Optimizing Instruction-Level Parallelism ILP VLIW Architecture

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

September 2024 PMPH Lecture Slides

Course Organization

W	HARDWARE		SOFTWARE	LAB/CUDA
1	Trends Vector Machine	⟵	List HOM (Map-Reduce)	Intro & Simple Map Programming
2	In Order Processor	⟶ ⟵	VLIW Instr Scheduling	Scan & Reduce
3	Cache Coherence		Reasoning About Parallelism	Sparse Vect Matrix Mult
4	Interconnection Networks		Case Studies & Optimizations	Transpose & Matrix Matrix Mult
5	Memory Consistency		Optimising Locality	Sorting & Profiling & Mem Optimizations
6	OoO, Spec Processor		Thread-Level Speculation	Project Work

Three narative threads: the path to complex & good design:

- Design Space tradeoffs, constraints, common case, trends.
- Reasoning: from simple to complex, Applying Concepts.

Acknowledgments

This lecture presents selected Topics from Chapter 3 of the "Parallel Computer Organization and Design" book,
by Michel Dubois, Murali Annavaram and Per Stenstrom.

1. Instruction-Scheduling for ILP
   - Local Scheduling: Simple Code Motion
   - Cyclic Scheduling: Loop Unrolling
   - Cyclic Scheduling: Software Pipelining
2. Case Study: VLIW Architecture
   - Architectural View
   - Code Example: Loop Unrolling & Software Pipelining
   - Trace Scheduling
   - Predicated Instructions
   - Speculative Memory Disambiguation
   - VLIW Handling of Exceptions
3. Basic Blocks, CFG, Loops, Reducible CFG, Simple Optims
   - Basic Blocks Constructions and The Control-Flow Graph
   - Identifying Loops
   - Control-Flow-Graph Reducibility
   - Simple/Enabling Optimizations

Instruction Scheduling

- Dynamic by hardware: the ever-wider dispatch width $\Rightarrow$ more & more complex, slower, and power-hungry hardware.
- *Static* by compiler: instruction reordering aimed at minimizing the number of stalls, e.g., linear pipeline, superpipeline, superscalar, VLIW.

Instruction Scheduling

- Dynamic by hardware: the ever-wider dispatch width $\Rightarrow$ more & more complex, slower, and power-hungry hardware.
- *Static* by compiler: instruction reordering aimed at minimizing the number of stalls, e.g., linear pipeline, superpipeline, superscalar, VLIW.

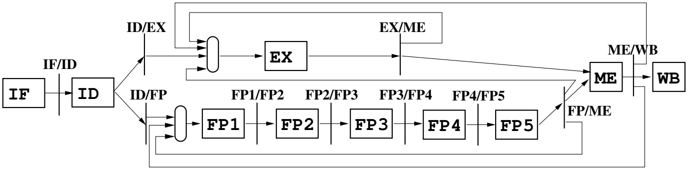
Compiler has a better structural view of the entire code, but has less reliable info @ execution time than hardware. Code Motion:

- *Local*, i.e., within a basic block (BB)?.
- *Global* (across BBs):
  - *cyclic*, meaning on loops?, e.g., loop unrolling, software pipelining.
  - *non-cyclic*, e.g., trace scheduling: schedule issued for the most likely program trace based on static prediction (profiling) + compensation code for all other possible traces.

Local Scheduling: Code Motion In a Basic Block

HL Code	Unoptimized

for(i=0;i<100;i++)
  A[i]=A[i]+B[i]
	<pre>//R3=100, R1/2=@A/B
	Loop: L.S F0, 0(R1) (1)
	      L.S F1, 0(R2) (1)
	      ADD.S F2, F1, F0 (2)
	      S.S F2, 0(R1) (5)
	      ADDI R1, R1, #4 (1)
	      ADDI R2, R2, #4 (1)
	      SUBI R3, R3, #1 (1)
	      BNEZ R3, Loop (3)</pre>



Local Scheduling: Code Motion In a Basic Block

HL Code	Unoptimized

`for(i=0;i<100;i++)`
`  A[i]=A[i]+B[i]`
	`//R3=100, R1/2=@A/B`
	`Loop: L.S    F0, 0(R1)    (1)`
	`      L.S    F1, 0(R2)    (1)`
	`      ADD.S  F2, F1, F0   (2)`
	`      S.S    F2, 0(R1)    (5)`
	`      ADDI   R1, R1, #4   (1)`
	`      ADDI   R2, R2, #4   (1)`
	`      SUBI   R3, R3, #1   (1)`
	`      BNEZ   R3, Loop     (3)`

In parenthesis is shown # of clocks taken by each instr in a single-issue static pipeline with separate integer an float execution:

- *branch*: 1 if untaken, 3 if taken (2 instr are flushed) ⇒ move it to the top of the loop.
  Delayed branches (ISA extension): put two instrs after the branch to be executed regardless of the branch.
- *other*: # cycles spent in ID stage due to data-hazards stalls,
- Unoptimized code #clocks = 15.

Local Scheduling: Code Motion In a Basic Block

HL Code	Unoptimized	After Code Motion

`for(i=0;i<100;i++)`
`  A[i]=A[i]+B[i]`
	`//R3=100, R1/2=@A/B`
	`Loop: L.S    F0, 0(R1)    (1)`
	`      L.S    F1, 0(R2)    (1)`
	`      ADD.S  F2, F1, F0   (2)`
	`      S.S    F2, 0(R1)    (5)`
	`      ADDI   R1, R1, #4   (1)`
	`      ADDI   R2, R2, #4   (1)`
	`      SUBI   R3, R3, #1   (1)`
	`      BNEZ   R3, Loop     (3)`
		`//R3=100, R1/2=@A/B`
		`Loop: L.S    F0, 0(R1)    (1)`
		`      L.S    F1, 0(R2)    (1)`
		`      SUBI   R3, R3, #1   (1)`
		`      ADD.S  F2, F1, F0   (1)`
		`      ADDI   R1, R1, #4   (1)`
		`      ADDI   R2, R2, #4   (1)`
		`      S.S    F2, -4(R1)   (3)`
		`      BNEZ   R3, Loop     (3)`

In parenthesis is shown # of clocks taken by each instr in a single-issue static pipeline with separate integer an float execution:

- `SUB.I` moved up after `L.S` to eliminate the stall caused by a load followed by a dependent use.
- S.S moved down far away from the ADD.S it depends on, but
- introduces WAR dependency, solved by adjusting `S.S`'s displ.

Unoptim#clocks = 15, Optimized#clocks = 12, Speedup = 1.25×.

Loop Unrolling

Basic Blocks have on average 4-5 instructions ⇒ limited benefits.

Loop Unrolling: schedules computed across several iterations.

Original	Unroll Twice	Rename Registers	Loads↑ Stores↓
	`for(i=0;i<100;i+=2)`	`for(i=0;i<100;i+=2)`	`for(i=0;i<100;i+=2)`
	`t1 = A[i]`	`t1 = A[i]`	`t1 = A[i]`
`for(i=0;i<100;i++)`	`t2 = B[i]`	`t2 = B[i]`	`t2 = B[i]`
`t1 = A[i]`	`t3 = t1+t2`	`t3 = t1+t2`	`t4 = A[i+1]`
`t2 = B[i]`	`A[i] = t3`	`A[i] = t3`	`t5 = B[i+1]`
`t3 = t1+t2`	`t1 = A[i+1]`	`t4 = A[i+1]`	`t3 = t1+t2`
`A[i] = t3`	`t2 = B[i+1]`	`t5 = B[i+1]`	`t6 = t4+t5`
	`t3 = t1+t2`	`t6 = t4+t5`	`A[i] = t3`
	`A[i+1] = t3`	`A[i+1] = t6`	`A[i+1] = t6`

Loop Unrolling

Basic Blocks have on average 4-5 instructions ⇒ limited benefits.

