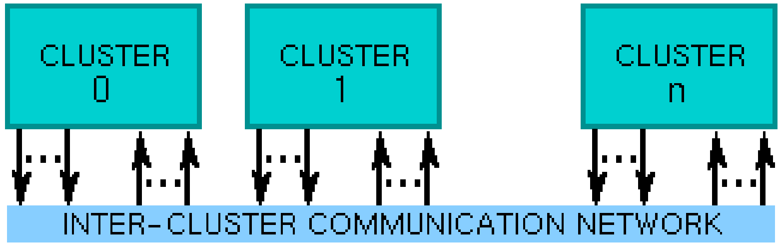
CARS: A New Code Generation Framework for Clustered ILP Processors

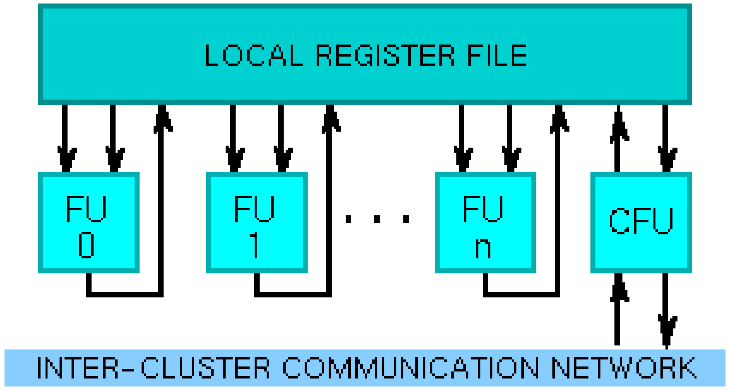
Krishnan Kailas
Ashok Agrawala
University of Maryland
College Park, MD

Kemal Ebcioglu
T. J. Watson Research Center
Yorktown Heights, NY

Clustered ILP Processors

- basic idea
  - partition resources into smaller units
  - group them into clusters



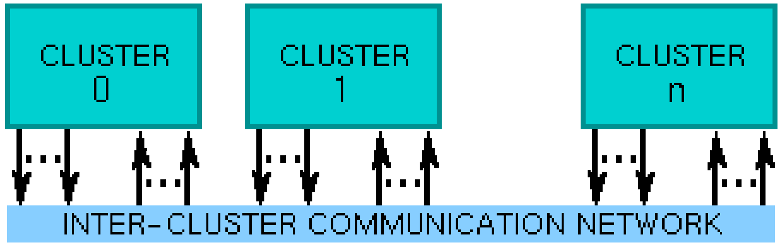


An Individual Cluster

FU Function Unit
CFU Communication Function Unit

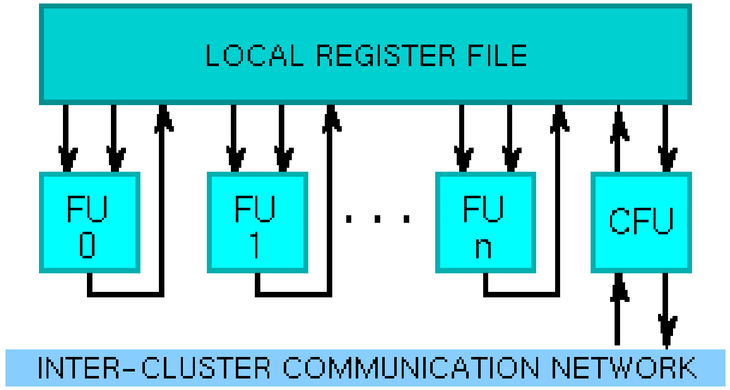
Clustered ILP Processors

- **basic idea**
  - **partition resources into smaller units**
  - **group them into clusters**



- shared bus interconnect
- partitioned register file
- **`rcopy`** operation

- **clusters**
  - **small local register file and cache**
  - **less number of ports**
  - **short wires within clusters**
    - ➢ **fast processing in clusters**
  - **slowing down the entire chip**



An Individual Cluster

FU Function Unit
CFU Communication Function Unit

vs. inter-cluster communication > 1 cycle

Clustering

- **longer wires for inter-cluster communication**
- **wire delays + finite communication bandwidth**
  - ➢ **inter-cluster communication expensive**

Clustering

- longer wires for inter-cluster communication
- wire delays + finite communication bandwidth
  - ➢ inter-cluster communication expensive

