Instruction Scheduling

Michael O'Boyle

February, 2011

Course Structure

- Introduction and Recap
- Course Work
- Scalar optimisation and dataflow
- L5 Code generation
- L6 Instruction scheduling
- Then parallel approaches followed by adaptive compilation

Overview

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

Aim

- Order instructions to minimise execution time. Hide latency of instructions such as loads and branches by executing instructions in their shadow
- Exploit instruction level parallelism by making sure there are multiple instructions available to be simultaneously executed
- Two flavours of ILP: Superscalar and vliw. Both require similar analysis but vliw is static scheduled and requires more explicit treatment
- Affected by machine resources - number and type of functional unit, number of registers
- Assume register allocation is separately performed later.

Example Superscalar, 1 FU: New Op each cycle iff operands ready

w = w*2*x*y*z. Assume global activation pointer in r0

load/stores 3 cycles, mults 2, others 1

```
 1 loadAI r0,@w -> r1
 4 add r1,r1 ->r1
 5 loadAI r0,@x -> r2
 8 mult r1,r2->r1
 9 loadAI r0,@y ->r2
 12 mult r1,r2 ->r1
 13 loadAI r0,@z->r2
 16 mult r1,r2 ->r1
 18 storeAI r1->r0,@w
 21 r1 is free
```

```
1 loadAI r0,@w -> r1
2 loadAI r0,@x -> r2
3 loadAI r0,@y ->r3
4 add r1,r1 ->r1
5 mult r1,r2->r1
6 loadAI r0,@z->r2
7 mult r1,r3 ->r1
9 mult r1,r2 ->r1
11 storeAI r1->r0,@w
14 r1 is free
```

Second version - extra register, move loads earlier. Space vs time

List Scheduling

- Build a dependence graph of operations and delays
- Determine schedule to minimise execution time
- NP-complete: difficulty comes with determining which of the many available operands to schedule- need a priority function for tie breaking
- Use a greedy approach - list scheduling for local blocks
- Extend to greater scope later.

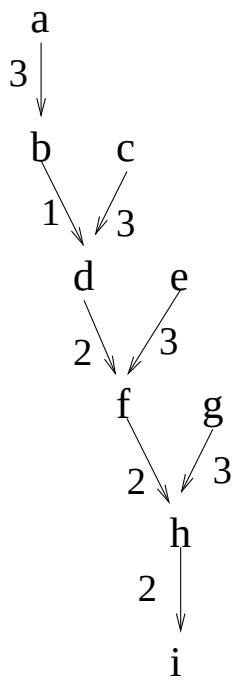
List Scheduling

```
cycle = 0
ready = leaves of dependence graph G
active = empty
while (ready union active != empty)
 if available remove an instruction from ready based on priority
 add instruction to active

 for each instruction in active
  if completed remove from active
  for each successor of instruction
    if successors operand ready then add to ready
```

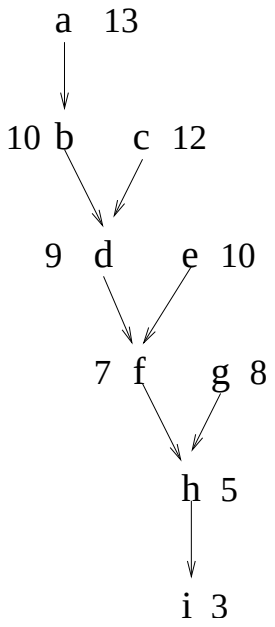
Example:

```
a loadAI r0,@w -> r1
b add r1,r1 ->r1
c loadAI r0,@x -> r2
d mult r1,r2->r1
e loadAI r0,@y ->r2
f mult r1,r2 ->r1
g loadAI r0,@z->r2
h mult r1,r2 ->r1
i storeAI r1->r0,@w
```



Ignore anti-dependences - assume unlimited registers

Critical path a b d f h i



Example

```
1 a loadAI r0,@w -> r1
2 c loadAI r0,@x -> r2
3 e loadAI r0,@y ->r3
4 b add r1,r1 ->r1
5 d mult r1,r2->r1
6 g loadAI r0,@z->r2
7 f mult r1,r3 ->r1
9 h mult r1,r2 ->r1
11 i  storeAI r1->r0,@w
```