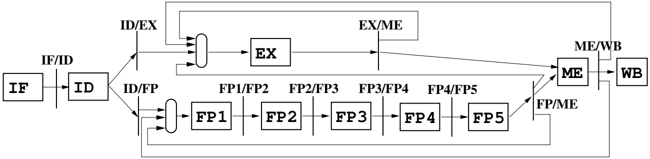
Loop Unrolling: schedules computed across several iterations.

Original	Unroll Twice	Rename Registers	Loads↑ Stores↓
	`for(i=0;i<100;i+=2)`	`for(i=0;i<100;i+=2)`	`for(i=0;i<100;i+=2)`
	`t1 = A[i]`	`t1 = A[i]`	`t1 = A[i]`
`for(i=0;i<100;i++)`	`t2 = B[i]`	`t2 = B[i]`	`t2 = B[i]`
`t1 = A[i]`	`t3 = t1+t2`	`t3 = t1+t2`	`t4 = A[i+1]`
`t2 = B[i]`	`A[i] = t3`	`A[i] = t3`	`t5 = B[i+1]`
`t3 = t1+t2`	`t1 = A[i+1]`	`t4 = A[i+1]`	`t3 = t1+t2`
`A[i] = t3`	`t2 = B[i+1]`	`t5 = B[i+1]`	`t6 = t4+t5`
	`t3 = t1+t2`	`t6 = t4+t5`	`A[i] = t3`
	`A[i+1] = t3`	`A[i+1] = t6`	`A[i+1] = t6`

With our example `for(i=0;i<100;i++) A[i]+=B[i]` in TAC form:

- unroll twice, but epilog is required if unknown loop count,
- code motion in the expanded body limited by WAW, WAR deps,
- which are fixed by **renaming** registers & adjusting displacements,
- finally, loads are moved up and stores down

Loop Unrolling on A Single-Issue Pipeline



TAC Optim Code

```
for(i=0;i<100;i+=2)
  t1 = A[i]
  t2 = B[i]
  t4 = A[i+1]
  t5 = B[i+1]
  t3 = t1+t2
  t6 = t4+t5
  A[i] = t3
  A[i+1] = t6
```

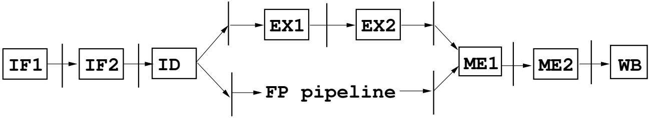
MIPS Optim Code | **# clocks in ID**

```
Loop: L.S    F0, 0(R1)      (1)
      L.S    F1, 0(R2)      (1)
      L.S    F3, 4(R1)      (1)
      L.S    F4, 4(R2)      (1)
      ADD.S  F2, F1, F0     (1)
      ADD.S  F5, F3, F4     (1)
      SUBI   R3, R3, #2     (1)
      ADDI   R1, R1, #8     (1)
      ADDI   R2, R2, #8     (1)
      S.S    F2, -8(R1)     (1)
      S.S    F5, -4(R1)     (1)
      BNEZ   R3, Loop       (3)
```

clocks for 2 iters: 14 $\Rightarrow$ Speedup $= 15/7 = 2.14\times$

Scratch Page

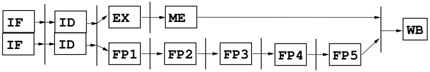
Loop Unrolling on A 5-stage-FP-Unit Super-Pipeline



TAC Optim Code	MIPS Optim Code	# clocks in ID
	Loop: L.S F0, 0(R1)	(1)
for(i=0;i<100;i+=2)	L.S F1, 0(R2)	(1)
t1 = A[i]	L.S F3, 4(R1)	(1)
t2 = B[i]	L.S F4, 4(R2)	(1)
t4 = A[i+1]	ADD.S F2, F1, F0	(2)
t5 = B[i+1]	ADD.S F5, F3, F4	(2)
t3 = t1+t2	SUBI R3, R3, #2	(1)
t6 = t4+t5	ADDI R1, R1, #8	(1)
A[i] = t3	ADDI R2, R2, #8	(1)
A[i+1] = t6	S.S F2, -8(R1)	(1)
	S.S F5, -4(R1)	(1)
	BNEZ R3, Loop	(4)

17 clocks for 2 iters, or 8.5 clocks per iteration, but the clock rate is twice as fast $\Rightarrow$ Speedup $= 2*15/8.5 = 3.52\times$.

Loop Unrolling on A Super-Scalar

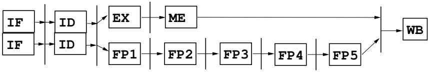


Modest speedup 2.14× because fraction of FP instrs is small:

TAC Optim Code	MIPS Optim Code		# clocks in ID
`for(i=0;i<100;i+=2)`	`Loop: L.S F0, 0(R1)`		(1)
`t1 = A[i]`	`L.S F1, 0(R2)`		(1)
`t2 = B[i]`	`L.S F3, 4(R1)`		(1)
`t4 = A[i+1]`	`L.S F4, 4(R2)`		(1)
`t5 = B[i+1]`	`SUBI R3, R3, #2`	`ADD.S F2, F1, F0`	(1)
`t3 = t1+t2`	`ADDI R1, R1, #8`	`ADD.S F5, F3, F4`	(1)
`t6 = t4+t5`	`ADDI R2, R2, #8`		(1)
`A[i] = t3`	`S.S F2, -8(R1)`		(3)
`A[i+1] = t6`	`S.S F5, -4(R1)`		(1)
	`BNEZ R3, Loop`		(3)

Drawbacks of Loop Unrolling:

Loop Unrolling on A Super-Scalar



Modest speedup 2.14× because fraction of FP instrs is small:

TAC Optim Code	MIPS Optim Code		# clocks in ID
	`Loop: L.S F0, 0(R1)`		(1)
`for(i=0;i<100;i+=2)`	`L.S F1, 0(R2)`		(1)
`t1 = A[i]`	`L.S F3, 4(R1)`		(1)
`t2 = B[i]`	`L.S F4, 4(R2)`		(1)
`t4 = A[i+1]`	`SUBI R3, R3, #2`	`ADD.S F2, F1, F0`	(1)
`t5 = B[i+1]`	`ADDI R1, R1, #8`	`ADD.S F5, F3, F4`	(1)
`t3 = t1+t2`	`ADDI R2, R2, #8`		(1)
`t6 = t4+t5`	`S.S F2, -8(R1)`		(3)
`A[i] = t3`	`S.S F5, -4(R1)`		(1)
`A[i+1] = t6`	`BNEZ R3, Loop`		(3)

Drawbacks of Loop Unrolling:

- Ineffective for loop-carried dependencies (restrict code motion),
- Consumes many addressable registers (since renaming is critical),
- High code expansion of the loop body.

