Compiler Optimisation

6 – Instruction Scheduling

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2019

Introduction

This lecture:

- Scheduling to hide latency and exploit ILP
- Dependence graph
- Local list Scheduling + priorities
- Forward versus backward scheduling
- Software pipelining of loops

Latency, functional units, and ILP

- Instructions take clock cycles to execute (*latency*)
- Modern machines issue several operations per cycle
- Cannot use results until ready, can do something else
- Execution time is *order-dependent*
- Latencies not always constant (cache, early exit, etc)

Operation	**Cycles**
`load, store`	3
`load` $\notin$ cache	100s
`loadI, add, shift`	1
`mult`	2
`div`	40
`branch`	$0-8$

Machine types

- In order
  - Deep pipelining allows multiple instructions
- Superscalar
  - Multiple functional units, can issue > 1 instruction
- Out of order
  - Large window of instructions can be reordered dynamically
- VLIW
  - Compiler statically allocates to FUs

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	
	`add`	r_1, r_1	$\Rightarrow r_1$	
	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[1] loads/stores 3 cycles, mults 2, adds 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2				r_1
3				$\mathbf{r_1}$
	`add`	r_1, r_1	$\Rightarrow r_1$	
	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[1] `loads`/`stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2				r_1
3				$\mathbf{r_1}$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}$
	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[1] loads/stores 3 cycles, mults 2, adds 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2				r_1
3				$\mathbf{r_1}$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}$
5	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_2
6				r_2
7				$\mathbf{r_2}$
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[1] `loads`/`stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2				r_1
3				$\mathbf{r_1}$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}$
5	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_2
6				r_2
7				$\mathbf{r_2}$
8	`mult`	r_1, r_2	$\Rightarrow r_1$	r_1
9	**Next op does not use r_1**			$\mathbf{r_1}$
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[1] loads/stores 3 cycles, mults 2, adds 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2				r_1
3				$\mathbf{r_1}$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}$
5	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_2
6				r_2
7				$\mathbf{r_2}$
8	`mult`	r_1, r_2	$\Rightarrow r_1$	r_1
9	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	$\mathbf{r_1}, r_2$
10				r_2
11				$\mathbf{r_2}$
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[1] `loads/stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2				r_1
3				$\mathbf{r_1}$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}$
5	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_2
6				r_2
7				$\mathbf{r_2}$
8	`mult`	r_1, r_2	$\Rightarrow r_1$	r_1
9	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	$\mathbf{r_1}, r_2$
10				r_2
11				$\mathbf{r_2}$
12	`mult`	r_1, r_2	$\Rightarrow r_1$	r_1
13				$\mathbf{r_1}$
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[1] loads/stores 3 cycles, mults 2, adds 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Simple schedule[1] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2				r_1
3				$\mathbf{r_1}$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}$
5	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_2
6				r_2
7				$\mathbf{r_2}$
8	`mult`	r_1, r_2	$\Rightarrow r_1$	r_1
9	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_2$	$\mathbf{r_1}, r_2$
10				r_2
11				$\mathbf{r_2}$
12	`mult`	r_1, r_2	$\Rightarrow r_1$	r_1
13				$\mathbf{r_1}$
14	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	store to complete
15				store to complete
16				**store to complete**
	Done			

[1] `loads/stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	
	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	
	`add`	r_1, r_1	$\Rightarrow r_1$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[2] `loads/stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	
	`add`	r_1, r_1	$\Rightarrow r_1$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`mult`	r_1, r_3	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[2] `loads`/`stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_1, r_2
	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	
	`add`	r_1, r_1	$\Rightarrow r_1$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`mult`	r_1, r_3	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[2] `loads/stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2]   `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_1, r_2
3	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	$\mathbf{r_1}, r_2, r_3$
	`add`	r_1, r_1	$\Rightarrow r_1$	
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`mult`	r_1, r_3	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[2] `loads`/`stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_1, r_2
3	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	$\mathbf{r_1}, r_2, r_3$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}, \mathbf{r_2}, r_3$
	`mult`	r_1, r_2	$\Rightarrow r_1$	
	`mult`	r_1, r_3	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[2] `loads/stores` 3 cycles, `mults` 2, `adds` 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_1, r_2
3	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	$\mathbf{r_1}, r_2, r_3$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}, \mathbf{r_2}, r_3$
5	`mult`	r_1, r_2	$\Rightarrow r_1$	$r_1, \quad \mathbf{r_3}$
6				$\mathbf{r_1}$
	`mult`	r_1, r_3	$\Rightarrow r_1$	
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[2] loads/stores 3 cycles, mults 2, adds 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2] `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_1, r_2
3	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	$\mathbf{r_1}, r_2, r_3$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}, \mathbf{r_2}, r_3$
5	`mult`	r_1, r_2	$\Rightarrow r_1$	$r_1, \mathbf{r_3}$
6				$\mathbf{r_1}$
7	`mult`	r_1, r_3	$\Rightarrow r_1$	r_1
8				$\mathbf{r_1}$
	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	
	Done			

[2] loads/stores 3 cycles, mults 2, adds 1

Effect of scheduling

Superscalar, 1 FU: New op each cycle if operands ready

Schedule loads early[2]   `a := 2*a*b*c`

Cycle	Operations			Operands waiting
1	`loadAI`	$r_{arp}, @a$	$\Rightarrow r_1$	r_1
2	`loadAI`	$r_{arp}, @b$	$\Rightarrow r_2$	r_1, r_2
3	`loadAI`	$r_{arp}, @c$	$\Rightarrow r_3$	$\mathbf{r_1}, r_2, r_3$
4	`add`	r_1, r_1	$\Rightarrow r_1$	$\mathbf{r_1}, \mathbf{r_2}, r_3$
5	`mult`	r_1, r_2	$\Rightarrow r_1$	r_1, $\mathbf{r_3}$
6				$\mathbf{r_1}$
7	`mult`	r_1, r_3	$\Rightarrow r_1$	r_1
8				$\mathbf{r_1}$
9	`storeAI`	r_1	$\Rightarrow r_{arp}, @a$	store to complete
10				store to complete
11				**store to complete**
	Done			

Uses one more register

11 versus 16 cycles – 31% faster!

