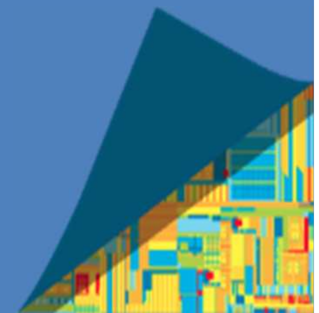
Allocating Rotating Registers by Scheduling

Hongbo Rong
Hyunchul Park
Cheng Wang
Youfeng Wu

Intel Labs

MICRO-46, Dec. 2013



Memory disambiguation in dynamic compilers

- Dynamic languages are increasing popular
  - JavaScript 88.9% in client-side websites (W3Techs)
  - PhP 81.5% in sever side (W3Techs)
- Huge amount of legacy code
  - Dynamic binary translator
  - Optimization scope tends to be small: For a loop, usually a loop iteration
    - Software pipelining: enlarge the scope
  - Memory aliases are a bottleneck to performance

Memory disambiguation in dynamic compilers (Cont.)

- A fundamental component in compilers
  - Determines if two memory operations are aliased
- Approach 1: Alias analysis at compile time
  - Too expensive or conservative for dynamic compilers
  - Only simple alias analysis can be done
- Approach 2: Alias detection at runtime with hardware support

Alias Registers

- Enable data speculation
  - Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  - Compiler guards every memory operation with an alias register to catch any alias when the schedule runs

Alias Registers

- Enable data speculation
  - Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  - Compiler guards every memory operation with an alias register to catch any alias when the schedule runs
- Example:

Alias Registers

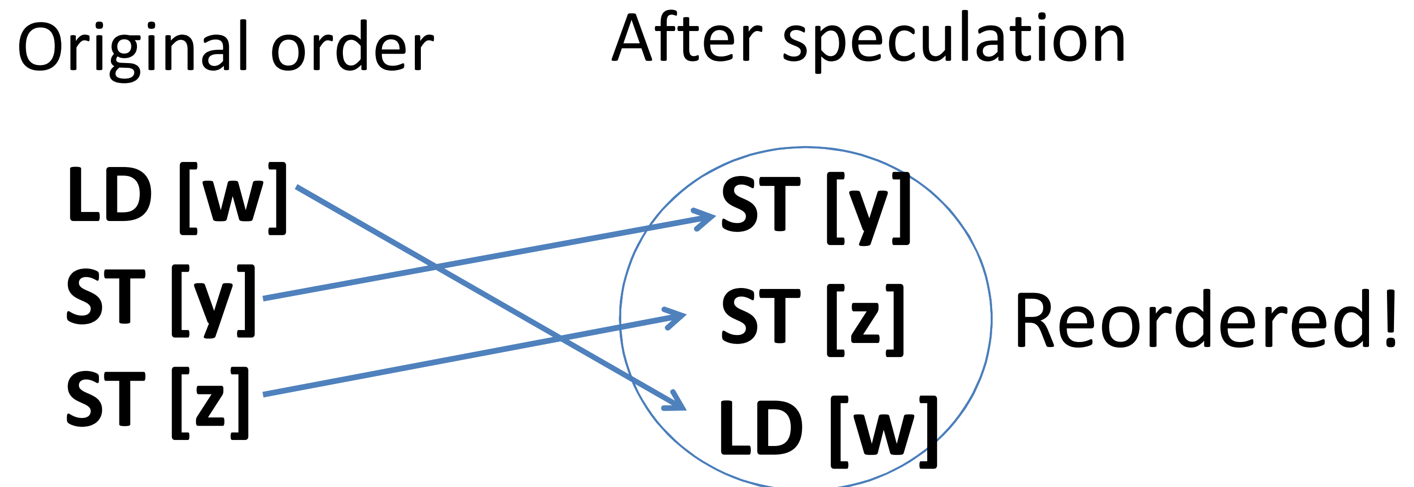
- Enable data speculation
  - Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  - Compiler guards every memory operation with an alias register to catch any alias when the schedule runs
- Example:

Original order

LD [w]
ST [y]
ST [z]

Alias Registers

- Enable data speculation
  - Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  - Compiler guards every memory operation with an alias register to catch any alias when the schedule runs
- Example:



Static Alias Registers

Static Alias Registers

SR0

SR1

SR2

SR3

Static Alias Registers

set

check

SR0

SR1

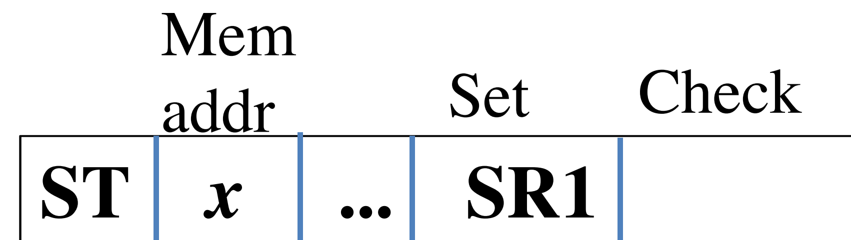
SR2

SR3

Static Alias Registers

set

check



Static Alias Registers

set

check

	Mem addr		Set	Check
ST	***x***	**...**	**SR1**	

SR0

SR1 *x*

SR2

SR3

Static Alias Registers

set

check

SR0

SR1 x

SR2

SR3

ST	Mem addr		Set	Check
ST	x	...	SR1	
ST	y	...	SR2	

Static Alias Registers

set

check

SR0

SR1 x

SR2 y

SR3

ST	Mem addr	...	Set	Check
ST	x	...	SR1	
ST	y	...	SR2	

Static Alias Registers

ST	x	...	SR1	
ST	y	...	SR2	
LD	z	...		SR1,2

Static Alias Registers

set

check

SR0

SR1 *x*

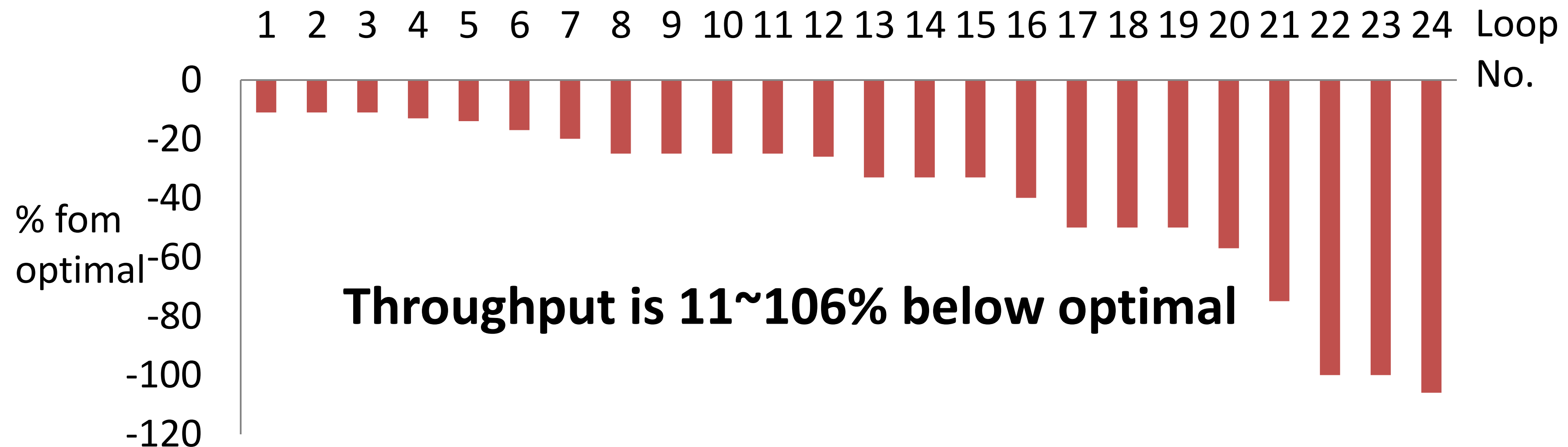
SR2 *y*

SR3

	Mem addr		Set	Check
ST	*x*	...	SR1	
ST	*y*	...	SR2	
LD	*z*	...		SR1,2

Static alias registers are not enough

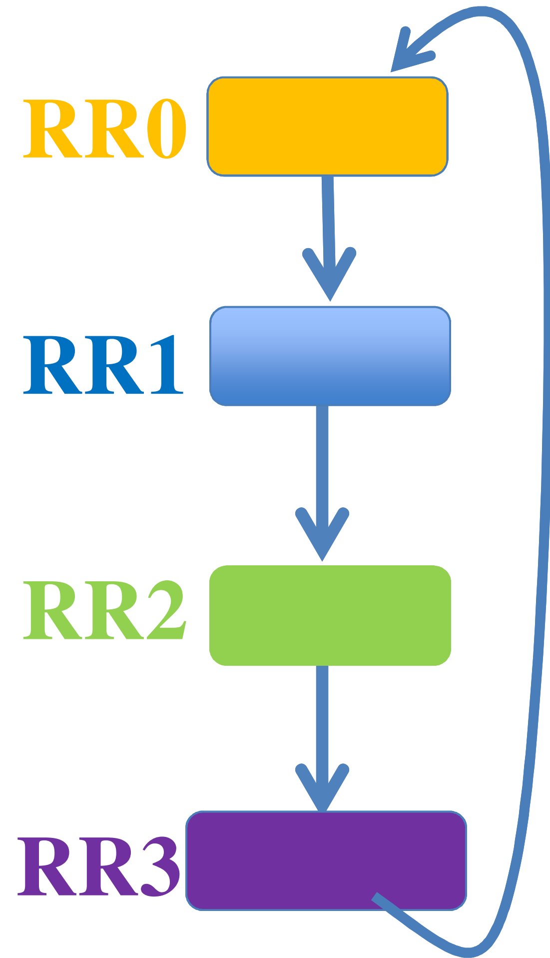
- 24 hot loops from SPEC2000, with static alias registers only, loop throughput is far below optimal
- Need more scalable hardware



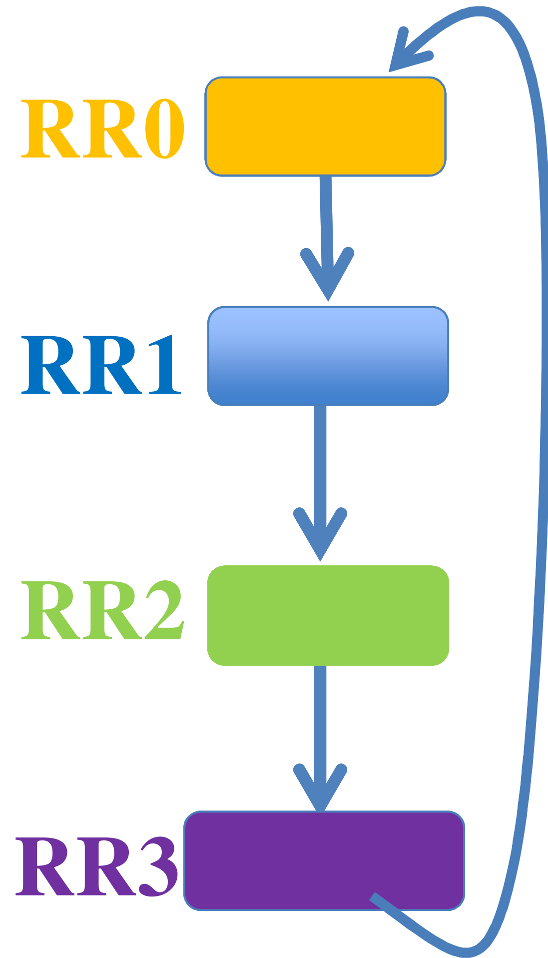
Experiments on Transmeta Efficeon with 14 static alias regs

Rotating Alias Registers

set
check



Rotating Alias Registers



	Mem addr		Set	Check
ST	***x***	**...**	**RR1**	

Rotating Alias Registers

Rotating Alias Registers

Rotating Alias Registers

Rotating Alias Registers

Rotating Alias Registers

set
check

RR0	
RR1	x
RR2	y
RR3	

	Mem addr		Set	Check
ST	x	...	RR1	
ST	y	...	RR2	
LD	z	...		RR1

Rotating Alias Registers

Rotating Alias Registers

- Encode 1 register, check many: scalable

False positives

set
check

RR0

RR1 x

RR2 y

RR3

	Mem addr		Set	Check
ST	x	...	RR1	
ST	y	...	RR2	
LD	z	...		RR1

False positives

False positives

ST	z	...	RR3	
ST	x	...	RR1	
ST	y	...	RR2	
LD	z	...		RR1

False positives

- Avoid by a good allocation
- Resort to static alias registers

False positives

- Avoid by a good allocation
- Resort to static alias registers

False positives

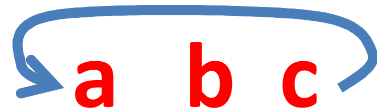
- Avoid by a good allocation
- Resort to static alias registers

Rotating alias registers (Cont.)

- Comparison:
  - ALAT in Itanium does not detect aliases between stores
  - DeAliaser [Ahn, Duan, Torrellas 2013] can only check all speculative stores
- Effective for sequential code [Wang et al. 2012]
- A new problem: How to apply them to loops?