Software Pipelining: Intuition

Original loop (`Orig_It1-4`) transformed into another loop (`P_It1-3`) which *pipelines* the original's dependent instrs across multiple iters:

	Orig_It1	Orig_It2	Orig_It3	Orig_It4
Prolog	L.S F0, 0(R1) L.S F1, 0(R2) ADD.S F2,F1,F0	-	-	-
P_It1	S.S F2, 0(R1)	L.S F0, 0(R1) L.S F1, 0(R2) ADD.S F2,F1,F0	-	-
P_It2		S.S F2, 0(R1)	L.S F0, 0(R1) L.S F1, 0(R2) ADD.S F2,F1,F0	-
P_It3			S.S F2, 0(R1)	L.S F0, 0(R1) L.S F1, 0(R2) ADD.S F2,F1,F0
Epilog				S.S F2, 0(R1)

Transformed code: `prolog; pipelined_loop; epilog;`
The order in which instrs are executed is the same row and column-wise.

Software Pipelining Schedule

Local Optimized Code

```
for(i=0;i<100;i++)
  A[i]=A[i]+B[i]

Loop: L.S    F0, 0(R1)    (1)
      L.S    F1, 0(R2)    (1)
      SUBI   R3, R3, #1   (1)
      ADD.S  F2, F1, F0   (1)
      ADDI   R1, R1, #4   (1)
      ADDI   R2, R2, #4   (1)
      S.S    F2, -4(R1)   (3)
      BNEZ   R3, Loop     (3)
```

- Avoids stalls without using more registers.
- Pipelined loop same size as original loop.
- Effective even when loop carries dependencies.

Pipelined Schedule:

```
Prolog: L.S    F0,0(R1)
        L.S    F1,0(R2)
        SUBI   R3,R3,#1
        ADD.S  F2,F1,F0
        ADDI   R1,R1,#4
        ADDI   R2,R2,#4
Loop:   S.S    F2,-4(R1)  (1)
        L.S    F0,0(R1)   (1)
        L.S    F1,0(R2)   (1)
        SUBI   R3,R3,#1   (1)
        ADD.S  F2,F1,F0   (1)
        ADDI   R1,R1,#4   (1)
        ADDI   R2,R2,#4   (1)
        BNEZ   R3, Loop   (3)
Epilog: S.S    F2,-4(R1)
```

Initiation Interval: after how many clocks can a new (orig) iter start (concurrently with previous ones), such that latencies are respected? In this case 3 cycles: moving S.S across exploits BNEZ latency.

Software Pipelining on Independent Loops

The pipelined-loop iteration contains the same instructions but each corresponds to a different iteration of the original loop.

- Assume Intra-Iteration Producer-Consumer Relation `A(i)`⇒`B(i)`⇒ ... ⇒`F(i)`
- NO Cross-Iteration dependencies (`A(i+1)` does not depend on `B(i)`),

Original	Pipelined	Original/Pipeline=Columns/Rows

<pre>for(i=1;i<=N;i++)
  A(i) // (3 cycle)
  B(i) // (3)
  C(i) // (3)
  D(i) // (12)
  E(i) // (3)
  F(i) // (3)
end</pre>
	<pre>prologue
	for(i=1;i<=N-6;i++)
	  A(i+6)
	  B(i+5)
	  C(i+4)
	  D(i+2) //skip i+3
	  E(i+1)
	  F(i)
	end
	epilogue</pre>	

Software Pipelining on Independent Loops

The pipelined-loop iteration contains the same instructions but each corresponds to a different iteration of the original loop.

- Assume Intra-Iteration Producer-Consumer Relation `A(i)`⇒`B(i)`⇒ ... ⇒`F(i)`
- NO Cross-Iteration dependencies (`A(i+1)` does not depend on `B(i)`),

Original	Pipelined	Original/Pipeline=Columns/Rows
	`prologue`	`A(1)                                        // prolog begins`
`for(i=1;i<=N;i++)`	`for(i=1;i<=N-6;i++)`	`A(2), B(1)`
`  A(i) // (3 cycle)`	`  A(i+6)`	`A(3), B(2), C(1)`
`  B(i) // (3)`	`  B(i+5)`	`A(4), B(3), C(2)`
`  C(i) // (3)`	`  C(i+4)`	`A(5), B(4), C(3),     , D(1)`
`  D(i) // (12)`	`  D(i+2) //skip i+3`	`A(6), B(5), C(4),     , D(2), E(1) // loop begins`
`  E(i) // (3)`	`  E(i+1)`	`A(7), B(6), C(5),     , D(3), E(2), F(1)   //It1`
`  F(i) // (3)`	`  F(i)`	`A(8), B(7), C(6),     , D(4), E(3), F(2)   //It2`
`end`	`end`	`A(9), B(8), C(7),     , D(5), E(4), F(3)   //It3`
	`epilogue`	`.....                                       // epilogue begins`

- Latencies Resolved: 7 instrs between A(7) and B(7), 12 between C(5) and D(5)

Software Pipelining on Independent Loops

The pipelined-loop iteration contains the same instructions but each corresponds to a different iteration of the original loop.

- Assume Intra-Iteration Producer-Consumer Relation `A(i)`⇒`B(i)`⇒ ... ⇒`F(i)`
- NO Cross-Iteration dependencies (`A(i+1)` does not depend on `B(i)`),

Original	Pipelined	Original/Pipeline=Columns/Rows
	prologue	A(1) // prolog begins
for(i=1;i<=N;i++)	for(i=1;i<=N-6;i++)	A(2), B(1)
A(i) // (3 cycle)	A(i+6)	A(3), B(2), C(1)
B(i) // (3)	B(i+5)	A(4), B(3), C(2)
C(i) // (3)	C(i+4)	A(5), B(4), C(3), , D(1)
D(i) // (12)	D(i+2) //skip i+3	A(6), B(5), C(4), , D(2), E(1) // loop begins
E(i) // (3)	E(i+1)	A(7), B(6), C(5), , D(3), E(2), F(1) //It1
F(i) // (3)	F(i)	A(8), B(7), C(6), , D(4), E(3), F(2) //It2
end	end	A(9), B(8), C(7), , D(5), E(4), F(3) //It3
	epilogue	..... // epilogue begins

- Latencies Resolved: 7 instrs between A(7) and B(7), 12 between C(5) and D(5)
- Loop Unrolling needs a factor 12× ⇒ register & cache pressure,
- Soft Pipelining may require hardware support (rotating registers bank) to fix WAR deps, and needs prolog and epilog code.
- All stalls have been eliminated & Each instruction `A(i)...F(i)` takes one clock and
- Apply first *loop unrolling*, and then *softw pipelining* (because the latter introduces loop-carried dependencies!)