[2] `loads/stores` 3 cycles, `mults` 2, `adds` 1

Scheduling problem

- Schedule maps operations to cycle; $\forall a \in Ops, S(a) \in \mathbb{N}$
- Respect latency;
  $\forall a, b \in Ops, a\ dependson\ b \implies S(a) \geq S(b) + \lambda(b)$
- Respect function units; no more ops per type per cycle than FUs can handle

- Length of schedule, $L(S) = max_{a \in Ops}(S(a) + \lambda(a))$
- Schedule S is time-optimal if $\forall S_1, L(S) \leq L(S_1)$

- **Problem:** Find a time-optimal schedule[3]
- Even local scheduling with many restrictions is *NP-complete*

[3] A schedule might also be optimal in terms of registers, power, or space

List scheduling

Local greedy heuristic to produce schedules for single basic blocks

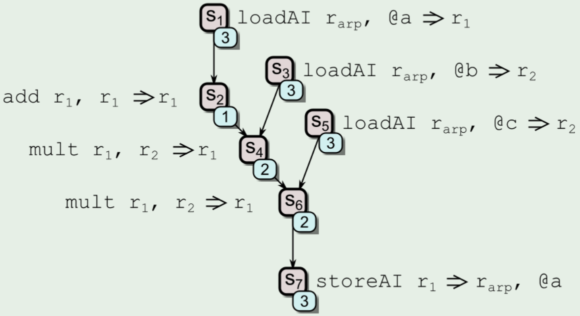
1. Rename to avoid anti-dependences
2. Build dependency graph
3. Prioritise operations
4. For each cycle
    1. Choose the highest priority ready operation & schedule it
    2. Update ready queue

List scheduling

Dependence/Precedence graph

- Schedule operation only when operands ready
- Build dependency graph of read-after-write (RAW) deps
    - Label with latency and FU requirements

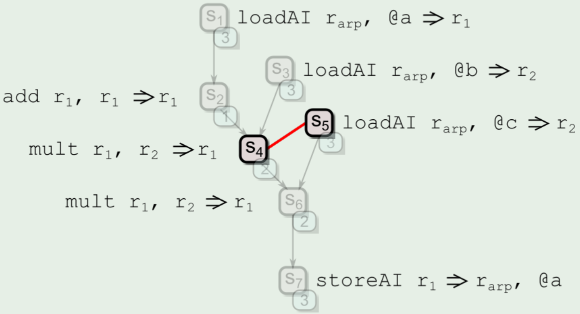
Example: a = 2*a*b*c



List scheduling
Dependence/Precedence graph

- Schedule operation only when operands ready
- Build dependency graph of read-after-write (RAW) deps
  - Label with latency and FU requirements
- Anti-dependences (WAR) restrict movement

Example: a = 2*a*b*c

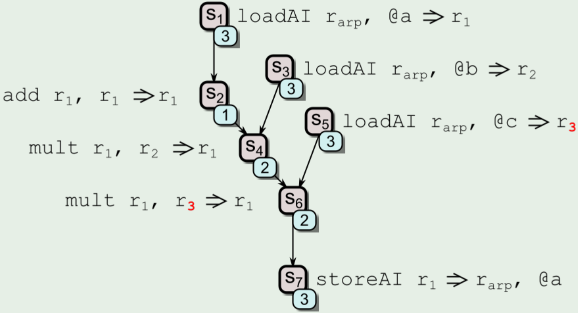


List scheduling

Dependence/Precedence graph

- Schedule operation only when operands ready
- Build dependency graph of read-after-write (RAW) deps
  - Label with latency and FU requirements
- Anti-dependences (WAR) restrict movement – renaming removes

Example: `a = 2*a*b*c`



List scheduling

List scheduling algorithm

```
Cycle ← 1
Ready ← leaves of (D)
Active ← ∅
while(Ready ∪ Active ≠ ∅)
    ∀a ∈ Active where S(a) + λ(a) ≤ Cycle
        Active ← Active - a
        ∀ b ∈ succs(a) where isready(b)
            Ready ← Ready ∪ b
    if ∃ a ∈ Ready and ∀ b, a_priority ≥ b_priority
        Ready ← Ready - a
        S(op) ← Cycle
        Active ← Active ∪ a
    Cycle ← Cycle + 1
```

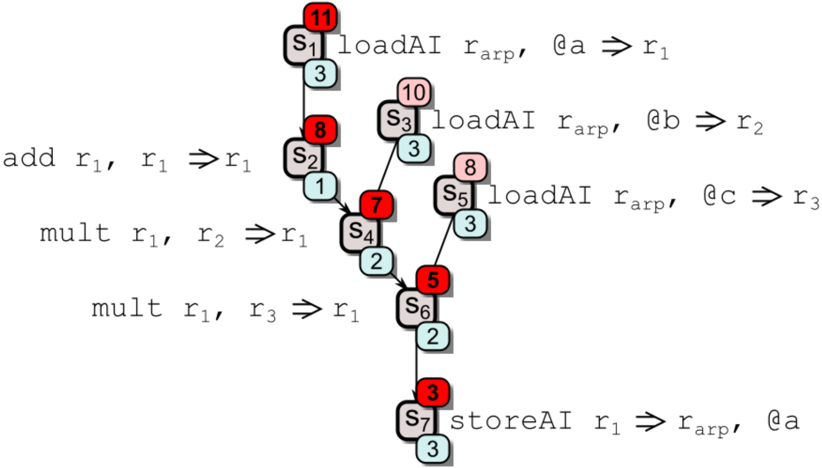
List scheduling

Priorities

- Many different priorities used
  - Quality of schedules depends on good choice
- The longest latency path or critical path is a good priority
- Tie breakers
  - Last use of a value - decreases demand for register as moves it nearer def
  - Number of descendants - encourages scheduler to pursue multiple paths
  - Longer latency first - others can fit in shadow
  - Random