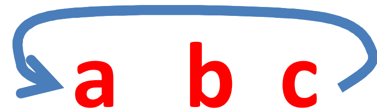
Software Pipelining

- A loop with 3 operations in order



Software Pipelining

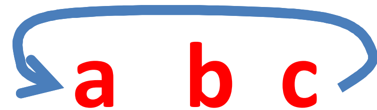
- A loop with 3 operations in order



Sequential execution

Software Pipelining

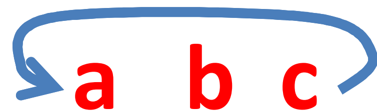
- A loop with 3 operations in order



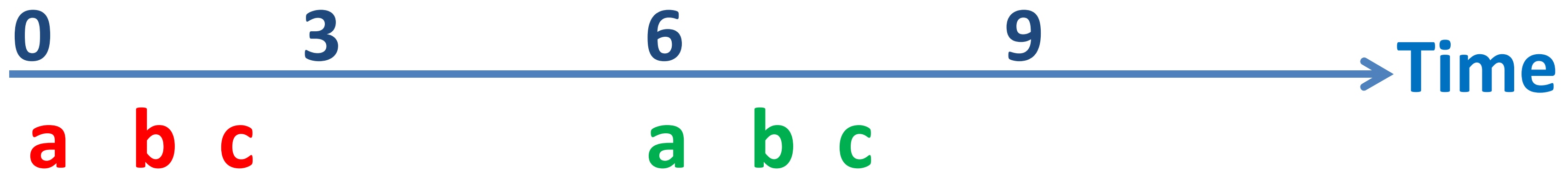
Sequential execution

Software Pipelining

- A loop with 3 operations in order

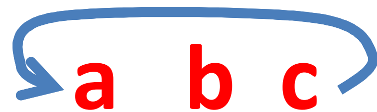


Sequential execution

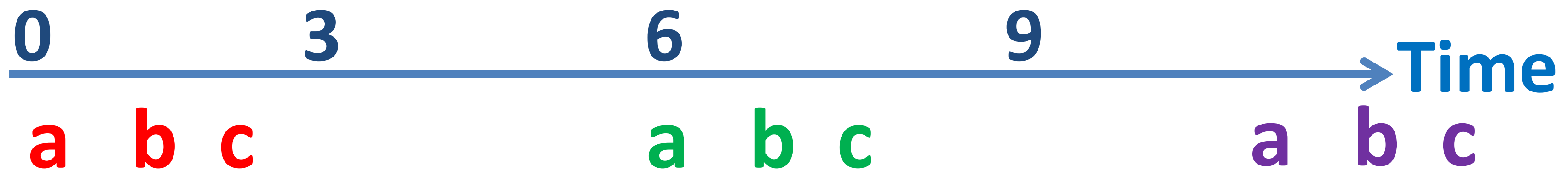


Software Pipelining

- A loop with 3 operations in order

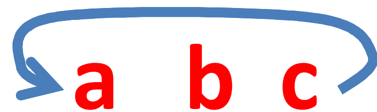


Sequential execution

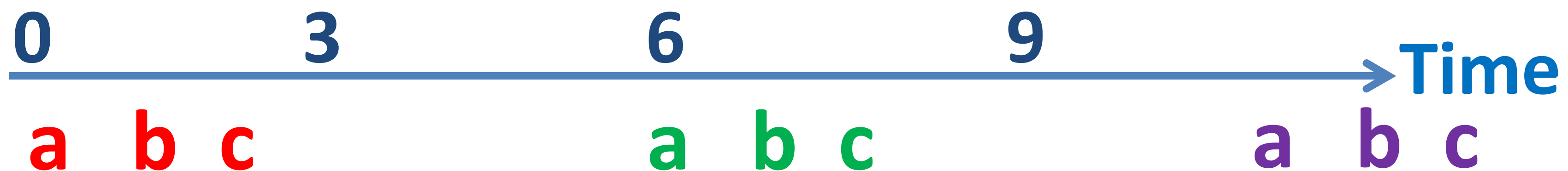


Software Pipelining

- A loop with 3 operations in order



Sequential execution



- To speedup
  - Schedule operations out of order

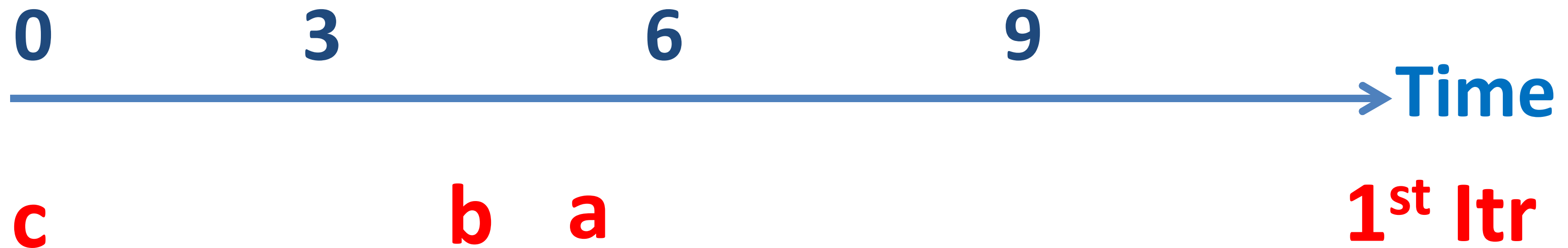
    Overlap the execution of iterations

Software Pipelining (Cont.)

- Software-pipelined execution

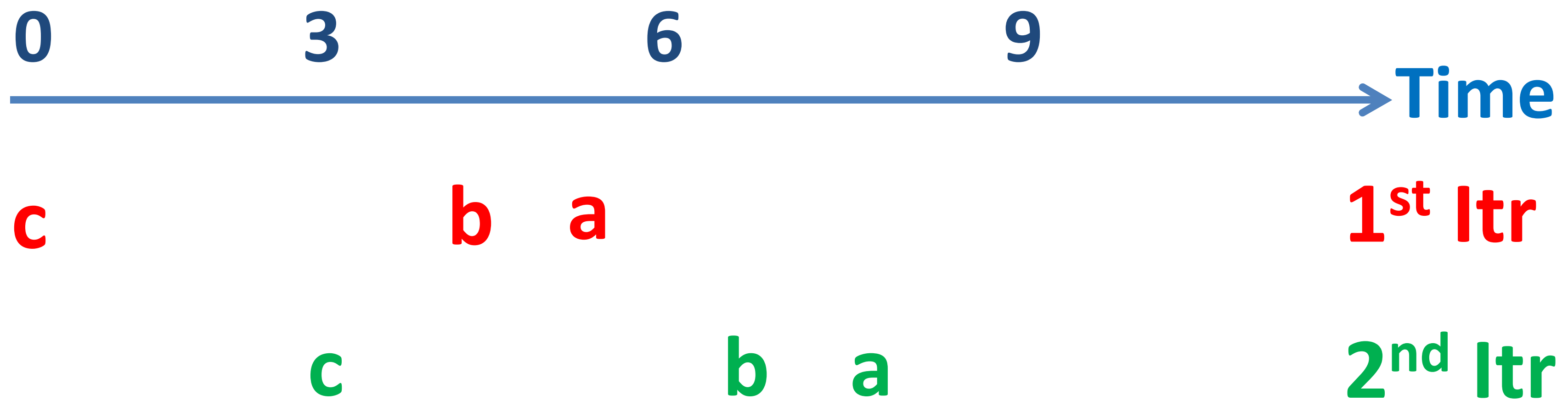
Software Pipelining (Cont.)