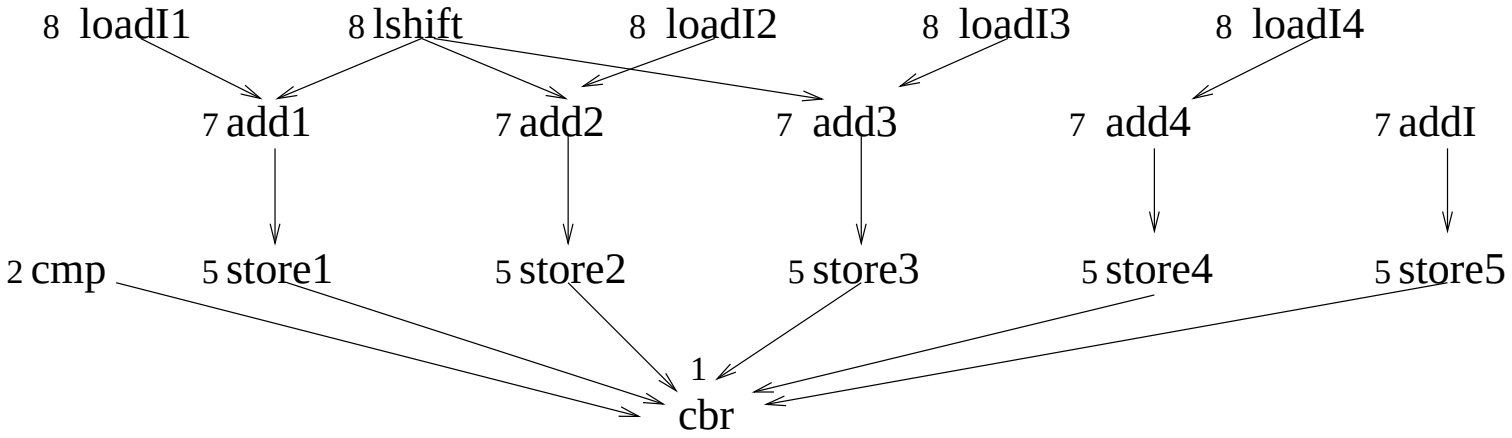
List Scheduling here uses critical path as priority. The labelled arcs denote critical path length for each instruction. Choose highest value first.

Priorities

- The longest latency path or critical path is a good priority
- 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
- Forward list scheduling does well but sometimes backward does better.

Forward vs Backward: 3 unit VLIW. Does NOT wait for operands

You are responsible for them being available: Fill delays with noops!



opcode	loadI	lshift	add	addI	cmp	store
latency	1	1	2	1	1	4

Schedule for 3 units - integer, integer and store

Priority to critical path - tie break left to right

Forward and Backward Scheduling: Blanks = noops

	Int	Int	Stores
1	loadI1	lshift	
2	loadI2	loadI3	
3	loadI4	add1	
4	add2	add3	
5	add4	addI	store1
6	cmp		store2
7			store3
8			store4
9			store5
10			
11			
12			
13	cbr		

	Int	Int	Stores
1	loadI1		
2	addI	lshift	
3	add4	loadI3	
4	add3	loadI2	store5
5	add2	loadI1	store4
6	add1		store3
7			store2
8			store1
9			
10			
11	cmp		
12	cbr		
13			

Loop scheduling

- Loop structures can dominate execution time
- Specialist technique software pipelining
- Calculation of minimum initiation interval
- This corresponds to the critical path of a loop
- Modulo Scheduling take into account resources

Software pipelining

- Scheme aimed at exploiting ILP in loops: Lam 1998. Significant impact on performance on statically scheduled vliw.
- Previous techniques need unrolling of loop to perform well.
- The recurrence or cyclic dependence length is the equivalent to the critical path
- Achieves performance by overlapping different iterations of a loop
- Has same effect as hardware pipelining available in out-of-order superscalar