Scratch Page

1. Instruction-Scheduling for ILP
   - Local Scheduling: Simple Code Motion
   - Cyclic Scheduling: Loop Unrolling
   - Cyclic Scheduling: Software Pipelining
2. Case Study: VLIW Architecture
   - Architectural View
   - Code Example: Loop Unrolling & Software Pipelining
   - Trace Scheduling
   - Predicated Instructions
   - Speculative Memory Disambiguation
   - VLIW Handling of Exceptions
3. Basic Blocks, CFG, Loops, Reducible CFG, Simple Optims
   - Basic Blocks Constructions and The Control-Flow Graph
   - Identifying Loops
   - Control-Flow-Graph Reducibility
   - Simple/Enabling Optimizations

VLIW: Birds Eye View

Very Long Instruction Word (VLIW): statically scheduled microarchitectures, in which each (very) long instruction contains several MIPS instrs, called "OPS". The OPS of the current LIW:

- are fetched at once, are decoded in parallel and
- are applied each to a different pipeline (with optimized decoding)
- correspond to independent instructions

Minimizes hwd complexity and power & enable superscalar arch with very-wide fetch and dispatch bandwidth. All hazards solved by compiler; if enough exploitable ILP $\Rightarrow$ no stalls:

VLIW: Birds Eye View

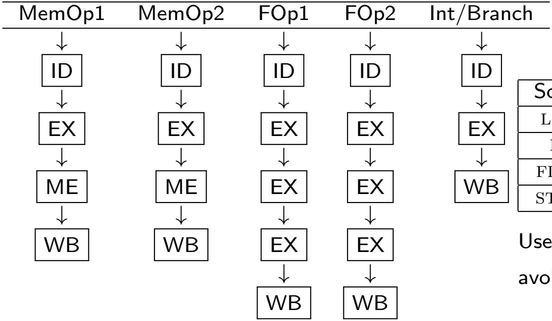
Very Long Instruction Word (VLIW): statically scheduled microarchitectures, in which each (very) long instruction contains several MIPS instrs, called "OPS". The OPS of the current LIW:

- are fetched at once, are decoded in parallel and
- are applied each to a different pipeline (with optimized decoding)
- correspond to independent instructions

Minimizes hwd complexity and power & enable superscalar arch with very-wide fetch and dispatch bandwidth. All hazards solved by compiler; if enough exploitable ILP $\Rightarrow$ no stalls:

- RAW on registers: solved by inserting enough instrs (cycles) between the source and destination of the dependency,
- WAR and WAW solved by renaming registers,
- structural & control hazards are avoided (by code scheduling),
- exceptions & indirect mem access: speculative mechanisms relying on patch-up code to cancel unwanted execution results.

VLIW Architecture

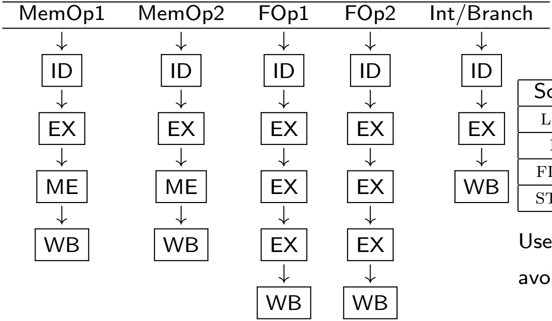


Operation Latencies For Various Forwarding (F) Assumptions:

Source	Dest	NoF	RegF	FullF
LOAD	any	3	2	1
INT	any	2	1	0
FLOAT	any	4	3	2
STORE	LOAD	0	0	0

Uses delayed branches by 2 instrs to avoid flushing (simplest solution)!

VLIW Architecture



Operation Latencies For Various Forwarding (F) Assumptions:

Source	Dest	NoF	RegF	FullF
LOAD	any	3	2	1
INT	any	2	1	0
FLOAT	any	4	3	2
STORE	LOAD	0	0	0

Uses delayed branches by 2 instrs to avoid flushing (simplest solution)!

Compiler needs to know the operation latency of each instruction:

- without forwarding (NoF) instr cannot be scheduled before its parent (source) **has passed** the WB stage.
- with register forwarding (RegF) a value written in the WB stage is forwarded **in the same cycle** to the (child) ID stage.
- with full forwarding (FullF) instruction can be scheduled **as soon as** the result is available (in EX or ME).

VLIW: Code Example

```
for(i=1000;i>0;i--)
  x[i] = x[i] + s
```

```
Loop: L.D    F0, 0(R1)
      ADD.D  F4, F0, F2
      S.D    F4, 0(R1)
      SUBI   R1, R1, #8
      BNE    R1, R2, Loop
```

Clock	MemOp1	MemOp2	FOp1	FOp2	Int/Branch
1	L.D F0, 0(R1)	NOOP	NOOP	NOOP	NOOP
2	NOOP	NOOP	NOOP	NOOP	SUBI R1,R1,#8
3	NOOP	NOOP	ADD.D F4,F0,F2	NOOP	NOOP
4	NOOP	NOOP	NOOP	NOOP	BNE R1,R2,Loop
5	NOOP	NOOP	NOOP	NOOP	NOOP
6	S.D F4, 8(R1)	NOOP	NOOP	NOOP	NOOP

VLIW: Code Example

```
for(i=1000;i>0;i--)          Loop: L.D    F0, 0(R1)
  x[i] = x[i] + s                  ADD.D  F4, F0, F2
                                   S.D    F4, 0(R1)
                                   SUBI   R1, R1, #8
                                   BNE    R1, R2, Loop
```

Clock	MemOp1	MemOp2	FOp1	FOp2	Int/Branch
1	L.D F0, 0(R1)	NOOP	NOOP	NOOP	NOOP
2	NOOP	NOOP	NOOP	NOOP	SUBI R1,R1,#8
3	NOOP	NOOP	ADD.D F4,F0,F2	NOOP	NOOP
4	NOOP	NOOP	NOOP	NOOP	BNE R1,R2,Loop
5	NOOP	NOOP	NOOP	NOOP	NOOP
6	S.D F4, 8(R1)	NOOP	NOOP	NOOP	NOOP

- one instr needs to be inserted between L.D (load) and the dependent ADD.D
- two instructions need to be inserted between ADD.D and the dependent S.D (store).
- branches are delayed, i.e., two instructions inserted after BNE which semantically execute before BNE.
- without cyclic optimizations $\Rightarrow$ 6 clocks/iteration (Terrible).

Schedule With Loop Unrolling

```
for(i=1000;i>0;i--)        Loop: L.D    F0, 0(R1)
  x[i] = x[i] + s                ADD.D F4, F0, F2
                                 S.D    F4, 0(R1)
                                 SUBI  R1, R1, #8
                                 BNE    R1, R2, Loop
```

Clock	MemOp1	MemOp2	FOp1	FOp2	Int/Branch
1	L.D F0, 0(R1)	L.D F6, - 8(R1)	NOOP	NOOP	NOOP
2	L.D F10,-16(R1)	L.D F14,-24(R1)	NOOP	NOOP	NOOP
3	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4, F0, F2	ADD.D F8, F6, F2	NOOP
4	L.D F26,-48(R1)	NOOP	ADD.D F12,F10,F2	ADD.D F16,F14,F2	NOOP
5	NOOP	NOOP	ADD.D F20,F18,F2	ADD.D F24,F22,F2	NOOP
6	S.D F4, 0(R1)	S.D F8, -8(R1)	ADD.D F28,F26,F2	NOOP	SUBI R1,R1,#56
7	S.D F12,40(R1)	S.D F16,32(R1)	NOOP	NOOP	DBNE R1,R2,Loop
8	S.D F20,24(R1)	S.D F24,16(R1)	NOOP	NOOP	NOOP
9	S.D F28, 8(R1)	NOOP	NOOP	NOOP	NOOP