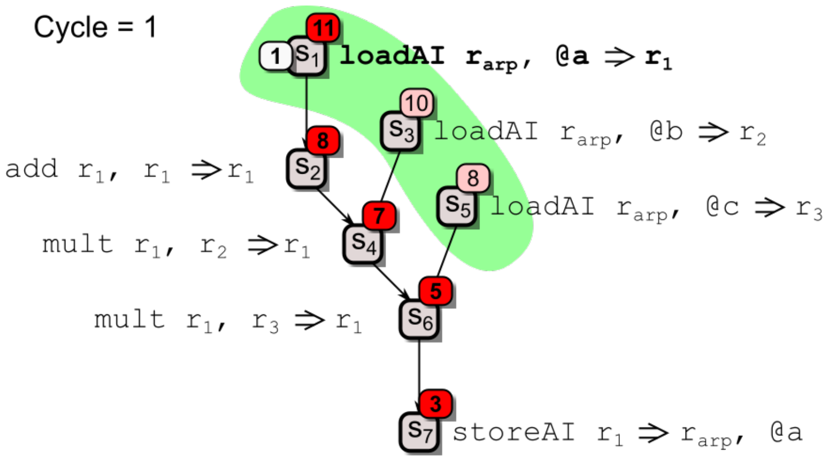
List scheduling

Example: Schedule with priority by critical path length



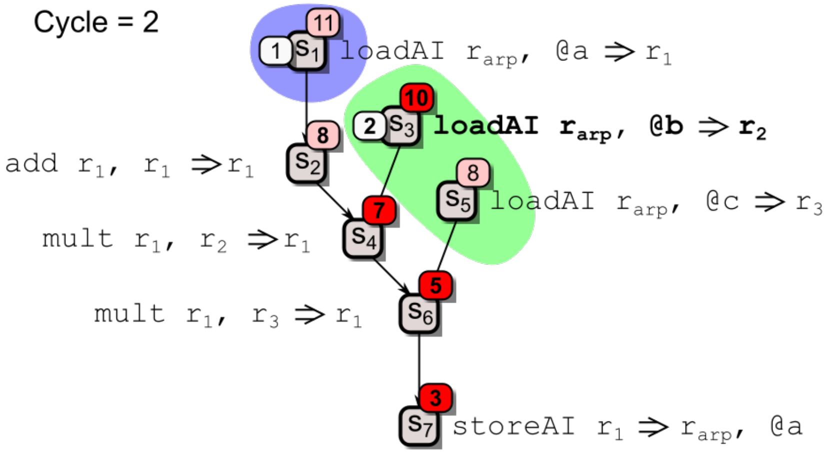
List scheduling

Example: Schedule with priority by critical path length



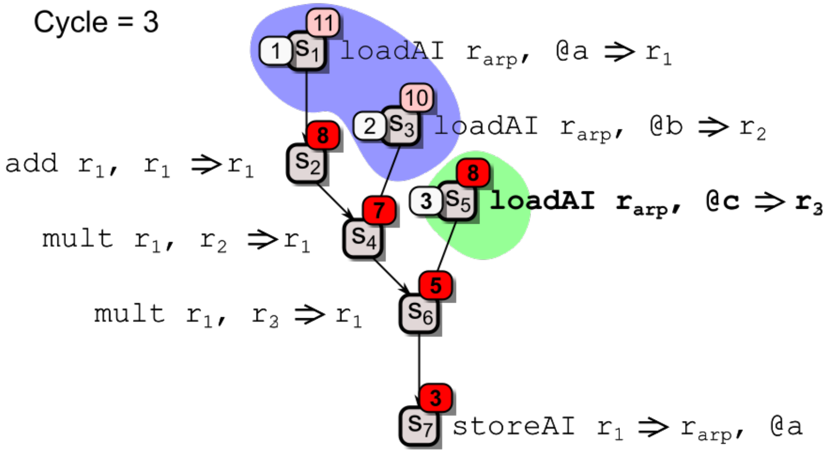
List scheduling

Example: Schedule with priority by critical path length



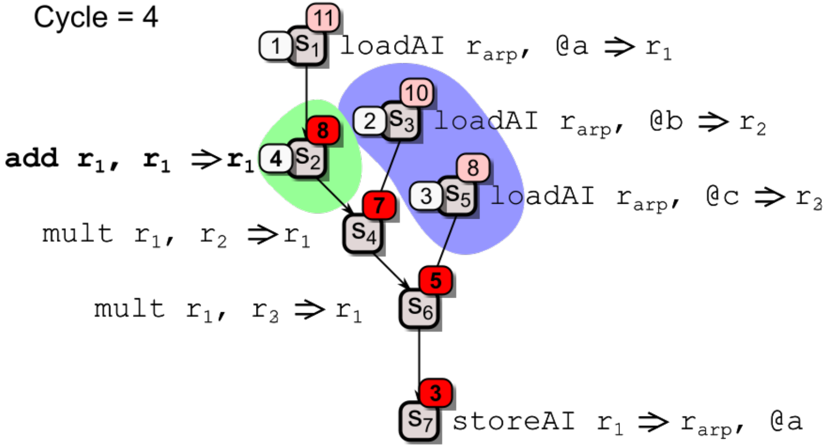
List scheduling

Example: Schedule with priority by critical path length



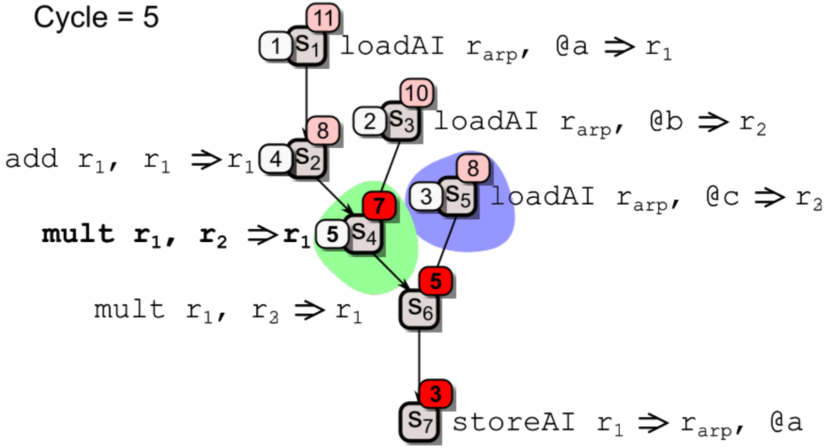
List scheduling

Example: Schedule with priority by critical path length



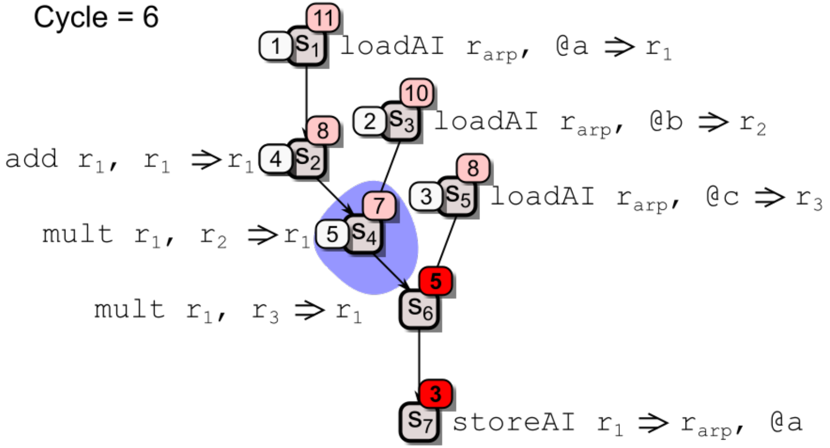
List scheduling

Example: Schedule with priority by critical path length



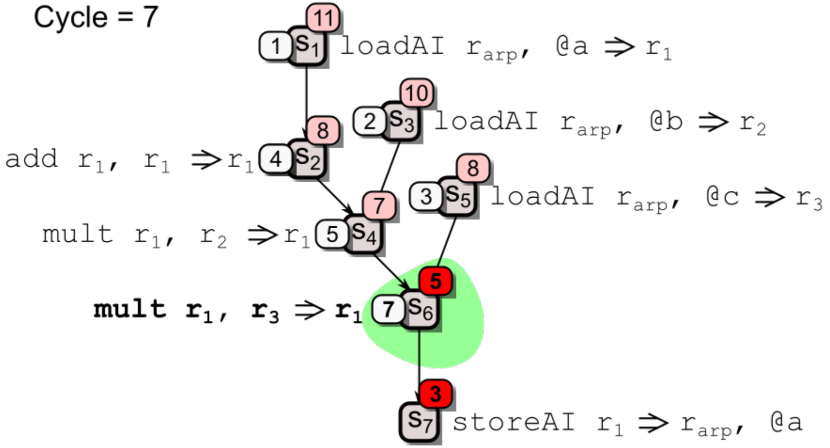
List scheduling

Example: Schedule with priority by critical path length



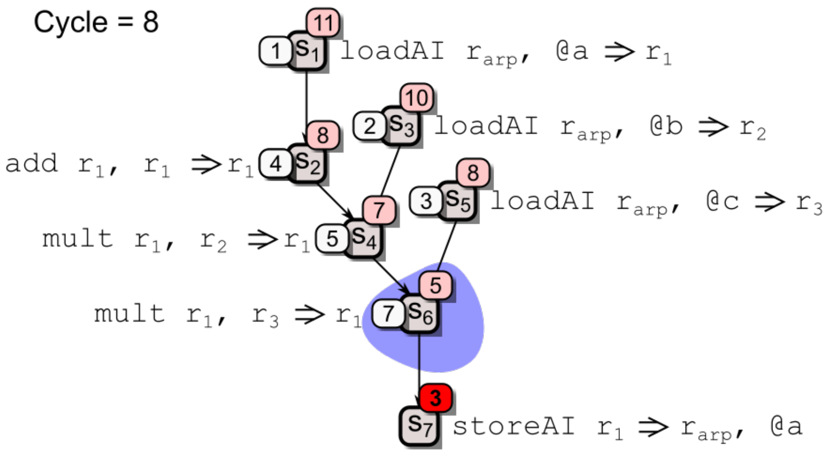
List scheduling

Example: Schedule with priority by critical path length



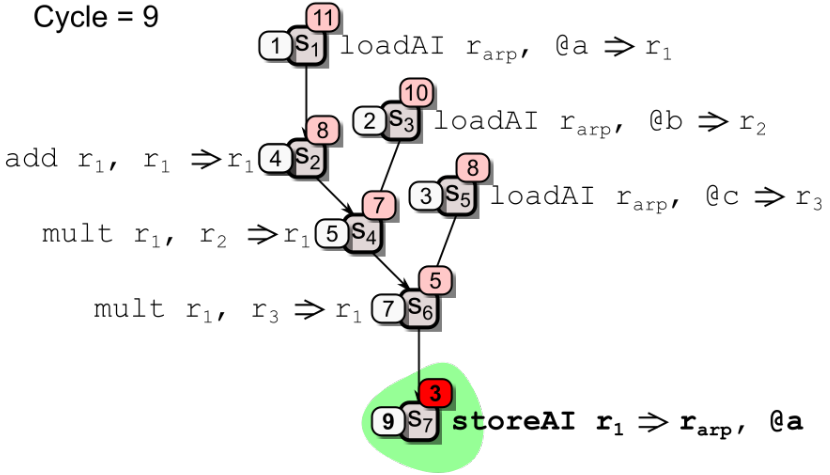
List scheduling

Example: Schedule with priority by critical path length