- solution:
  - – explicitly handle inter-cluster communication
    - minimize inter-cluster communication
    - hide inter-cluster communication latency
  - ➢ cluster scheduling

Main Contributions

- a new cluster scheduling framework – CARS
- combined reg allocation, cluster assignment and scheduling
- eliminates phase ordering problem
- fast on-the-fly register allocation and spilling
- out-of-order scheduling
- better performance than prior schemes

Outline

- code generation for clustered processors
- phase ordering problem
- CARS
- on-the-fly register allocation and spilling
- results

Code Generation for Clustered ILP Processors

primary goals

1. extract ILP
   – use standard techniques

2. increase function unit utilization
   (spread computation across clusters)
3. reduce inter-cluster communication

conflicting requirements?

Cluster scheduling

- reduce inter-cluster communication
  - exploit locality in space and time
- hide inter-cluster communication latency
  - prevent inter-cluster communication operations from stretching the critical path

Cluster scheduling

- register file access delay
  - function of spatial separation of OPs
  - inter-cluster bus availability

Cluster scheduling

- register file access delay
  - function of spatial separation of OPs
  - inter-cluster bus availability
- dependent constraints
  - availability of
    - resources (function units, buses, etc)
    - data (ready cycle)
    - register(s) for results

Code Generation for Clustered ILP Processors

- main tasks
  - instruction scheduling
  - register allocation
  - cluster allocation
    - optimality not a goal

- often carried out in several phases
  - ➢ phase ordering problem

Phase ordering problem

- an old open problem in compiler design
- due to breaking up an optimization problem with dependent constraints
- decisions made in one phase affects next
- caused by assumptions such as…
  - infinite number of resources
  - zero time copy operations

Phase ordering problem

unified processor example:

```
t1 ← A[i]
s  ← s + t1
t2 ← B[i]
p  ← p * t2
```

reg assignment before scheduling

`r0 ← A[r1]`	
`r2 ← r2 + r0`	`r0 ← B[r1]`
`r3 ← r3 * r0`	

3 cycles; 4 registers

Phase ordering problem

unified processor example:

```
t1 ← A[i]
s  ← s + t1
t2 ← B[i]
p  ← p * t2
```

reg assignment before scheduling

`r0 ← A[r1]`	
`r2 ← r2 + r0`	`r0 ← B[r1]`
`r3 ← r3 * r0`	

3 cycles; 4 registers

reg assignment after scheduling

`r0 ← A[r1]`	`r4 ← B[r1]`
`r2 ← r2 + r0`	`r3 ← r3 * r4`

2 cycles; 5 registers

- in a clustered processor ?

Phase ordering problem

- **clustered processor**

  has to deal w/ phase ordering between…

  - **cluster assignment ⇔ instruction scheduling**
  - **cluster assignment ⇔ register allocation**
  - **register allocation ⇔ instruction scheduling**

➢ **can affect performance of clustered processors**

Phase ordering problem
prior work

- lot of research papers on combining register allocation with instr. scheduling…

  [see our paper for a partial list]

- summary:
  - provide hints from one stage to other
  - embed more resource constraints in traditional interference graphs...

Phase ordering problem

prior work

- lot of research papers on combining register allocation with instr. scheduling…
  [see our paper for a partial list]

- summary:
  - provide hints from one stage to other,
  - embed more resource constraints in traditional interference graphs...

➢ **we have a better solution: CARS**

CARS Code Generation Framework

Combined cluster

Assignment,

Register allocation, and instruction

Scheduling

CARS Code Generation Framework

- handles all constraints in single pass
- completely eliminates phase-ordering problem
- uses a single-pass, non-iterative, non-backtracking algorithm
- avoids spilling
- built-in fast on-the-fly register allocation

CARS

- **input:**
  - **DFG (Dependence Flow Graph):**
    - **nodes are OPs, edges represent data/control flow** [Pingali et al. 1991]
  - **enhanced SSA (Static Single Assignment)**

- **pre-CARS initialization**
  - **computes…**
    - **height and depth of nodes**
    - **`use_count` of SSA DEFs**
  - **identifies DEFs with fixed register mappings**

CARS: Top-level Overview

- **partitions DFG into 3 regions:**

- **at each step:**
  - **identifies data-ready nodes and moves them into ready list**
  - **identifies the highest priority node in ready list**
  - ***CARScheduling:* assign cluster, function unit and registers for the highest-priority node**