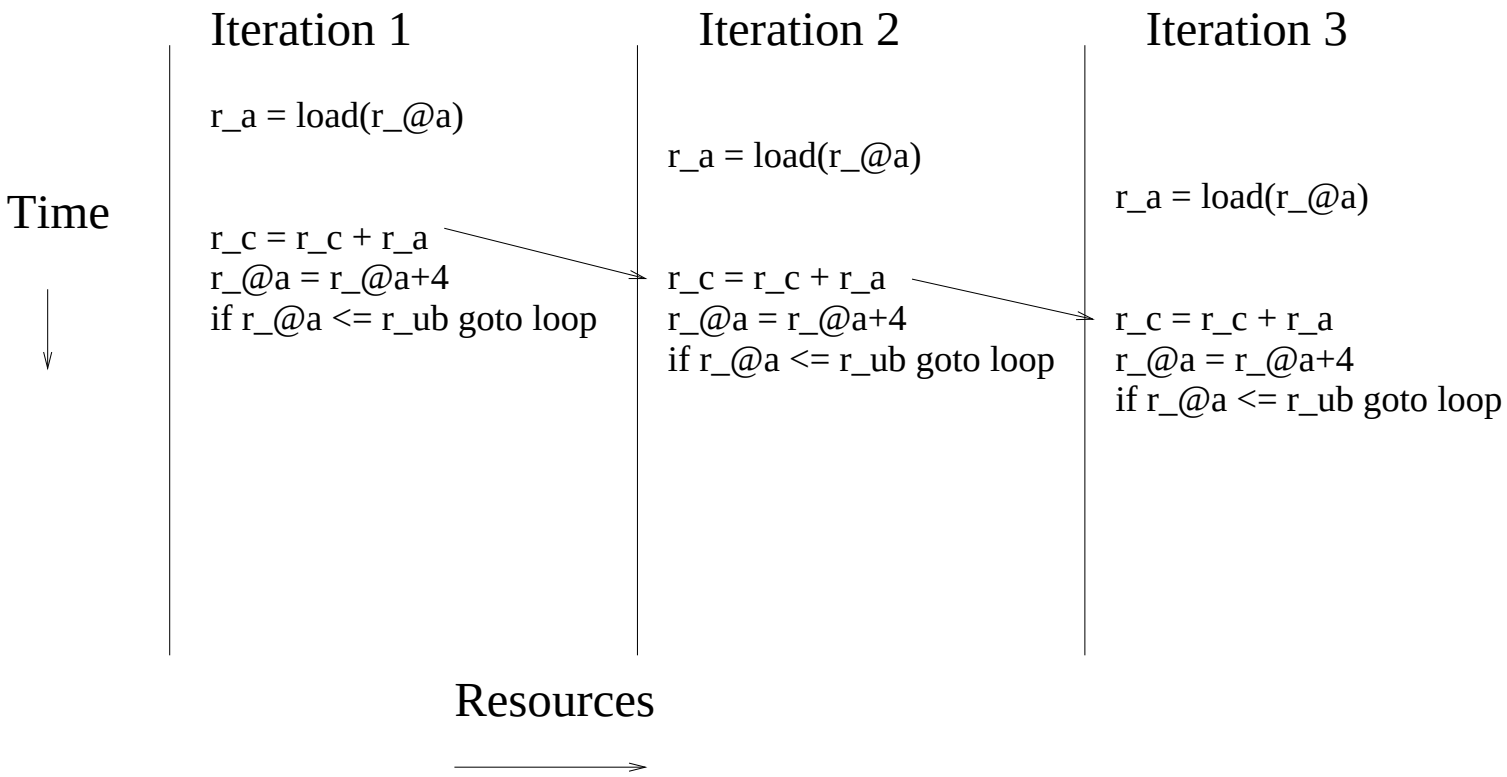
Example

```
c=0
for (i= 1,i <=N,i++)
 c = c+a[i];
```

```
      r_c = 0
      r_@a = @a
      r1 = n*4
      r_ub= r1+r_@a
      if r_@a >r_ub goto exit
loop: r_a = load(r_@a) -- 3 cycle stall
      r_c = r_c + r_a
      r_@a = r_@a +4
      if r_@a <= r_ub goto Loop
exit: store(c)=rc
```