- Software-pipelined execution



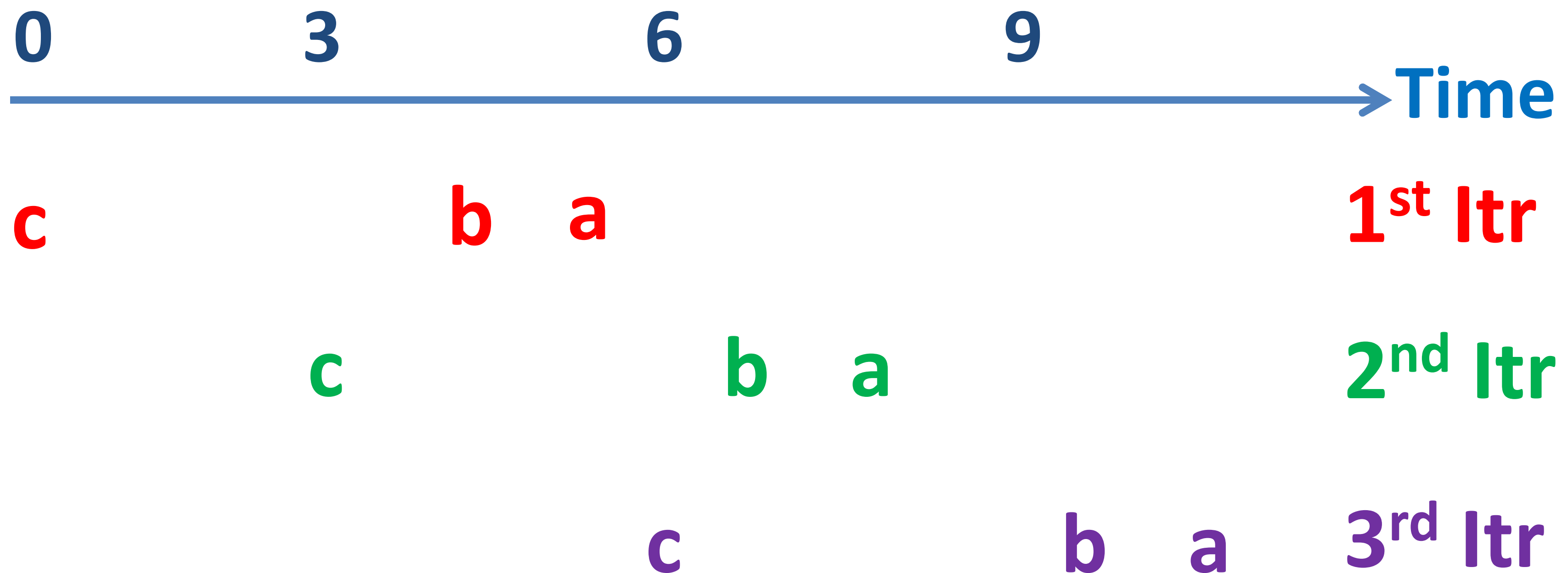
Software Pipelining (Cont.)

- Software-pipelined execution



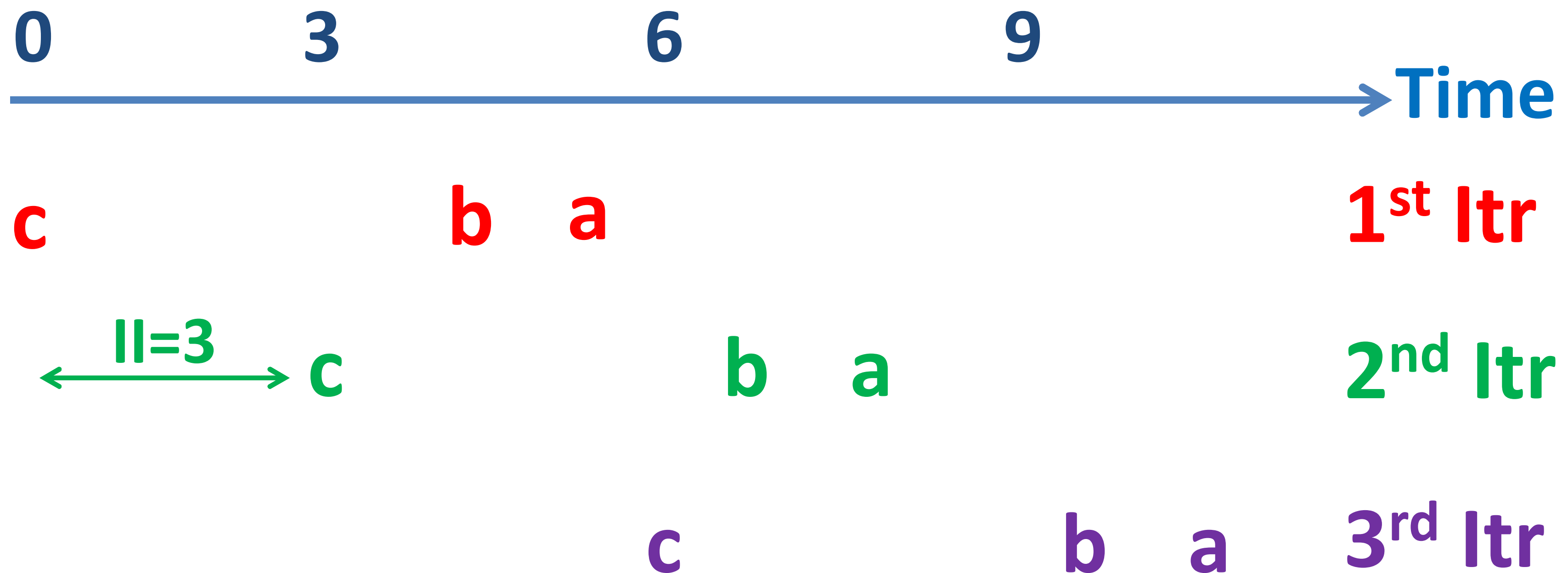
Software Pipelining (Cont.)

- Software-pipelined execution



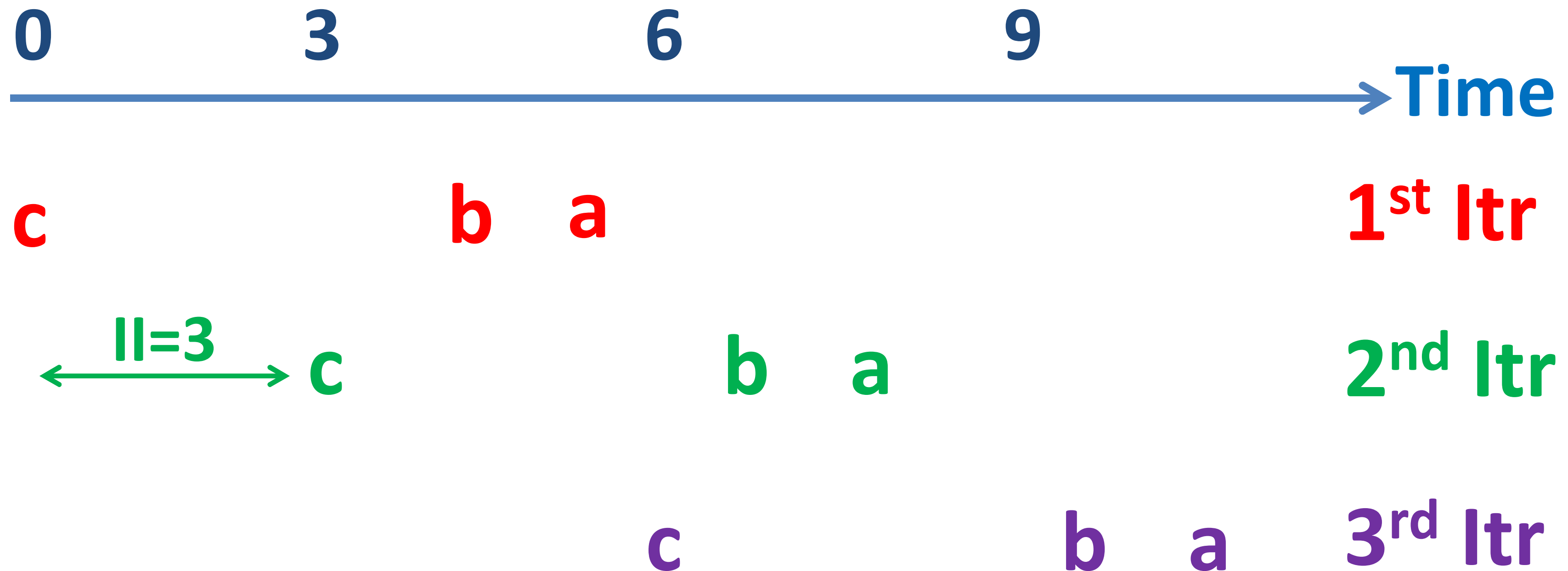
Software Pipelining (Cont.)