CARScheduling of an OP

- **modified list-scheduling:**

  If(`earliest_cycle` <= `current_vliw_cycle`)

  schedule OP on current vliw

  else move OP back to readylist

  where `earliest_cycle` =

  `min{schedule_cycle[0..max_clusters]}`

CARScheduling of an OP

- compute `schedule_cycle` of OP in all clusters

CARS Scheduling of an OP

- compute `schedule_cycle` of OP in all clusters
- `schedule_cycle` for a given cluster depends on:
  - availability of function unit
  - vliw cycle in which source nodes are scheduled
  - cluster assignment of source nodes
    - `rcopy` OP is needed to access remote regs
    - is `rcopy` OP already scheduled?
    - if not, what is the earliest cycle `rcopy` OP can be scheduled?
  - availability of dest. register of required type

➢ fully resource-constraint cluster scheduling

CARS heuristics

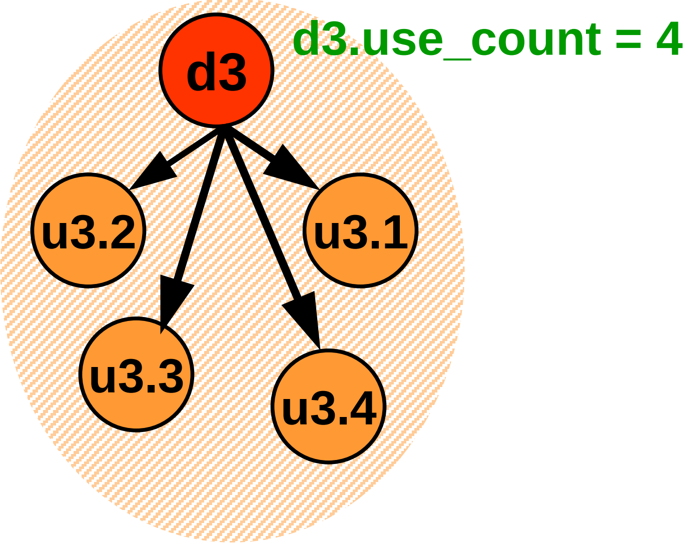
- select a node from ready list based on (infinite resource) height + (scheduled) depth
  - critical paths can change dynamically
  - update depth after scheduling each node
- select a cluster…
  - that does not require `rcopy` OPS
  - with lower register pressure
  - lowest / highest cluster-id
- cluster scheduling based on
  - local and global resource availability
  - critical path at each scheduling step

CARS: on-the-fly register allocation

- global register allocation technique

- make use of…
  - usage counts
  - dynamic register map propagation
  - register bitvectors