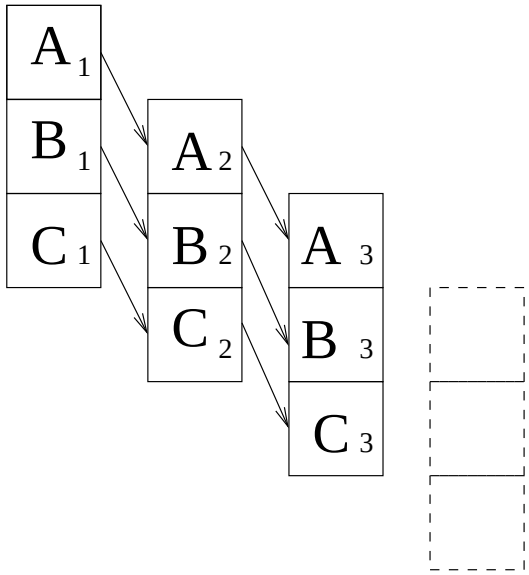
If branches take 1 cycle - each iteration takes 5 cycles after scheduling the loop body. r_@ = can be performed in shadow of load

Iterations can overlapped: Recurrence on r_c shown

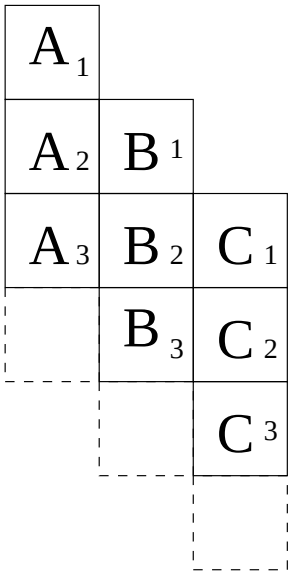