- Software-pipelined execution



Software Pipelining (Cont.)

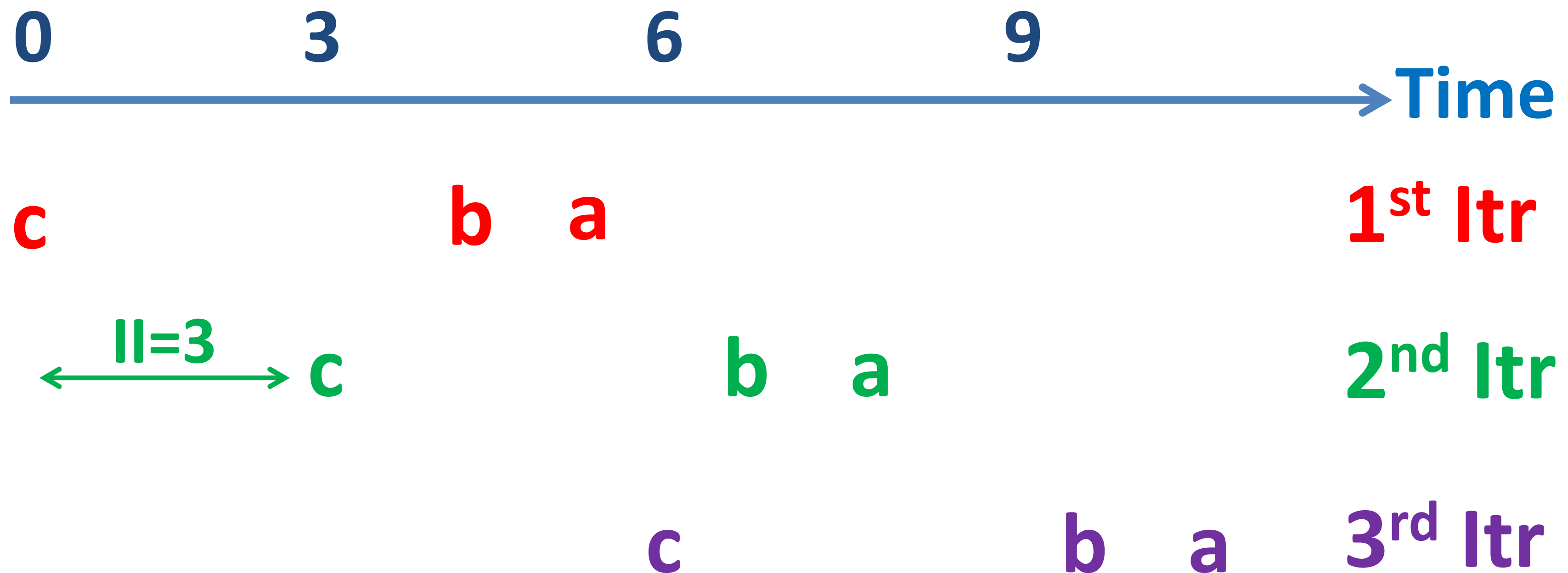
- Software-pipelined execution



- Modulo scheduling
  - Modulo property

Software Pipelining (Cont.)

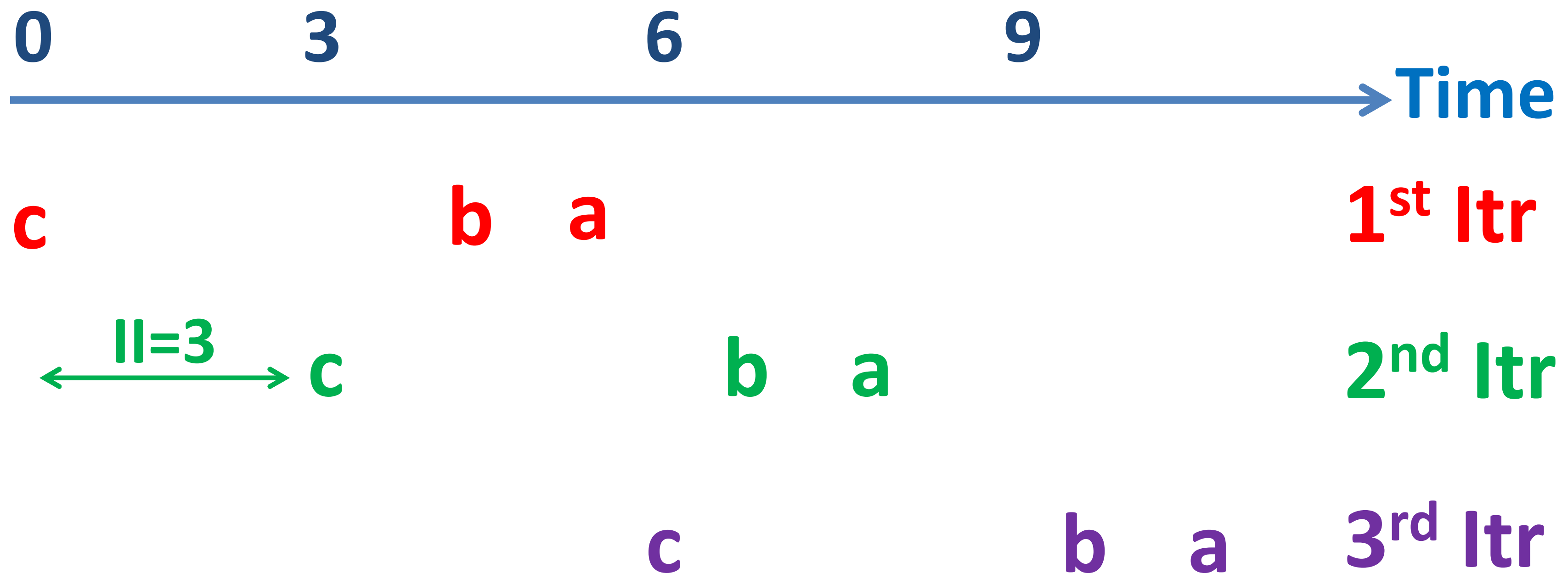
- Software-pipelined execution



- Modulo scheduling
  - Modulo property
  - Dependence constraints

Software Pipelining (Cont.)