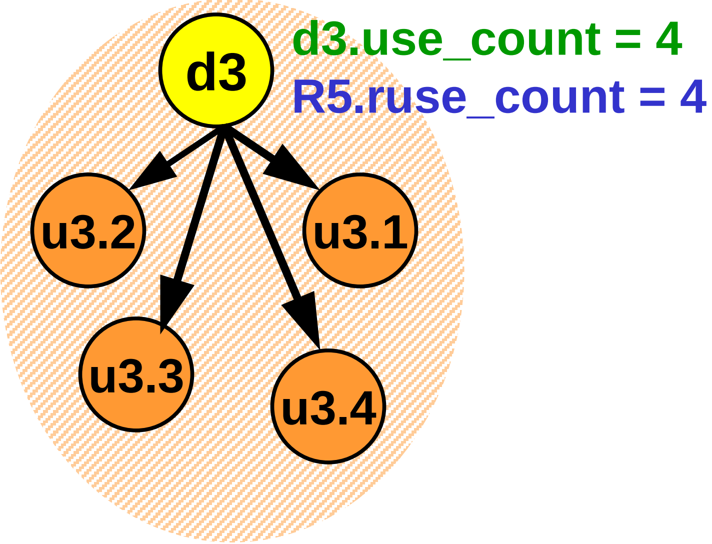
Register allocation using usage counts



- **old idea**

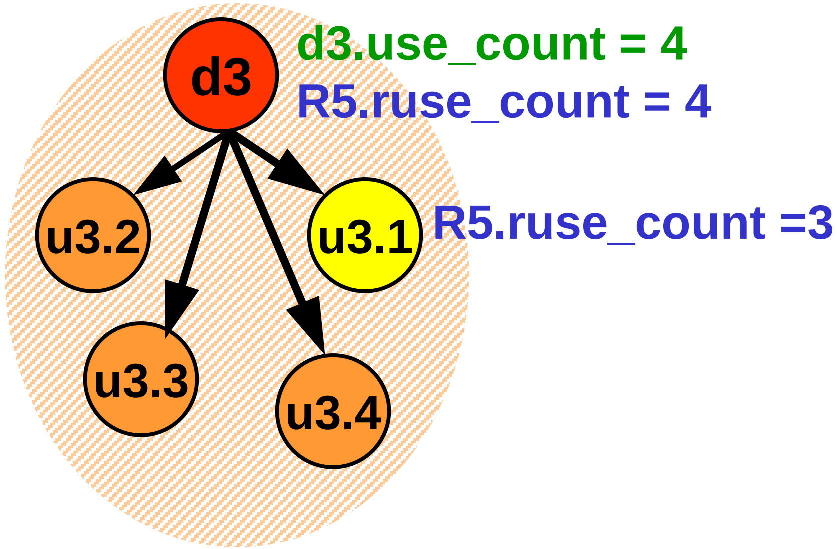
  [Freiburghouse 1974]
- **good for local register allocation**
- **live/dead status based on `ruse_count` of register**

Register allocation using usage counts



- **old idea**
  [Freiburghouse 1974]
- **good for local register allocation**
- **live/dead status based on `ruse_count` of register**

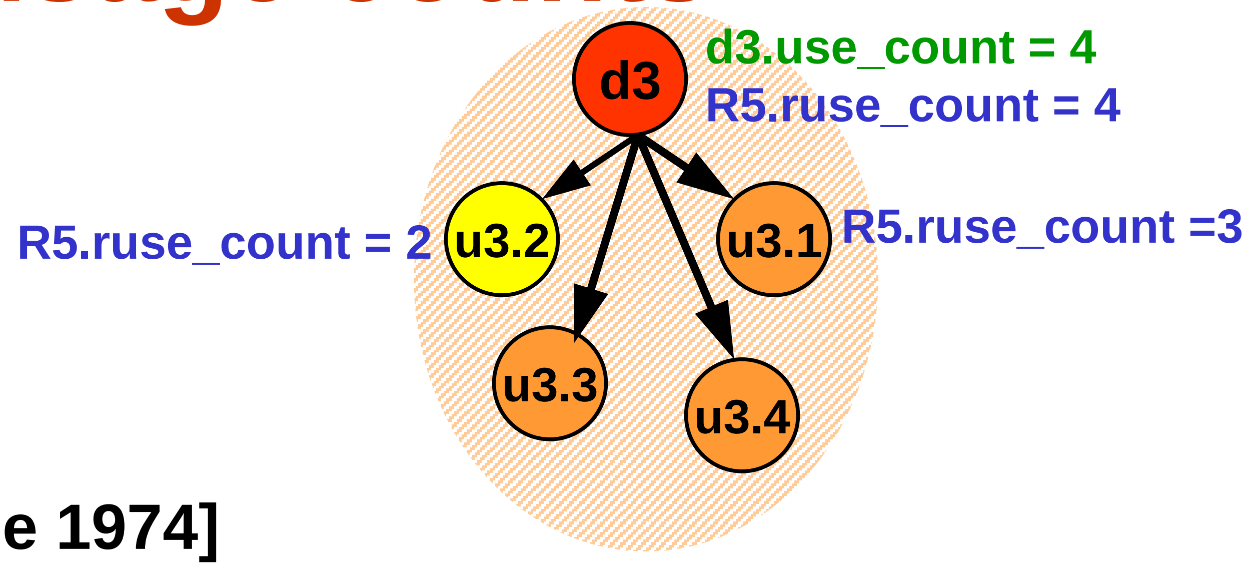
Register allocation using usage counts



- **old idea**

  [Freiburghouse 1974]
- **good for local register allocation**
- **live/dead status based on `ruse_count` of register**