- original loop was unrolled 7 times,
- register renaming and adjusting load/store displacements fixed WAR and WAW dependencies,
- code size of the loop increases $\sim 7\times$, & high register pressure,
- but schedule executes 7 original-loop iterations in 9 clocks, while respecting all dependencies. Speedup $= 6*7 \;/\; 9 = 4.67\times$

Schedule With Software Pipelining

```
for(i=1000;i>0;i--)        Loop: L.D   F0, 0(R1)     (o1)   // we only consider
  x[i] = x[i] + s               ADD.D F4, F0, F2    (o2)   // the problematic
                                S.D   F4, 0(R1)     (o3)   // accesses o1, o2 and o3
```

	O_It1	O_It2	O_It3	O_It4	O_It5	O_It6	O_It7
P_It1	o1						
P_It2	-	o1					
P_It3	o2	-	o1				
P_It4	-	o2	-	o1			
P_It5	-	-	o2	-	o1		
P_It6	o3	-	-	o2	-	o1	
P_It7		o3	-	-	o2	-	o1
P_It8			o3	-	-	o2	-
P_It9				o3	-	-	o2
P_It10					o3	-	-
P_It11						o3	-
P_It12							o3

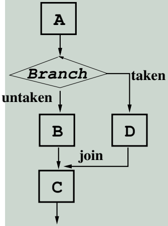
Clock	MemOp1	MemOp2	FOp1	FOp2	Int/Branch
1	S.D RR0, 40(R1)	L.D RR6, 0(R1)	ADD.D RR3,RR4,F2	NOOP	NOOP

- Initiation Interval = 1 resolves all dependencies, according to schedule:
  - the STORE of each VLIW instr is 3 instrs (iters) behind the ADD,
  - the ADD of each VLIW instr in 2 instrs (iters) behind the LOAD.
- Registers `F4` and `F2` need to be replaced with rotating registers to solve WARs.
- Loop body is 1 VLIW instr, but we have not considered SUBI and BNE:
  - they cannot both fit in the instr because only one slot for INT/branch
  - loop unrolling + soft pipeling gets the best benefit.

Non-Cyclic, Trace Scheduling

Effective when a branch is highly predictable statically (profiling).

Structure



Original Code

```
        LW   R4, 0(R1)
        ADDI R6,R4,1
/* End Block A */
        BEQ  R5,R4,LAB
        LW   R6, 0(R2)
/* End Block B */
LAB:
/* End Empty Block D */
        SW   R6, 0(R2)
/* End Block C */
```

Trace Scheduled

```
        LW   R4, 0(R1)
        ADDI R6,R4,1
        LW   R6, 0(R2)
/* End Block A;B */
        BEQ  R5,R4,LABCC
/*CompensationCode if taken*/
LAB:
/* End Empty Block D */
        SW   R6, 0(R2)
/* End Block C */
...
LABCC:ADDI R6, R4, 1
      J    LAB
/*EndCompensationCode*/
```

Optim Code

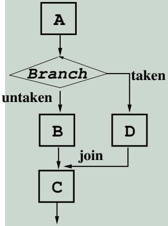
```
        LW   R4, 0(R1)
        LW   R6, 0(R2)
/* End Block A;B */
        BEQ  R5,R4,LABCC
LAB:    SW   R6, 0(R2)
/* End Block C */
...
LABCC:   ADDI R6, R4, 1
         J    LAB
/*EndCompensationCode*/
```

- Assume A→B→C is the common path,

Non-Cyclic, Trace Scheduling

Effective when a branch is highly predictable statically (profiling).

Structure



Original Code

```
        LW   R4, 0(R1)
        ADDI R6,R4,1
/* End Block A */
        BEQ  R5,R4,LAB
        LW   R6, 0(R2)
/* End Block B */
LAB:
/* End Empty Block D */
        SW   R6, 0(R2)
/* End Block C */
```

Trace Scheduled

```
      LW   R4, 0(R1)
      ADDI R6,R4,1
      LW   R6, 0(R2)
/* End Block A;B */
      BEQ  R5,R4,LABCC
/*CompensationCode if taken*/
LAB:
/* End Empty Block D */
      SW   R6, 0(R2)
/* End Block C */
...
LABCC:ADDI R6, R4, 1
      J    LAB
/*EndCompensationCode*/
```

Optim Code

```
      LW   R4, 0(R1)
      LW   R6, 0(R2)
/* End Block A;B */
      BEQ  R5,R4,LABCC
LAB:  SW   R6, 0(R2)
/* End Block C */
...
LABCC:   ADDI R6, R4, 1
         J    LAB
/*EndCompensationCode*/
```

- Assume A→B→C is the common path,
- Basic Blocks A and B are speculatively merged;
- branch target is modified to jump to compensation code, from where
- it jumps back to the beginning of D.
- Process can be repeated aggressively for branches in B.
- Optimized Code: the two L.D can be scheduled in the same VLIW instruction.

Predicated Instructions, e.g., IA-64

Effective when branches:

- are unbiased (50-50) or hard to predict statically, AND
- guard small basic blocks

```
CLWZ  R1, 0(R2), R3   /* Load Mem[0(R2)] in R1 if R3 is     0 */
CLWNZ R1, 0(R2), R3   /* Load Mem[0(R2)] in R1 if R3 is NOT 0 */
```

Original Code	Predicated	Predicated Optim
`LW   R4, 0(R1)`	`LW    R4, 0(R1)`	`LW     R4, 0(R1)`
`ADDI R6, R4, #1`	`ADDI  R6, R4, #1`	`SUB    R3, R5, R4`
`BEQ  R5, R4, LAB`	`SUB   R3, R5, R4`	`CADDIZ R6, R4, #1, R3`
`LW   R6, 0(R2)`	`CLWNZ R6, 0(R2), R3`	`CLWNZ  R6, 0(R2), R3`
`LAB: SW   R6, 0(R1)`	`SW    R6, 0(R1)`	`SW     R6, 0(R1)`

- BEQ and LW translated as a predicated load, i.e., SUB and CLWNZ.

Predicated Instructions, e.g., IA-64

Effective when branches:

- are unbiased (50-50) or hard to predict statically, AND
- guard small basic blocks

```
CLWZ  R1, 0(R2), R3   /* Load Mem[0(R2)] in R1 if R3 is     0 */
CLWNZ R1, 0(R2), R3   /* Load Mem[0(R2)] in R1 if R3 is NOT 0 */
```

Original Code	Predicated	Predicated Optim
`LW   R4, 0(R1)`	`LW    R4, 0(R1)`	`LW     R4, 0(R1)`
`ADDI R6, R4, #1`	`ADDI  R6, R4, #1`	`SUB    R3, R5, R4`
`BEQ  R5, R4, LAB`	`SUB   R3, R5, R4`	`CADDIZ R6, R4, #1, R3`
`LW   R6, 0(R2)`	`CLWNZ R6, 0(R2), R3`	`CLWNZ  R6, 0(R2), R3`
`LAB: SW   R6, 0(R1)`	`SW    R6, 0(R1)`	`SW     R6, 0(R1)`

- BEQ and LW translated as a predicated load, i.e., SUB and CLWNZ.
- Since R6 is overwritten when the conditional load holds, ADDI is translated to its predicated form CADDIZ which holds exactly when CLWNZ does not.
- Now CLWNZ and CADDIZ can be scheduled in the same VLIW instruction.
- Predicated instructions are not allowed to change the architectural state or to raise an exception unless their condition holds.
- Predicated instructions transform control dependencies, which are a barrier to code motion, into data dependencies on registers, which are easier to handle.