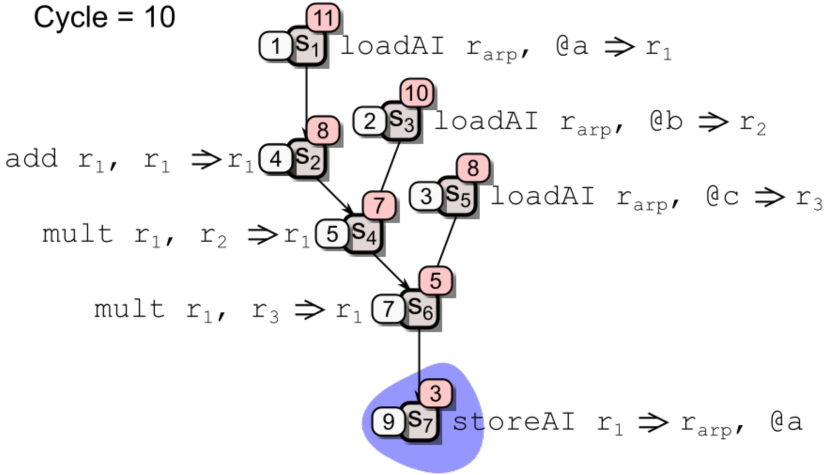
List scheduling

Example: Schedule with priority by critical path length



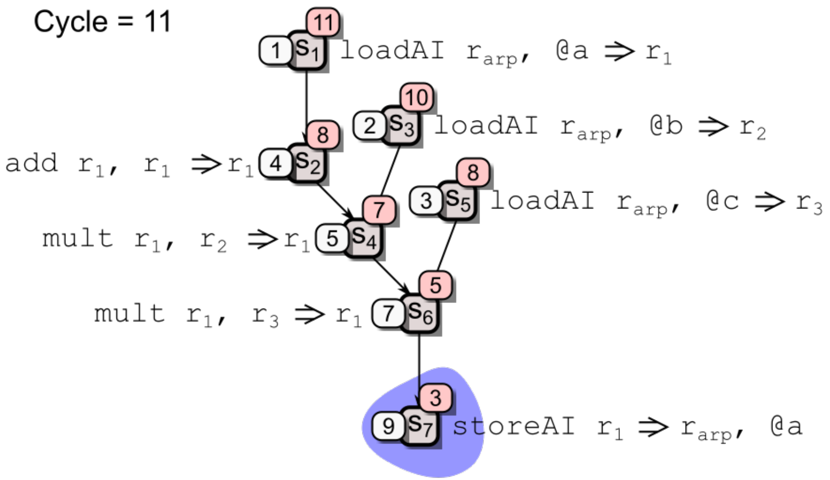
List scheduling

Example: Schedule with priority by critical path length



List scheduling

Example: Schedule with priority by critical path length

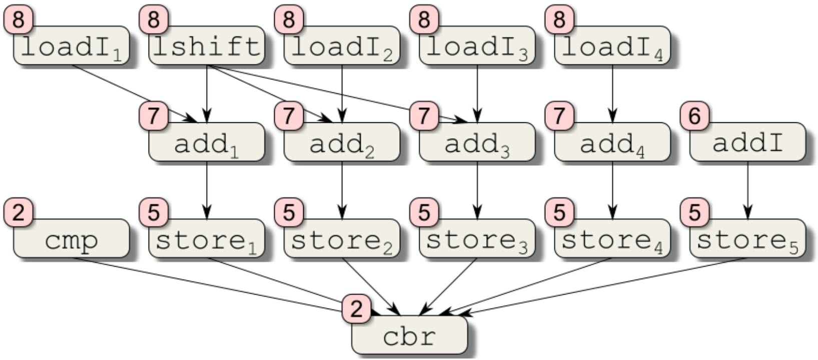


List scheduling

Forward vs backward

- Can schedule from root to leaves (backward)
- May change schedule time
- List scheduling cheap, so try both, choose best

List scheduling
Forward vs backward



Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

List scheduling

Forward vs backward

Forwards			
	Int	Int	Stores
1	$\texttt{loadI}_1$	`lshift`	
2	$\texttt{loadI}_2$	$\texttt{loadI}_3$	
3	$\texttt{loadI}_4$	$\texttt{add}_1$	
4	$\texttt{add}_2$	$\texttt{add}_3$	
5	$\texttt{add}_4$	`addI`	$\texttt{store}_1$
6	`cmp`		$\texttt{store}_2$
7			$\texttt{store}_3$
8			$\texttt{store}_4$
9			$\texttt{store}_5$
10			
11			
12			
13	`cbr`		

Backwards			
	Int	Int	Stores
1	$\texttt{loadI}_1$		
2	`addI`	`lshift`	
3	$\texttt{add}_4$	$\texttt{loadI}_3$	