- Software-pipelined execution



- Modulo scheduling
  - Modulo property
  - Dependence constraints
  - Resource constraints

Objective

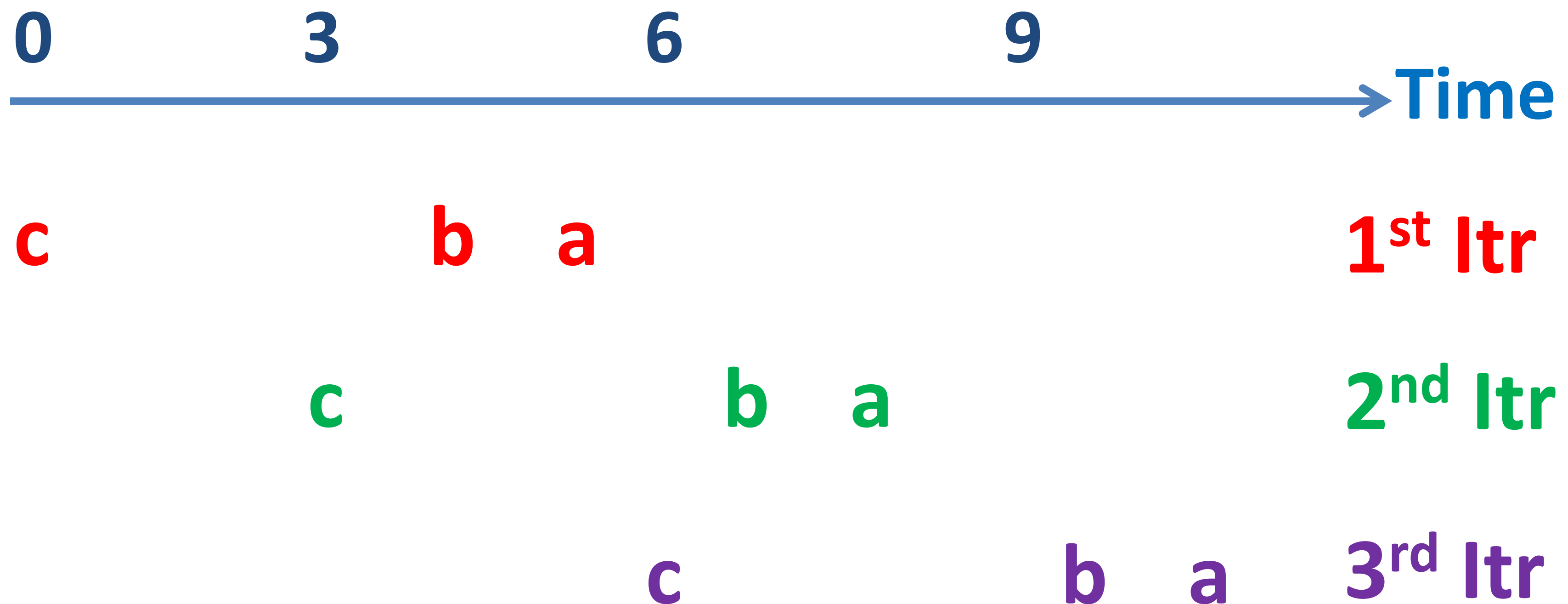
- Given a software-pipelined schedule of a loop, how to allocate rotating alias registers for the memory operations?
  - Detect ALL aliases
  - NO false positive
  - Minimal usage of rotating & static alias registers

Rotating Register Allocation ≡ Scheduling !

- It is a software pipelining problem
  - A register allocation is a modulo schedule of lifetimes
  - Any existing modulo scheduling algorithm can solve it
- Contributions
  - Framework
  - A simple algorithm LCP
  - LCP usually achieves the best allocations in the least time
  - Generalization: allocation of general-purpose rotating registers is also a software pipelining problem
    - It derives bin-packing of Rau et al. 1992

Software Pipelining Reviewed

- Software-pipelined execution



Software Pipelining Reviewed

- Software-pipelined execution

0 3 6 9

Time

c <- - - - - - b a 1st Itr

c b a 2nd Itr

c b a 3rd Itr

<- - - Check

Software Pipelining Reviewed

- Software-pipelined execution

0 3 6 9

Time

c <- - - - - - b a 1st Itr

c b a 2nd Itr

c b a 3rd Itr

<- - - Check

Software Pipelining Reviewed

- Software-pipelined execution

0 3 6 9

Time

c <- - - - - - b a 1st Itr

c b a 2nd Itr

c b a 3rd Itr

<- - - Check

A Register Allocation

0 3 6 9

Time

Register

0: b a

1: c

1st Itr

2: b a

3: c

2nd Itr

4: b a

5: c

3rd Itr

A Register Allocation

A Register Allocation

0 3 6 9

Time

0 b a

1 c

RegII=2

1st Itr

Check

2 b a

3 c

2nd Itr

4 b a

5 c

3rd Itr

Register

A Register Allocation

A Register Allocation

0 3 6 9 "Resource"

0 1 2 3 4 5 "Time"

b a
c
Check
RegII=2
1st Itr
b a
c
2nd Itr
b a
c
3rd Itr

A Register Allocation

0 3 6 9 "Resource"

"Time" 0 1 2 3 4 5

b a
c
1st Itr

RegII=2

Check

b a
c
2nd Itr

b a
c
3rd Itr

Modulo schedule of lifetimes!

Problem formulation

View Lifetimes as "Operations"

Registers as "Time"

Time as "Resources"

Then the allocation is a modulo schedule:

- Modulo property: A constant initiation interval R

$$r(a, i+1) = r(a, i) + R, \quad \forall i$$

- Dependence constraints
  - The ordering requirement between lifetimes

$$r(a, i) \leq r(b, i+d) \quad \forall i$$

- Resource constraints
  - Two lifetimes in the same register cannot overlap in time

Unifying Dependence and Resource Constraints

$$DIST(a,b) = DIST_{dep}(a,b) \bigcap DIST_{res}(a,b)$$

$$\begin{aligned} DIST_{dep}(a,b) &= \bigcap_{\forall \text{dependence } (a \to b, \delta, d)} [\delta - d * R, +\infty) \\ &\bigcap \bigcap_{\forall \text{dependence } (b \to a, \delta, d)} (-\infty, -\delta + d * R] \end{aligned}$$

$$DIST_{res}(a,b) = \begin{cases} DIST_{res_1}(a,b) & \text{if } a \text{ or } b \text{ is a pure checker} \\ DIST_{res_2}(a,b) \bigcup DIST_{res_3}(a,b) & \text{otherwise} \end{cases}$$

$$DIST_{res_1}(a,b) = (-\infty, +\infty)$$

$$DIST_{res_2}(a,b) = (-\infty, +\infty) \setminus R * (-\infty, +\infty)$$

$$\begin{aligned} DIST_{res_3}(a,b) = \quad & R * (-\infty, +\infty) \bigcap \\ & \{[-\left\lfloor \frac{start(a) - end(b)}{II} \right\rfloor * R, +\infty) \bigcup \\ & (-\infty, -\left\lceil \frac{end(a) - start(b)}{II} \right\rceil * R]\} \end{aligned}$$