Speculative Memory Disambiguation

Problem: A *load* cannot be moved across a *store* if it cannot be established statically that the two memory addresses are different.

Original Code	Incorrect Code	Correct Code

`I1`
`SW  R1, 0(R3)`
`I2`
`LW  R4, 0(R2)`
`ADD R5, R4, R4`
	`LW  R4, 0(R2)`
	`ADD R5, R4, R4`
	`I1`
	`SW  R1, 0(R3)`
	`I2`
		`LW.a R4, 0(R2)`
		`ADD  R5, R4, R4`
		`I1`
		`SW   R1, 0(R3)`
		`I2`
		`CHECK.a 0(R2), repair`

- The hoisted *load* is marked speculative via instruction `LW.a`, and
- a guardian is inserted at the position from where it was moved.

Speculative Memory Disambiguation

Problem: A *load* cannot be moved across a *store* if it cannot be established statically that the two memory addresses are different.

Original Code	Incorrect Code	Correct Code

I1
SW R1, 0(R3)
I2
LW R4, 0(R2)
ADD R5, R4, R4
	LW R4, 0(R2)
	ADD R5, R4, R4
	I1
	SW R1, 0(R3)
	I2
		LW.a R4, 0(R2)
		ADD R5, R4, R4
		I1
		SW R1, 0(R3)
		I2
		CHECK.a 0(R2), repair

- The hoisted *load* is marked speculative via instruction `LW.a`, and
- a guardian is inserted at the position from where it was moved.
- `LW.a` records its address in a small, fully associative table;
- Every store looks up and removes its address from this table;
- If the guardian instruction CHECK.a does not find the address in the table then a violation has occurred and a repair handle is launched. In our case the latter re-executes the speculatively-hoisted instructions, i.e., `LW` and `ADD`.
- Note that a *store* cannot be moved across a *load* because this miss-speculation cannot be easily repaired (memory is written). Rather move the *store* downwards.

VLIW Handling of Exceptions

Cache misses freeze the pipeline $\Rightarrow$ compiler schedule is preserved.

Exceptions from cyclic scheduling NOT a problem because all instrs of the transformed program are also executed in the original program.

Exception from non-cyclic scheduling are problematic because speculatively executed instructions may trigger exceptions that would not arise in the original code, e.g., array out of bounds.

- User-invisible exceptions are harmless and can always be taken, e.g., page faults caused by speculative memory disambiguation.
- Unwanted User-Visible Exceptions, e.g., termination due to index out of bounds caused by a speculative load, must be repressed if speculation does not hold.

A common solution is **deferred exceptions**: whenever a speculative instruction triggers a user-visible exception, the destination register is poisoned (poisoned bit set & exception report).

VLIW Handling of Exceptions (Cont.)

Deferred Exceptions via Poisoning:

- When another speculative instruction reads a poisoned value, poison is propagated to its destination register as well, without raising exception.
- When an instruction with no exception writes a poisoned register, the poison bit is reset ⇒ the poisoned value was useless.
- Exception is taken when a non-speculative instr uses a poisoned value (as operand).

Orignal	Trace Scheduled	Deferred Exception
	`LW R4, 0(R1)`	`LW R4, 0(R1)`
	`ADDI R6,R4,1`	`ADDI R6,R4,1`
`LW R4, 0(R1)`	`LW R6, 0(R2)`	`sLW R6, 0(R2)`
`ADDI R6,R4,1`	`BEQ R5,R4,LABCC`	`BEQ R5,R4,LABCC`
`BEQ R5,R4,LAB`	`LAB: SW R6, 0(R2)`	`LAB: SW R6, 0(R2)`
`LW R6, 0(R2)`	`...`	`...`
`LAB: SW R6, 0(R2)`	`LABCC:ADDI R6, R4, 1`	`LABCC:ADDI R6, R4, 1`
	`J LAB`	`J LAB`
	`/*Compensation Code*/`	`/*Compensation Code*/`

- When a user-visible exception occurs on sLW ⇒ R6 is poisoned:
- If branch is taken

VLIW Handling of Exceptions (Cont.)

Deferred Exceptions via Poisoning:

- When another speculative instruction reads a poisoned value, poison is propagated to its destination register as well, without raising exception.
- When an instruction with no exception writes a poisoned register, the poison bit is reset ⇒ the poisoned value was useless.
- Exception is taken when a non-speculative instr uses a poisoned value (as operand).

Orignal	Trace Scheduled	Deferred Exception
	`LW R4, 0(R1)`	`LW R4, 0(R1)`
	`ADDI R6,R4,1`	`ADDI R6,R4,1`
`LW R4, 0(R1)`	`LW R6, 0(R2)`	`sLW R6, 0(R2)`
`ADDI R6,R4,1`	`BEQ R5,R4,LABCC`	`BEQ R5,R4,LABCC`
`BEQ R5,R4,LAB`	`LAB: SW R6, 0(R2)`	`LAB: SW R6, 0(R2)`
`LW R6, 0(R2)`	`...`	`...`
`LAB: SW R6, 0(R2)`	`LABCC:ADDI R6, R4, 1`	`LABCC:ADDI R6, R4, 1`
	`J LAB`	`J LAB`
	`/*Compensation Code*/`	`/*Compensation Code*/`

- When a user-visible exception occurs on sLW ⇒ R6 is poisoned:
- If branch is takenpoison goes away because R6 is re-written (LABCC: ADDI). This is Correct Behavior: branch was miss-speculated & the load should not have occurred.
- If branch is not taken, then the exception is raised by non-speculative SW which uses R6. Again, Correct Behavior: the speculation was correct and resulted in exception.

Scratch Page

1. Instruction-Scheduling for ILP
   - Local Scheduling: Simple Code Motion
   - Cyclic Scheduling: Loop Unrolling
   - Cyclic Scheduling: Software Pipelining
2. Case Study: VLIW Architecture
   - Architectural View
   - Code Example: Loop Unrolling & Software Pipelining
   - Trace Scheduling
   - Predicated Instructions
   - Speculative Memory Disambiguation
   - VLIW Handling of Exceptions
3. Basic Blocks, CFG, Loops, Reducible CFG, Simple Optims
   - Basic Blocks Constructions and The Control-Flow Graph
   - Identifying Loops
   - Control-Flow-Graph Reducibility
   - Simple/Enabling Optimizations

Basic Blocks & Control Flow Graph

We use Three Address Code (TAC) rather than MIPS:

Basic Blocks & Control Flow Graph

We use Three Address Code (TAC) rather than MIPS:

- instruction has at most 2 operands & one result, e.g., `s := s+i`
- Jump labels, `goto`, conditional jump (`if`), e.g., `if i=0 goto L2`
- Memory load/store, function call/return (not used here).

Three Address Code Example

```
    i := 20
    s := 0
L1: if i=0 goto L2
    s := s + i
    i := i - 1
    goto L1
L2: ...
```

Interm-lang optimizations, e.g., TAC, portable to various backends,
... but Can we rebuild the control flow structure from TAC?