4	$\texttt{add}_3$	$\texttt{loadI}_2$	$\texttt{store}_5$
5	$\texttt{add}_2$	$\texttt{loadI}_1$	$\texttt{store}_4$
6	$\texttt{add}_1$		$\texttt{store}_3$
7			$\texttt{store}_2$
8			$\texttt{store}_1$
9			
10			
11	`cmp`		
12	`cbr`		

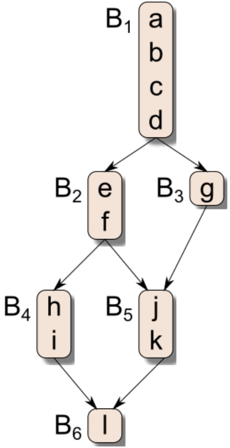
Scheduling Larger Regions

- Schedule extended basic blocks (EBBs)
    - Super block cloning
- Schedule traces
- Software pipelining

Scheduling Larger Regions
Extended basic blocks

Extended basic block

EBB is maximal set of blocks such that
Set has a single entry, B_i
Each block B_j other than B_i has
exactly one predecessor



Scheduling Larger Regions

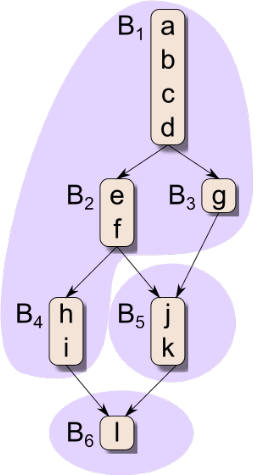
Extended basic blocks

Extended basic block

EBB is maximal set of blocks such that

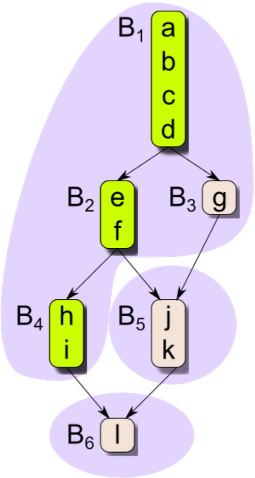
Set has a single entry, B_i

Each block B_j other than B_i has exactly one predecessor



Scheduling Larger Regions
Extended basic blocks

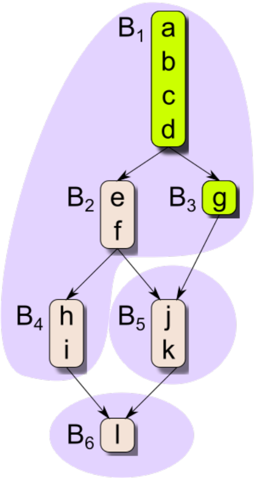
- Schedule entire paths through EBBs
- Example has four EBB paths