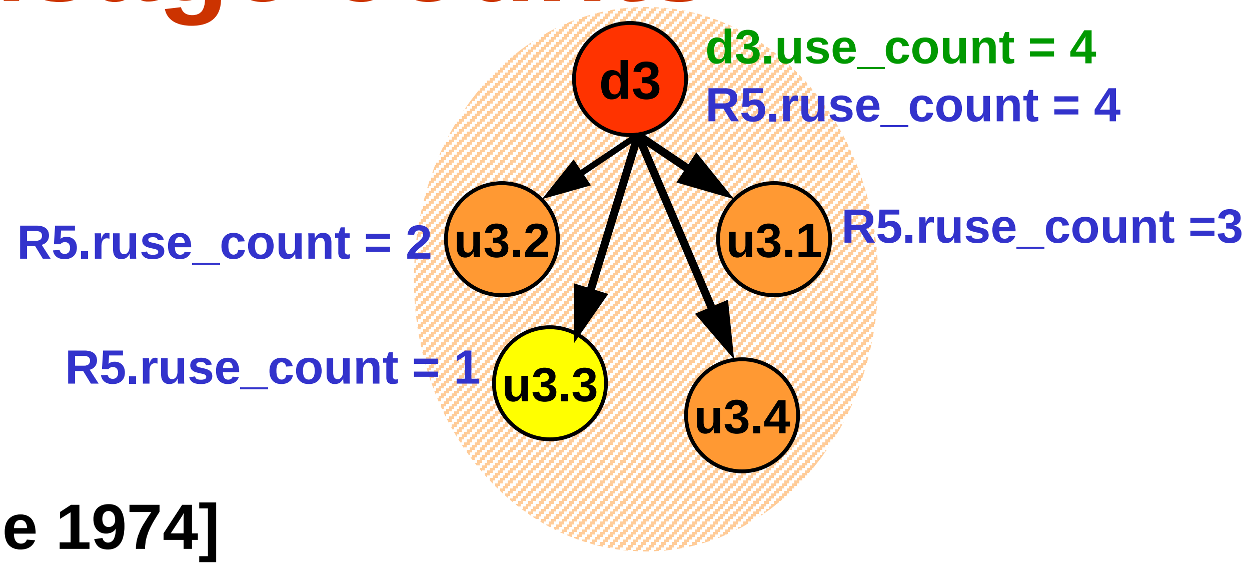
Register allocation using usage counts



- **old idea**

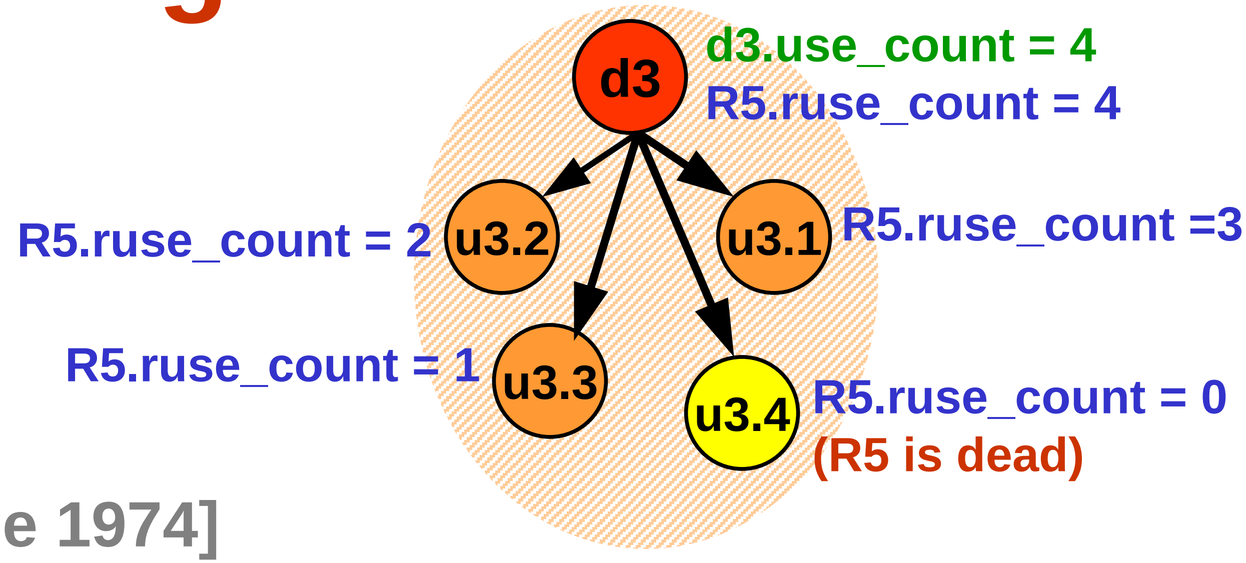
  [Freiburghouse 1974]
- **good for local register allocation**
- **live/dead status based on `ruse_count` of register**