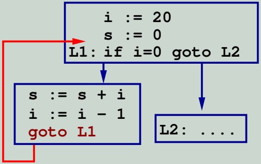
Basic Blocks (BB)

Basic Block: intuitively the maximal sequence of (consecutive) TAC instructions in which flow can enter/exit only via the first/last instr.

Code Example

```
    i := 20
    s := 0
L1: if i=0 goto L2
    s := s + i
    i := i - 1
    goto L1
L2: ...
```

Incorrect Basic Block



Identifying Basic Blocks (BB)

- Statements that start a basic block (BB):
  - first statement of any function
  - any labeled statement that is the target of a branch
  - any statement following a branch (conditional or unconditional)
- for each statement starting a BB, the BB consists of all stmts up to, but excluding, the start of a BB or the end of the program!

Identifying Basic Blocks (BB)

- Statements that start a basic block (BB):
  - first statement of any function
  - any labeled statement that is the target of a branch
  - any statement following a branch (conditional or unconditional)
- for each statement starting a BB, the BB consists of all stmts up to, but excluding, the start of a BB or the end of the program!

Code Example

```
    i := 20
    s := 0
L1: if i=0 goto L2
    s := s + i
    i := i - 1
    goto L1
L2: ...
```

Basic Blocks

B1
```
i := 20
s := 0
```

B2
```
L1: if i=0 goto L2
```

B3
```
s := s + i
i := i - 1
goto L1
```

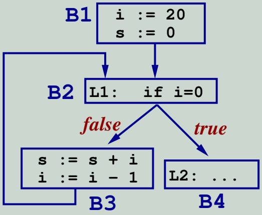
B4
```
L2: ...
```

Building The Control-Flow Graph

Place an arrow from node *A* to node *B* if it is possible for control to "flow" from *A* to *B* (and remove gotos).

Code Example and its Control-Flow Graph (CFG)

```
    i := 20
    s := 0
L1: if i=0 goto L2
    s := s + i
    i := i - 1
    goto L1
L2: ...
```



Identifying Loops, Preliminaries

Motivation: loops is where most of the time is spent!

Identifying Loops, Preliminaries

Motivation: loops is where most of the time is spent!

Definition: A loop, L, is a subgraph of the CFG such that:

- all nodes of the loop are strongly connected, i.e., the loop contains a path between any two loop nodes.
- the loop has an unique entry point, named header, such that
- the only way to reach a loop node is through the entry point.

A loop that contains no other loop is called an *innermost loop*.

Identifying Loops, Preliminaries

Motivation: loops is where most of the time is spent!

Definition: A loop, L, is a subgraph of the CFG such that:

- all nodes of the loop are strongly connected, i.e., the loop contains a path between any two loop nodes.
- the loop has an unique entry point, named header, such that
- the only way to reach a loop node is through the entry point.

A loop that contains no other loop is called an *innermost loop*.

Dominator Definition: a node p dominates node q if all paths from the start of the program to q go through p.

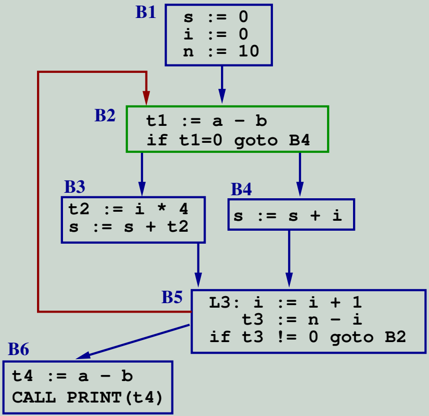
Identifying loops requires finding their "back edges":

- edges in the program in which the destination node dominates the source node.
- a loop must have an unique header, and one or more backedges.
- header dominates all blocks in the loop, otherwise not unique.

Example of a Loop CFG

Example: Finding Basic Blocks and The Control-Flow Graph (CFG)

```
    s := 0
    i := 0
    n := 10
L1:
    t1 := a - b
    if t1 = 0 goto L2
    t2 := i * 4
    s := s + t2
    goto L3
L2: s := s + i
L3: i := i + 1
    t3 := n - i
    if t3 != 0 goto L1
    t4 := a - b
    CALL PRINT(t4)
```



Identifying Loops

Algorithm for Dominators. $D(n)$ is the set of dominators of block n.

Input: CFG with node set N, initial node n_0. **Output**: $D(n), \forall n \in N$

$D(n_0) := \{n_0\}$
for $n \in N - \{n_0\}$ do $D(n) := N$

while changes to any $D(n)$ occur do
  for $n \in N - \{n_0\}$ do
    $D(n) := \{n\} \cup (\cap_{p \in pred(n)} D(p))$

Identifying Loops

Algorithm for Dominators. $D(n)$ is the set of dominators of block n.

Input: CFG with node set N, initial node n_0. **Output**: $D(n), \forall n \in N$

$D(n_0) := \{n_0\}$
for $n \in N - \{n_0\}$ do $D(n) := N$

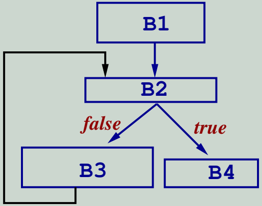
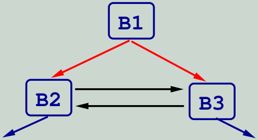
while changes to any $D(n)$ occur do
    for $n \in N - \{n_0\}$ do
        $D(n) := \{n\} \cup (\cap_{p \in pred(n)} D(p))$

High-Level Algorithm: With each backedge $n \rightarrow d$ (d is the loop header), we associate a natural loop (of $n \rightarrow d$) consisting of node d and all nodes that can reach n without going through d.

Intuition: since d is the only entry to the loop, a path from any block outside the loop must pass through d.

Reducible Control-Flow Graphs (CFG)

A CFG is reducible if it can be partitioned in forward and backward edges, and the forward edges form a directed-acyclic graph.

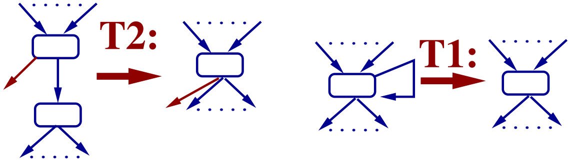
Reducible CFG	Irreducible CFG
	

Irreducible CFG: due to unstructured GOTOs,
e.g., jumps in the middle of a loop.

How to test CFG reducibility?

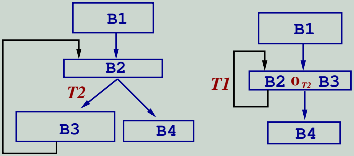
Testing CFG Reducibility via T1-T2 Transformation

Why Is Reducibility Important? 1. A reducible CFG can be written only in terms of (while/do) loops, if statements and function calls.



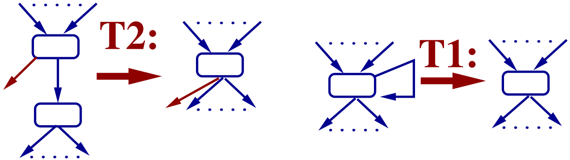
Reducible CFG *iff* can be reduced to 1 node via T1/T2 applications.