Scheduling Larger Regions

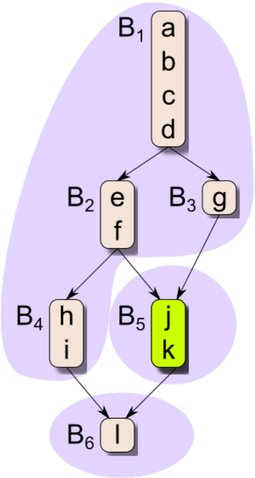
Extended basic blocks

- Schedule entire paths through EBBs
- Example has four EBB paths



Scheduling Larger Regions
Extended basic blocks

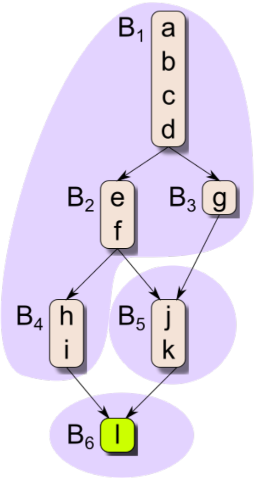
- Schedule entire paths through EBBs
- Example has four EBB paths



Scheduling Larger Regions

Extended basic blocks

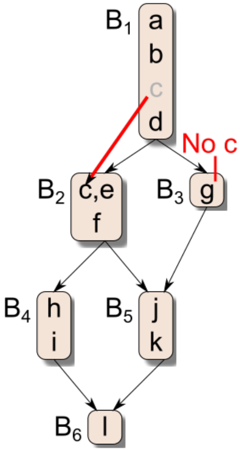
- Schedule entire paths through EBBs
- Example has four EBB paths



Scheduling Larger Regions

Extended basic blocks

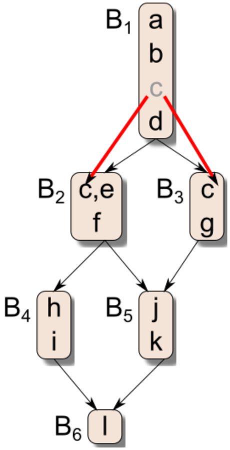
- Schedule entire paths through EBBs
- Example has four EBB paths
- Having B_1 in both causes conflicts
  - Moving an op **out of** B_1 causes problems



Scheduling Larger Regions

Extended basic blocks

- Schedule entire paths through EBBs
- Example has four EBB paths
- Having B_1 in both causes conflicts
  - Moving an op **out of** B_1 causes problems
  - Must insert compensation code

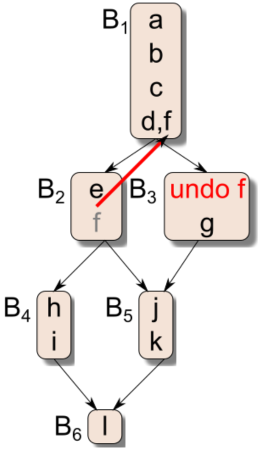


Scheduling Larger Regions

Extended basic blocks

- Schedule entire paths through EBBs
- Example has four EBB paths
- Having B_1 in both causes conflicts

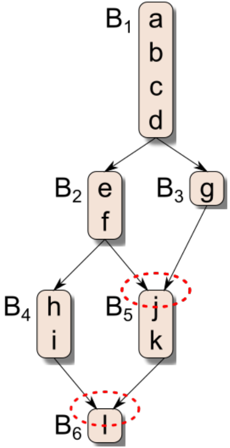
  - Moving an op **into** B_1 causes problems



Scheduling Larger Regions

Superblock cloning

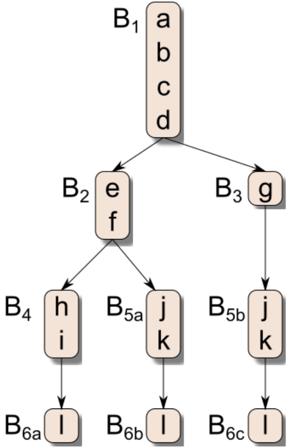
- Join points create context problems



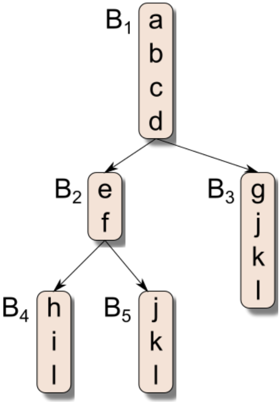
Scheduling Larger Regions

Superblock cloning

- Join points create context problems
- Clone blocks to create more context

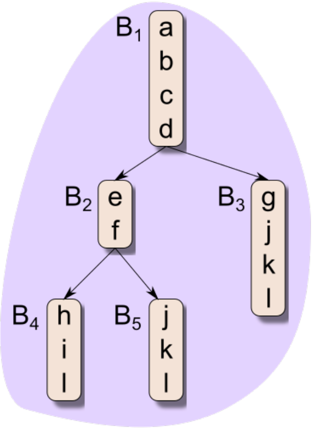


- Join points create context problems
- Clone blocks to create more context
- Merge any simple control flow



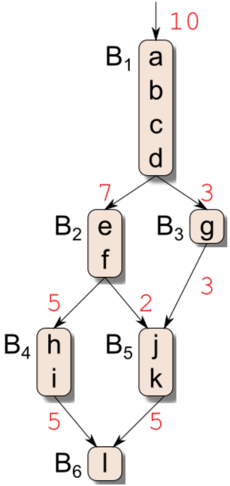
Scheduling Larger Regions
Superblock cloning

- Join points create context problems
- Clone blocks to create more context
- Merge any simple control flow
- Schedule EBBs



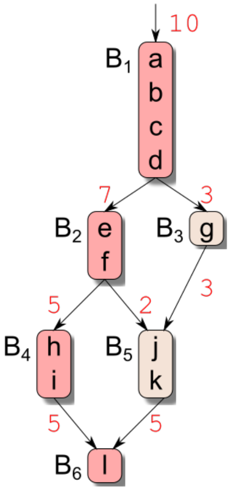
Scheduling Larger Regions
Trace scheduling

