case delay through an IF statement
   - **even** if one of the then or else clauses is missing

2. Software Pipelining using Enhanced Percolation Scheduling can maintain different initation intervals in loops with IF statements, **but**
   - does not perform well with limited resources
   - relies on homogeneous functional units and shared register file

3. This work presents a technique that can software pipeline loops containing IF statements
   - **well** (often with the minimum II on each path through the loop)
   - **with limited resource**s

   This same technique can also perform **as well as standard techniques** on loops without IF statements.