Software piplining

Unbounded Iterations

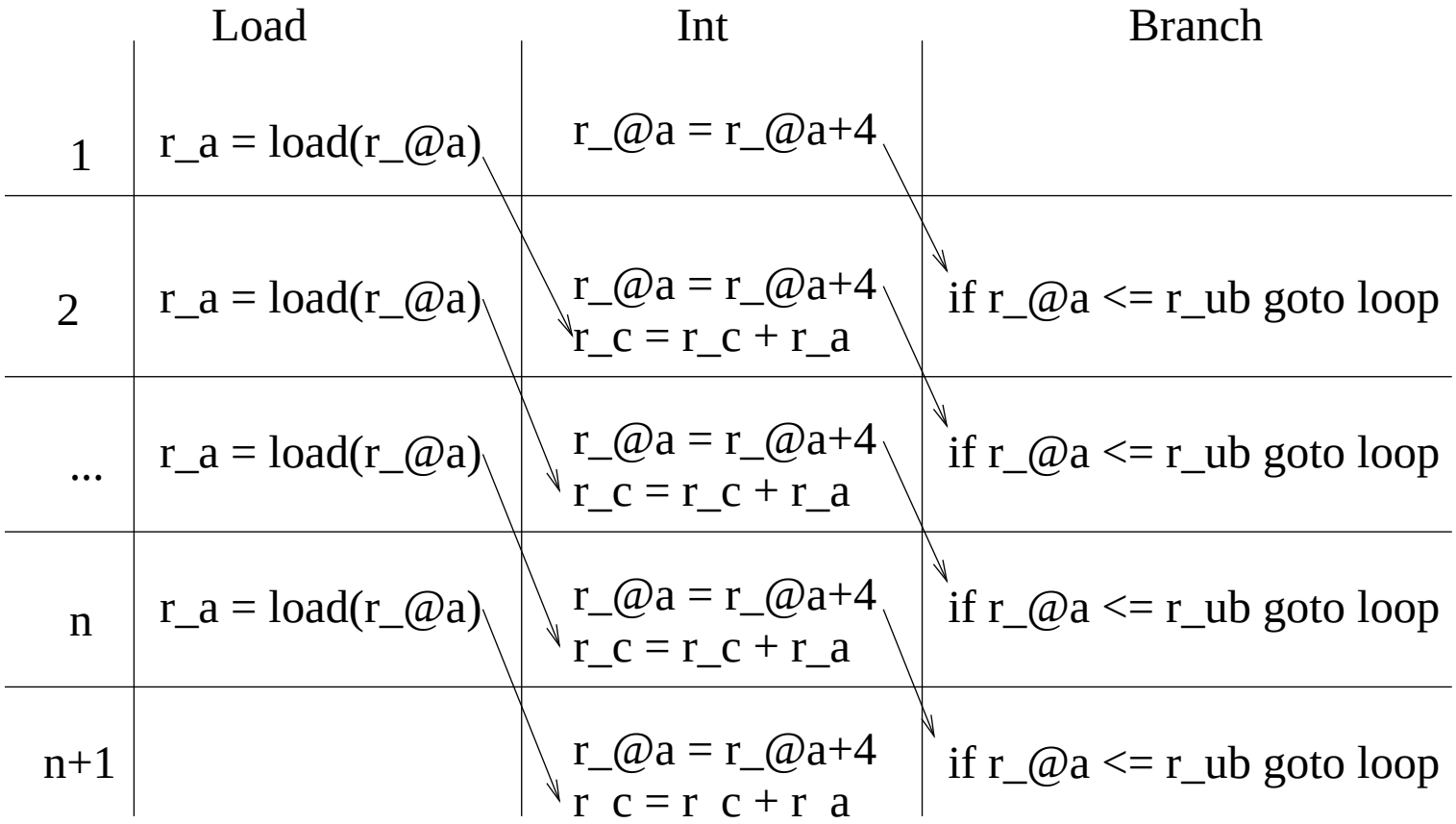


Fixed Resources

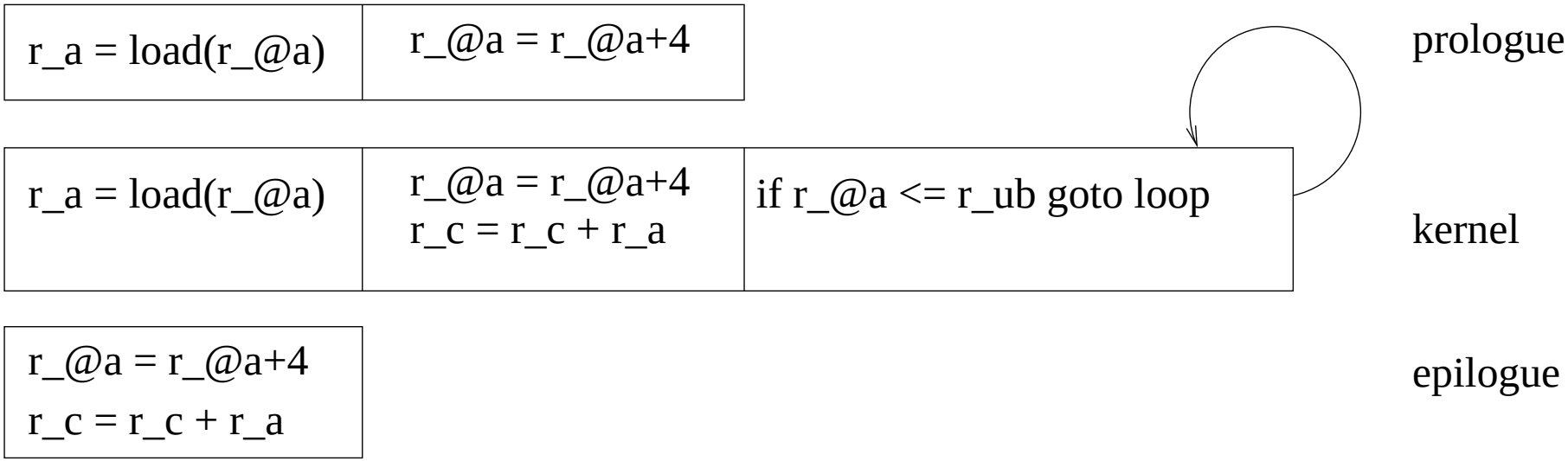


Each unit is reponsble for part of the computation of an iteration. An iteration is pipelined across several units