LCP (Local Compaction followed by Pacing)

0 3 6 9

Time

Register	
0	b a
1	c
2	b a
3	c
4	b a
5	c

LCP (Local Compaction followed by Pacing)

0 3 6 9 Time

0 1 2 3 4 5 Register

b a

c

b a

RegII=1

c

b a

c

LCP (Local Compaction followed by Pacing)

0 3 6 9 Time

0 1 2 3 4 5 Register

b a

c

RegII=1

b a

c

b a

c

- Two heuristics to reduce false positives
  - Max R and Earliest start

Overall flow

Overall flow

Loop

Modulo scheduling (operations)

JITSP(CGO'14)

Overall flow

Loop

Modulo scheduling (operations) — JITSP(CGO'14)

Rotating alias register allocation — JITSP, LCP, RS2, DESP, Ideal

Overall flow

Loop

↓

Modulo scheduling (operations)

JITSP(CGO'14)

↓

Rotating alias register allocation

JITSP, LCP, RS2, DESP, Ideal

↓

Post-processing false positives

Overall flow

Loop

↓

Modulo scheduling (operations) — JITSP(CGO'14)

↓

Rotating alias register allocation — JITSP, LCP, RS2, DESP, Ideal

↓

Post-processing false positives

↓

Code generation

Overall flow

Loop

↓

Modulo scheduling (operations) — JITSP(CGO'14)

↓

Rotating alias register allocation — JITSP, LCP, RS2, DESP, Ideal

↓

Post-processing false positives

↓

Code generation

- Implemented in Transmeta Code Morphing Software (CMS)
- simulated with a variable-size rotating alias register file

% loops allocated successfully

100
80
60
40
20
0
%loops
Ideal
LCP
RS2
DESP
JITSP
0 16 32 48 64 80 96 112 128
#rotating alias registers

- 11,825 software-pipelined loops from SPEC2000 dynamic traces

#false positives and allocation time

- LCP has 0.16 false positives per loop iteration
- LCP takes 2.46% translation time, roughly about 0.07% total time
- #false positives and allocation time relative to LCP

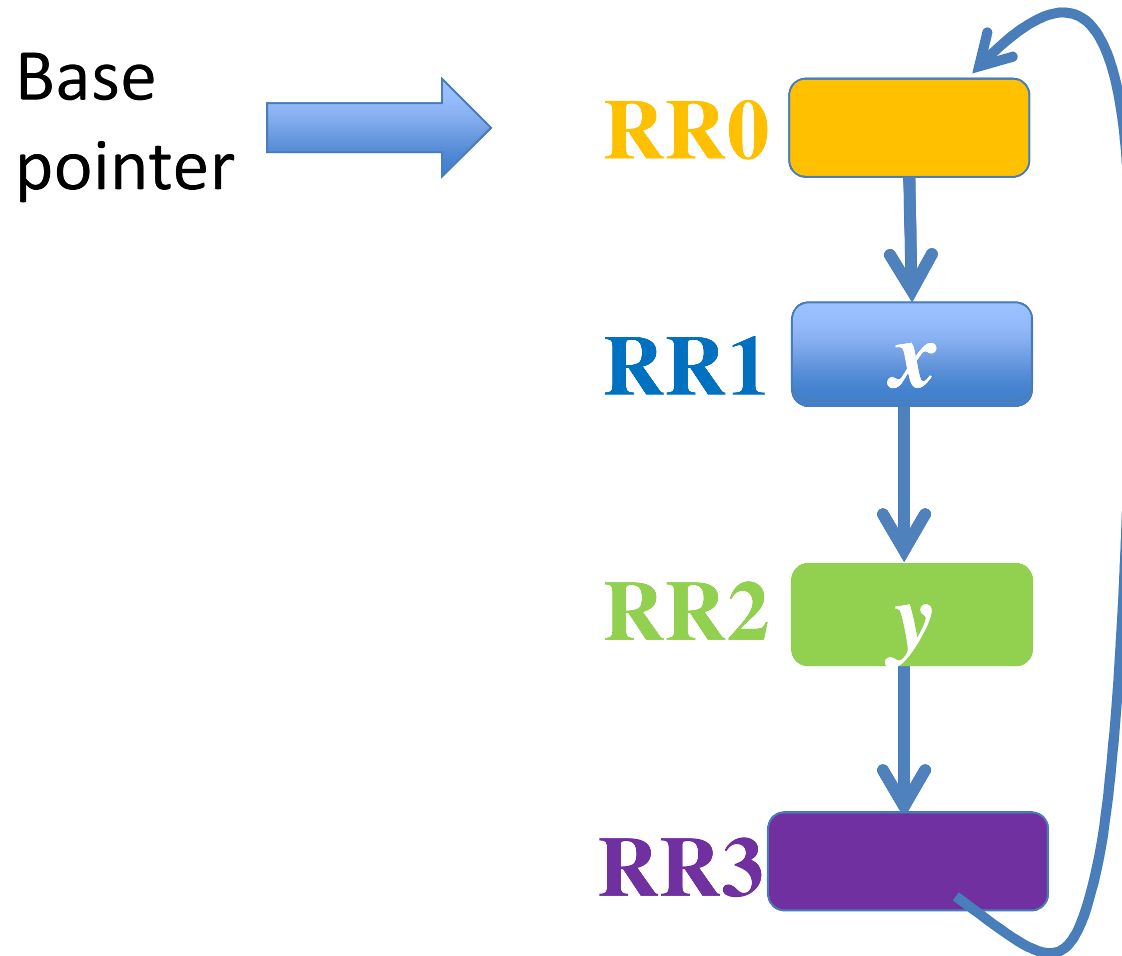
	#false positive	Allocation time
RS2	19X	4X
DESP	12X	1.6X
JITSP	14X	1.7X

Summary

- Rotating alias registers detect memory aliases at runtime
  - Relatively new hardware
- Converting the allocation problem to scheduling problem
  - Software pipelining scheduling has been studied for 3 decades
  - Benefit from reusing well-known/stable algorithms
  - Not limited to software pipelined loops: non-pipelined loops can be treated as pipelined ones with only one stage
- Proposed LCP algorithm
  - Simple and fast
- Extended to Itanium general rotating registers
  - RegII ≡ 1
  - Resource constraints