- Edge frequency from profile (not block frequency)



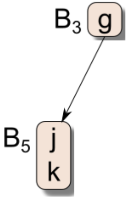
Scheduling Larger Regions
Trace scheduling

- Edge frequency from profile (not block frequency)
- Pick "hot" path
- Schedule with compensation code



Scheduling Larger Regions
Trace scheduling

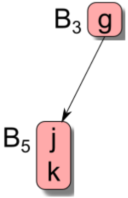
- Edge frequency from profile (not block frequency)
- Pick "hot" path
- Schedule with compensation code
- Remove from CFG



Scheduling Larger Regions

Trace scheduling

- Edge frequency from profile (not block frequency)
- Pick "hot" path
- Schedule with compensation code
- Remove from CFG
- Repeat



Loop scheduling

- Loop structures can dominate execution time
- Specialist technique software pipelining
- Allows application of list scheduling to loops

- Why not loop unrolling?

Loop scheduling

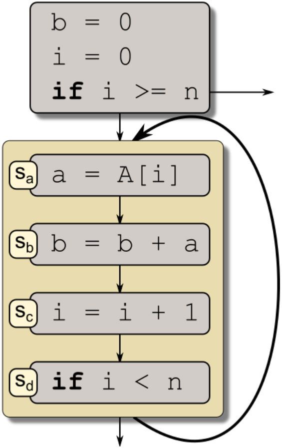
- Loop structures can dominate execution time
- Specialist technique software pipelining
- Allows application of list scheduling to loops

- Why not loop unrolling?
- Allows loop effect to become arbitrarily small, but
- Code growth, cache pressure, register pressure

Software pipelining

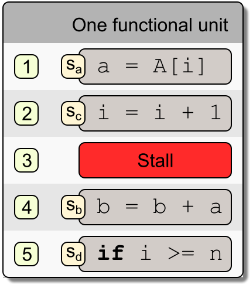
Consider simple loop to sum array

```
b = 0
for i = 0 to n
    b = b + A[i]
```



Software pipelining

Schedule on 1 FU - 5 cycles



`load` 3 cycles, `add` 1 cycle, branch 1 cycle

Software pipelining

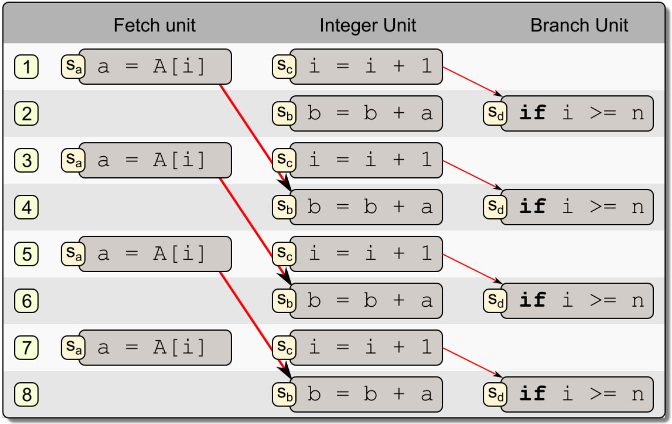
Schedule on VLIW 3 FUs - 4 cycles

	Fetch unit	Integer Unit	Branch Unit
1	S_a `a = A[i]`	S_c `i = i + 1`	
2		Stall	
3		Stall	
4		S_b `b = b + a`	S_d `if i < n` 1

`load` 3 cycles, `add` 1 cycle, `branch` 1 cycle

Software pipelining

A better steady state schedule exists



`load` 3 cycles, `add` 1 cycle, branch 1 cycle

Software pipelining

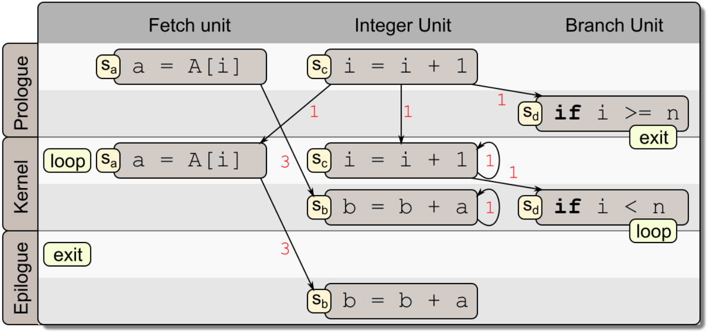
Requires prologue and epilogue (may schedule others in epilogue)

	Fetch unit	Integer Unit	Branch Unit
Prologue	S_a `a = A[i]`	S_c `i = i + 1`	
			S_d **if** `i >= n` exit
Kernel	loop S_a `a = A[i]`	S_c `i = i + 1`	
		S_b `b = b + a`	S_d **if** `i < n` loop
Epilogue	exit		
		S_b `b = b + a`	

`load` 3 cycles, `add` 1 cycle, `branch` 1 cycle

Software pipelining

Respect dependences and latency – including loop carries



`load` 3 cycles, `add` 1 cycle, branch 1 cycle

Software pipelining

Complete code

	Fetch unit	Integer Unit	Branch Unit
	`nop`	`b = 0`	`nop`
	`nop`	`i = 0`	`if i >= n` skip
	S_a `a = A[i]`	S_c `i = i + 1`	`nop`
	`nop`	`nop`	S_d `if i >= n` exit
loop	S_a `a = A[i]`	S_c `i = i + 1`	`nop`
	`nop`	S_b `b = b + a`	S_d `if i < n` loop
exit	`nop`	`nop`	`nop`
	`nop`	S_b `b = b + a`	`nop`
skip			

`load` 3 cycles, `add` 1 cycle, branch 1 cycle

Initiation interval (*ii*)

Number of cycles between initiating loop iterations

- Original loop had *ii* of 5 cycles
- Final loop had *ii* of 2 cycles

Recurrence

Loop-based computation whose value is used in later loop iteration

- Might be several iterations later
- Has dependency chain(s) on itself
- Recurrence latency is latency of dependency chain