Reducible CFG via T2



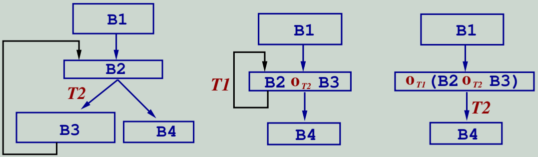
Testing CFG Reducibility via T1-T2 Transformation

Why Is Reducibility Important? 1. A reducible CFG can be written only in terms of (while/do) loops, if statements and function calls.



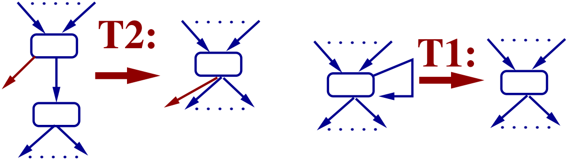
Reducible CFG *iff* can be reduced to 1 node via T1/T2 applications.

Reducing CFG via T1



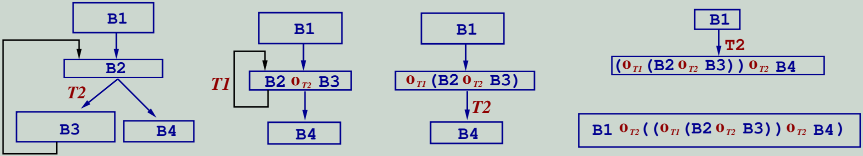
Testing CFG Reducibility via T1-T2 Transformation

Why Is Reducibility Important? 2. If ABSYN guarantees reducible CFG then data-flow rules associated with each ABSYN node; the result is composed in one program traversal, rather than fix-point iter.



Reducible CFG *iff* can be reduced to 1 node via T1/T2 applications.

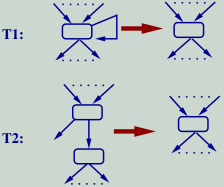
Reducing CFG via T2



Testing and Solving Ireducible CFG

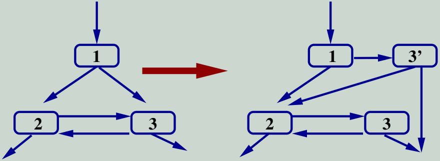
Alg for Testing Reducibility

In a copy of the CFG, apply `T1` and `T2` to a fixpoint. If the result is a single node than the CFG is reducible.



Node Splitting

Irreducible CFGs are difficult to optimize. It is always possible to solve irreducibility, but, in the worst case, at the cost of an exponential-code explosion:



Key property: if a CFG is reducible then all cycles are (regular) loops, and identifying the backedges is enough to find all loops.

CFG Reducibility via T1-T2 Transformation

Why Is Reducibility Important? 2. If AbSyn guarantees reducible CFG then equations associated with each AbSyn node; the result is composed in one program traversal, rather than fix-point iteration.

This allows optimizations to be implemented in the simpler, intuitive case-analysis style, e.g., live-function computation:

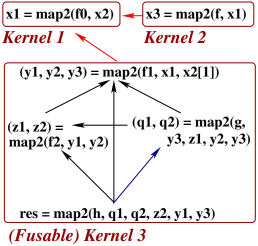
Pseudocode for computing the live functions (names)

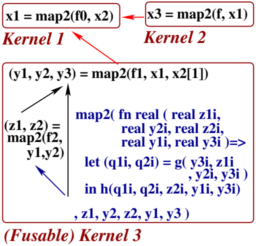
```
fun live_funs ( exp, ftab, livefs : string list ) : string list =
  case exp of
    Plus (e1, e2, pos) =>
        live_funs(e2, ftab, live_funs(e1, livefs, ftab) )
  | ...
  | FunApp(fid, args, pos) =>
        let val elives = foldl ( fn(e,lives)=>live_funs(e, ftab, lives) )
                          livefs args
        in  if( fid ∈ elives ) then elives
            else live_funs( fid's body, ftab, fid::elives )
        end
```

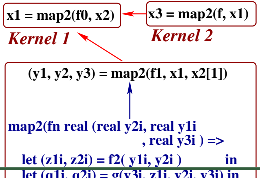
T1-T2 Graph Reduction

Why Is Reducibility Important? 3. Other less-conventional applications, such as map fusion without duplicating computation.

Troels Henriksen and C. Oancea, A T2 Graph-Reduction Approach To Fusion, *Procs. ACM Workshop on Funct. High Perf. Comp.* 2013.







Example of Simple Optimizations

Liveness analysis & Register allocation: compiler course.

Common-Subexpression Elimination (CSE): if the same expression e is computed twice, replace if possible the second occurrence of e with the temporary that holds the value of the first computation of e.

Copy Propagation (CP): after a statement `x := y`, we know that `x` and `y` have the same value, hence we can replace all occurrences of `x` with `y` between this assignment and the next definition of `x` or `y`.

Dead-Code Elimination (DC) (one reason is copy propagation):

- can safely remove any statement that defines a dead variable,
- a branch to dead code moved to whatever follows the dead code,
- if a branch-condition value is statically known $\Leftrightarrow$ merge two BBs.

Constant Folding and Propagation (CtF/P): if possible, expressions should be computed at compile time, and the constant result should be propagated. **And these are just a few!**

Example: Common-Subexpression Elimination (CSE) and Copy Propagation (CP)

Shown at basic-block level, but *Data-flow* analysis extends it on CFG.

Original	After CSE1	After CP1	After CSE2 & CP2
`t1 := 4  - 2`	`t1 := 4  - 2`	`t1 := 4  - 2`	`t1 := 4  - 2`
`t2 := t1 / 2`	`t2 := t1 / 2`	`t2 := t1 / 2`	`t2 := t1 / 2`
`t3 := a  * t2`	`t3 := a  * t2`	`t3 := a  * t2`	`t3 := a  * t2`
`t4 := t3 * t1`	`t4 := t3 * t1`	`t4 := t3 * t1`	`t4 := t3 * t1`
`t5 := t4 + b`	`t5 := t4 + b`	`t5 := t4 + b`	`t5 := t4 + b`
`t6 := t3 * t1`	`t6 := t4`	`t6 := t4`	`t6 := t4`
`t7 := t6 + b`	`t7 := t6 + b`	`t7 := t4 + b`	`t7 := t5`
`c  := t5 * t7`	`c  := t5 * t7`	`c  := t5 * t7`	`c  := t5 * t5`

Copy propagation makes further common-subexpression elimination possible and the reverse.

Example Continuation: Constant Folding (CFP) & Copy Propagation (CP) & Dead Code Elim (DCE)

Shown at basic-block level, but *Data-flow* analysis extends it on CFG.

Original	After CtFP	After CP	After DCE
`t1 := 4 - 2`	`t1 := 2`	`t1 := 2`	
`t2 := t1 / 2`	`t2 := 1`	`t2 := 1`	
`t3 := a * t2`	`t3 := a`	`t3 := a`	
`t4 := t3 * t1`	`t4 := t3 * 2`	`t4 := a * 2`	`t4 := a * 2`
`t5 := t4 + b`	`t5 := t4 + b`	`t5 := t4 + b`	`t5 := t4 + b`
`t6 := t4`	`t6 := t4`	`t6 := t4`	
`t7 := t5`	`t7 := t5`	`t7 := t5`	
`c := t5 * t5`	`c := t5 * t5`	`c := t5 * t5`	`c := t5 * t5`

Very useful in optimizing the critical-execution path or eliminating the redundancies introduced by various transformations (by the compiler).