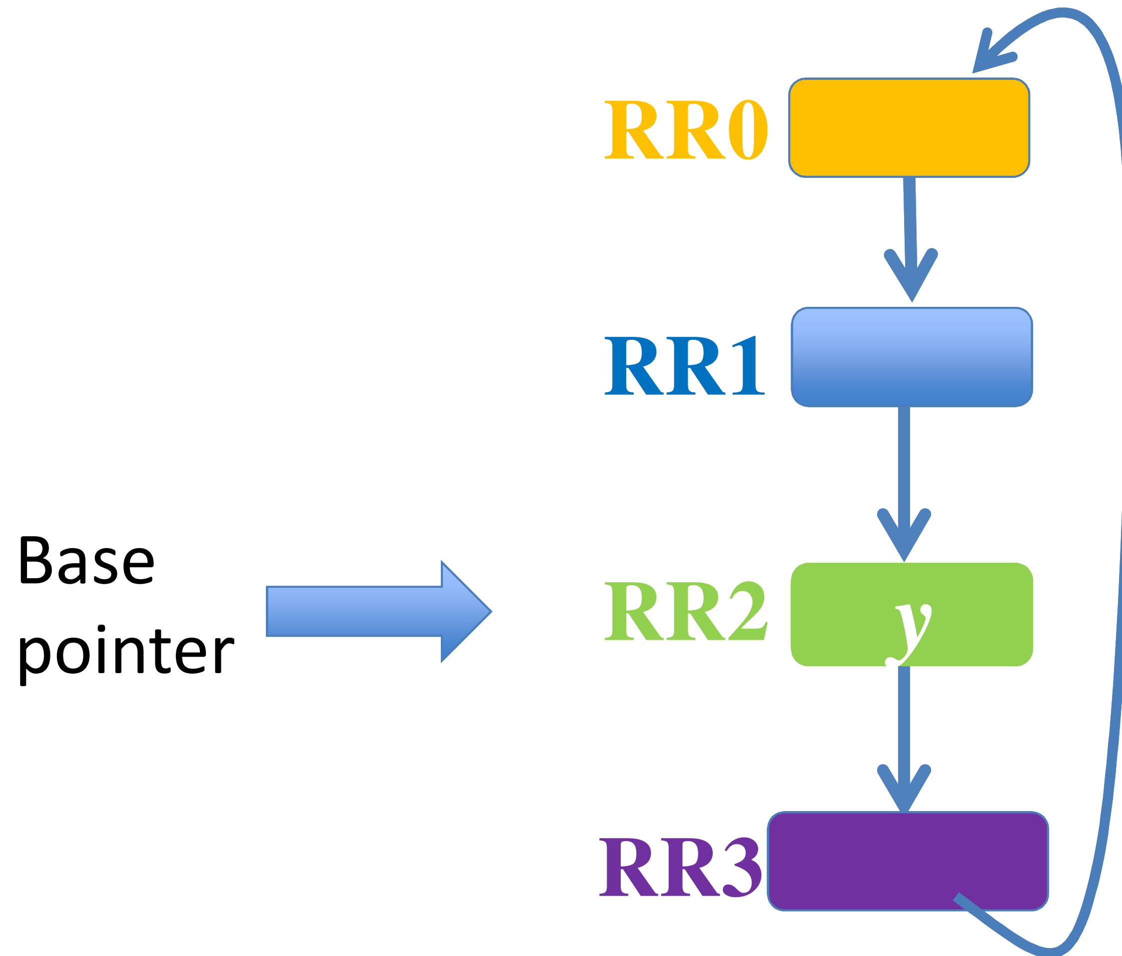
BACKUP

Rotation



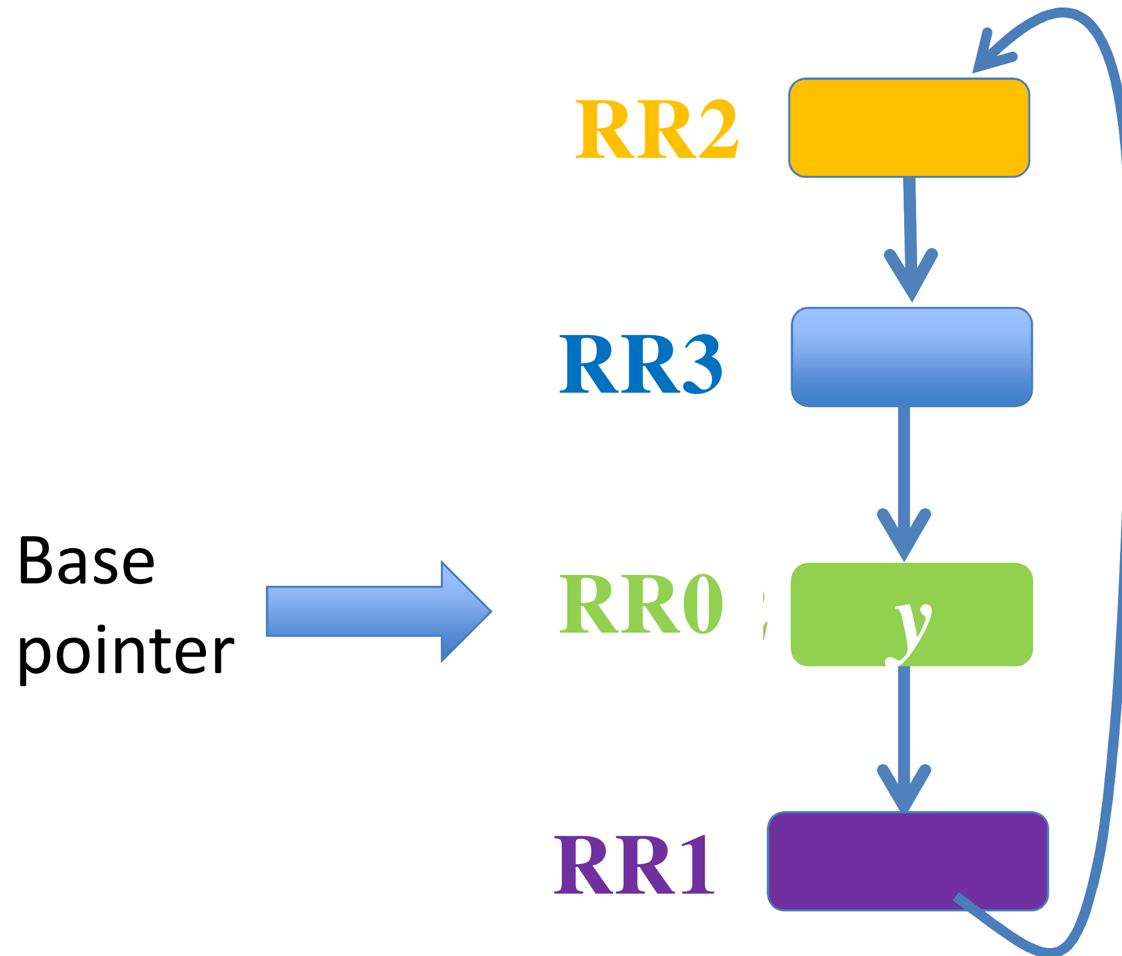
Demonstration: Rotation size 2

Rotation



Demonstration: Rotation size 2

Rotation



Demonstration: Rotation size 2

Register assignment

Rotate
Rotate
Rotate
0
3
6
9
Time
0
2
1
0
2
1
0
2
1

Rotate RegII registers every II time steps