Pipeline evaluation: Recurrence on r_c not shown



Code template



The schedule must consider function unit type, data dependences and latencies

Assume 3 functional units: Load, Int and Branch and vliw processor Generate this code filling in with noops

Code

	Load Unit	Integer Unit	Branch Unit
	nop	r_@a = @a	nop
	nop	r1 = n * 4	nop
	nop	r_ub = r1 + r_@a	nop
	r_a= load(r_@a)	rc = 0	nop
	nop	r_@a = r_@a + 4	if r_@a >r_ub goto exit
Loop:	r_a =load(r_@a)	r_@a = r_@a + 4	if r_@a >r_ub goto exit
	nop	r_c = r_c + r_a	nop
exit	nop	nop	nop
	nop	r_c = r_c +ra	nop

Respect dependencies and latencies. Inner loop takes just 2 cycles rather than 5

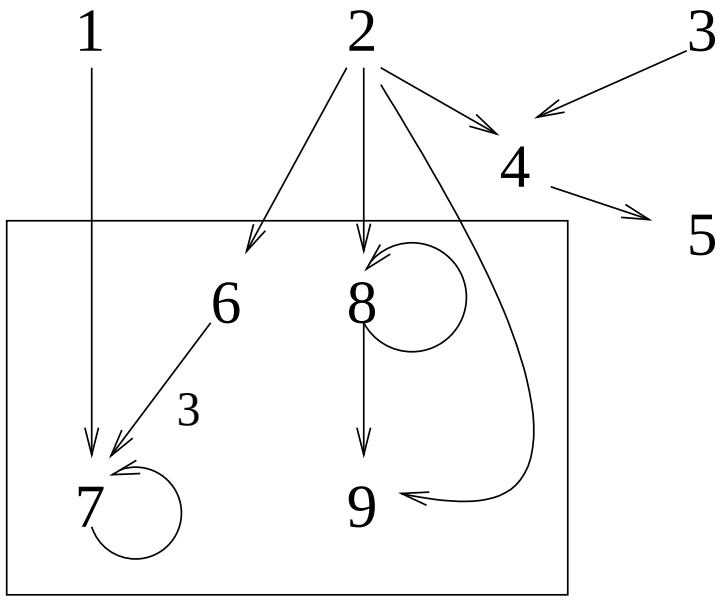
How do we do this automatically?

Applying software pipelining

- calculate an initiation interval - bounded by number of functional units and recurrence distance - smaller ii = smaller loop body =faster
- 2 integer ops, 1 unit, min ii = $\frac{2}{1}$. Recurrences on c delay 1 over 1 iteration so min ii is $\frac{1}{1}$. Combined min ii =2.
- Try scheduling with min ii using modulo scheduling
- If fails try with increased ii
- put in prologue and epilogue code
- May need to put in register copies etc - not considered here

Data Dependence graph and schedule

```
1:          r_c = 0
2:          r_@a = @a
3:          r1 = n*4
4:          r_ub= r1+r_@a
5:          if r_@a >r_ub goto exit
6: loop:    r_a = load(r_@a)
7:          r_c = r_c + r_a
8:          r_@a = r_@a +4
9:          if r_@a <= r_ub goto Loop
10:exit:    store(c)=rc
```



Schedule instructions to units modulo ii. 6 and 8 map into load and integer unit on cycle 0. 9 map into branch on cycle 1. 7 maps into integer on cycle 3 mod 2 = cycle 1.

Current research

- 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 -ie solve the NP-complete problem for basic blocks. Show that it is possible if only used when list scheduling fails
- Despite separation of concerns, code generation and ISA have an impact on scheduling. Cavazos et al PLDI 2004 look at using machine learning to really automate instruction scheduling

Summary

- Dependence graph - dependences between instructions + latency
- Local list Scheduling + critical path
- Superblock and trace scheduling - greater scope for optimisation
- Specialist technique software pipelining
- Calculation of minimum initiation interval
- Modulo Scheduling take into account resources