Software pipelining
Algorithm

- Choose an initiation interval, *ii*
    - Compute lower bounds on *ii*
    - Shorter ii means faster overall execution
- Generate a loop body that takes *ii* cycles
    - Try to schedule into *ii* cycles, using modulo scheduler
    - If it fails, increase *ii* by one and try again
- Generate the needed prologue and epilogue code
    - For prologue, work backward from upward exposed uses in the scheduled loop body
    - For epilogue, work forward from downward exposed definitions in the scheduled loop body

Software pipelining

Initial initiation interval (*ii*)

Starting value for *ii* based on minimum resource and recurrence constraints

Resource constraint

- *ii* must be large enough to issue every operation
- Let N_u = number of FUs of type u
- Let I_u = number of operations of type u
- $\lceil I_u / N_u \rceil$ is lower bound on *ii* for type u
- $\mathbf{max_u(\lceil I_u / N_u \rceil)}$ is lower bound on *ii*

Starting value for *ii* based on minimum resource and recurrence constraints

Recurrence constraint

- *ii* cannot be smaller than longest recurrence latency
- Recurrence r is over k_r iterations with latency λ_r
- $\lceil \lambda_r / k_u \rceil$ is lower bound on *ii* for type r
- $\mathbf{max_r}(\lceil \lambda_\mathbf{r} / \mathbf{k_u} \rceil)$ is lower bound on *ii*

Software pipelining

Initial initiation interval (*ii*)

Starting value for *ii* based on minimum resource and recurrence constraints

Start value $= max(max_u(\lceil I_u/N_u \rceil), max_r(\lceil \lambda_r/k_u \rceil)$

For simple loop

```
a = A[ i ]
b = b + a
i = i + 1
if i < n goto
end
```

Resource constraint

	Memory	**Integer**	**Branch**
I_u	1	2	1
N_u	1	1	1
$\lceil I_u/N_u \rceil$	1	**2**	1

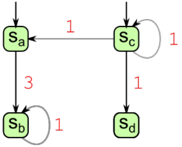
Recurrence constraint

	b	**i**
k_r	1	1
λ_r	2	1
$\lceil I_u/N_u \rceil$	**2**	1

Modulo scheduling

Schedule with cycle modulo initiation interval

S_a	`a = A[i]`
S_b	`b = b + a`
S_c	`i = i + 1`
S_d	`if i >= n`

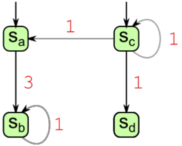


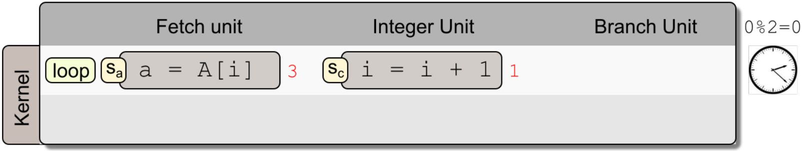
	Fetch unit	Integer Unit	Branch Unit
Kernel	loop		

Modulo scheduling

Schedule with cycle modulo initiation interval

```
Sa  a = A[i]
Sb  b = b + a
Sc  i = i + 1
Sd  if i >= n
```

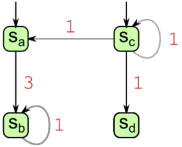


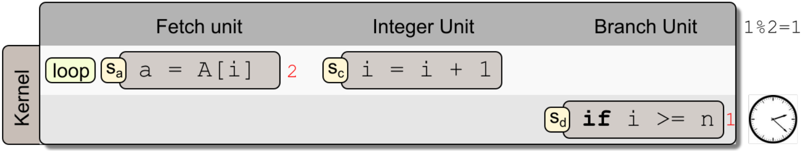


Modulo scheduling

Schedule with cycle modulo initiation interval

```
Sa  a = A[i]
Sb  b = b + a
Sc  i = i + 1
Sd  if i >= n
```

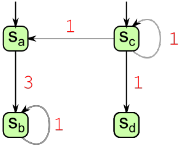


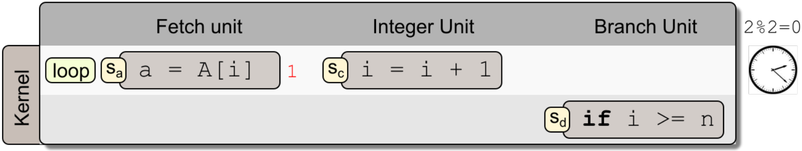


Modulo scheduling

Schedule with cycle modulo initiation interval

```
Sa  a = A[i]
Sb  b = b + a
Sc  i = i + 1
Sd  if i >= n
```

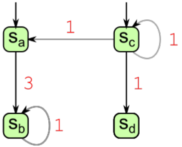




Modulo scheduling

Schedule with cycle modulo initiation interval

S_a a = A[i]
S_b b = b + a
S_c i = i + 1
S_d **if** i >= n



	Fetch unit	Integer Unit	Branch Unit
Kernel	loop S_a a = A[i]	S_c i = i + 1	
		S_b b = b + a	S_d **if** i >= n

3%2=1

- Much research in different software pipelining techniques
- Difficult when there is general control flow in the loop
- Predication in IA64 for example really helps here
- Some recent work in exhaustive scheduling -i.e. solve the NP-complete problem for basic blocks

Summary

- Scheduling to hide latency and exploit ILP
- Dependence graph - dependences between instructions + latency
- Local list Scheduling + priorities
- Forward versus backward scheduling
- Scheduling EBBs, superblock cloning, trace scheduling
- Software pipelining of loops

PPar CDT Advert

EPSRC Centre for Doctoral Training in

Pervasive Parallelism

- 4-year programme: MSc by Research + PhD
- Research-focused: Work on your thesis topic from the start
- Collaboration between:
  - University of Edinburgh's School of Informatics
    - Ranked top in the UK by 2014 REF
  - Edinburgh Parallel Computing Centre
    - UK's largest supercomputing centre
- Research topics in software, hardware, theory and application of:
  - Parallelism
  - Concurrency
  - Distribution
- Full funding available
- Industrial engagement programme includes internships at leading companies

The biggest revolution in the technological landscape for fifty years

Now accepting applications!
Find out more and apply at:
pervasiveparallelism.inf.ed.ac.uk

THE UNIVERSITY of EDINBURGH
informatics

EPSRC
Engineering and Physical Sciences Research Council