Register allocation using usage counts



- **old idea**
  [Freiburghouse 1974]
- **good for local register allocation**
- **live/dead status based on `ruse_count` of register**

Register allocation using usage counts



- old idea
  [Freiburghouse 1974]
- good for **local register allocation**
- live/dead status based on `ruse_count` of register

➢ **how can we use it for global allocation?**

Global register allocation using usage counts

- **multiple SSA DEFs can be mapped to a single physical register**
  - **need to handle webs of DEF-USE chains**
- **assume no live range info is available**
  - **why?**

➢ **what to do at forks and joins?**

Global register allocation using usage counts

1. first assignment of a DEF:

   `ruse_count += use_count`

2. after scheduling each USE of DEF:

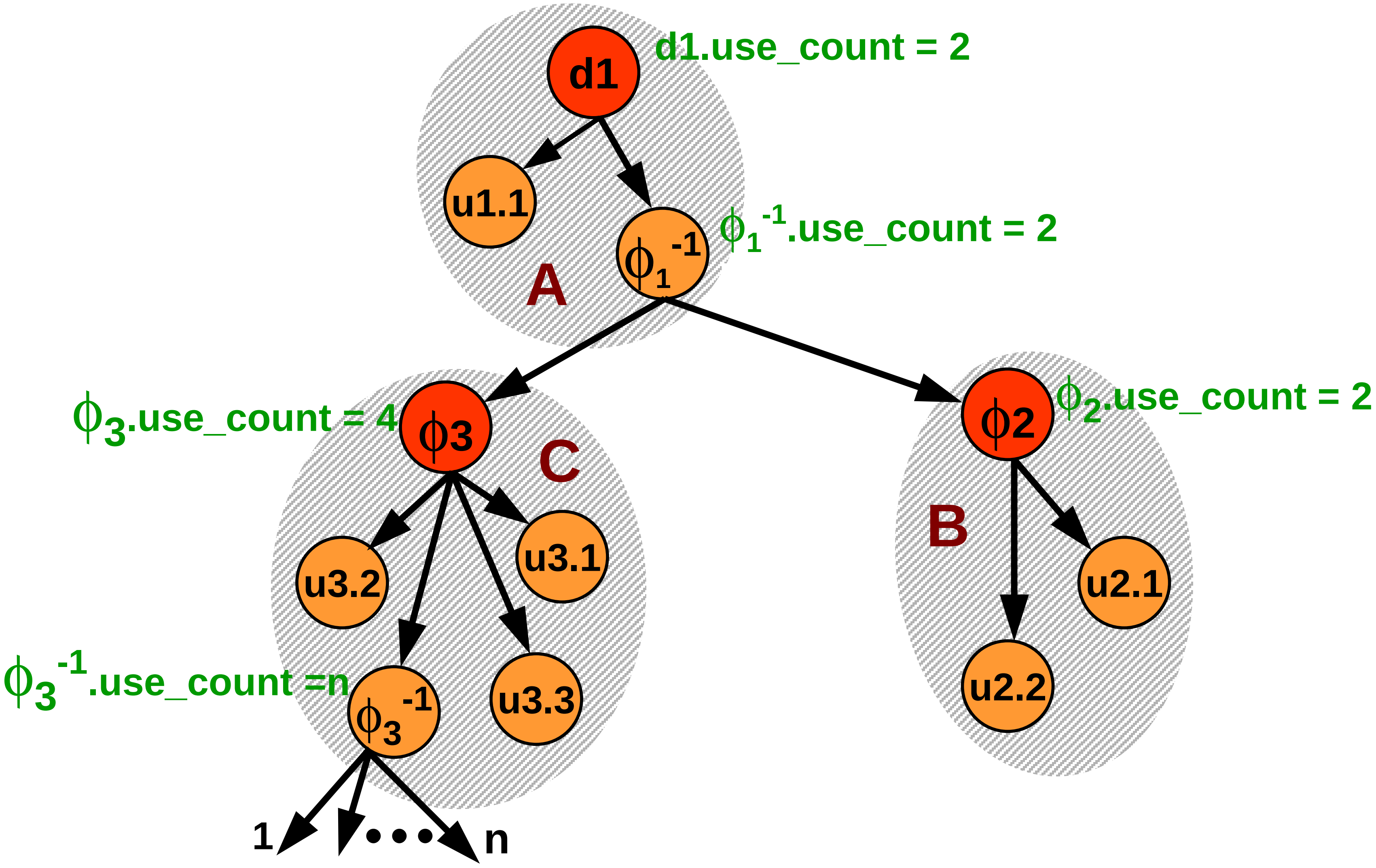
   `ruse_count--`

3. register is dead when `ruse-count == 0`
4. after scheduling fork:

   `ruse_count +=` # of unvisited join ϕs

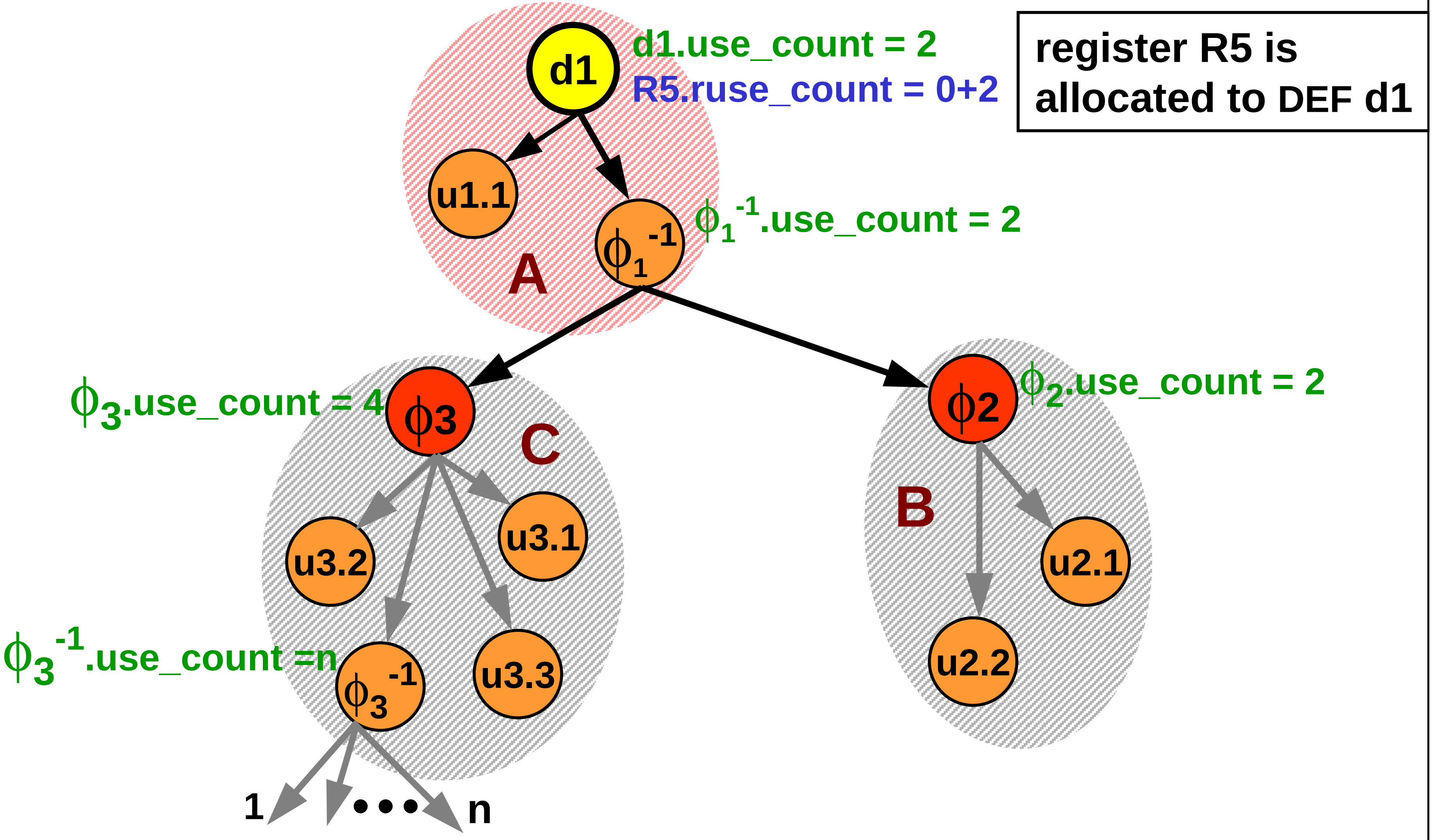
regions are scheduled in topological order

On-the-fly register allocation



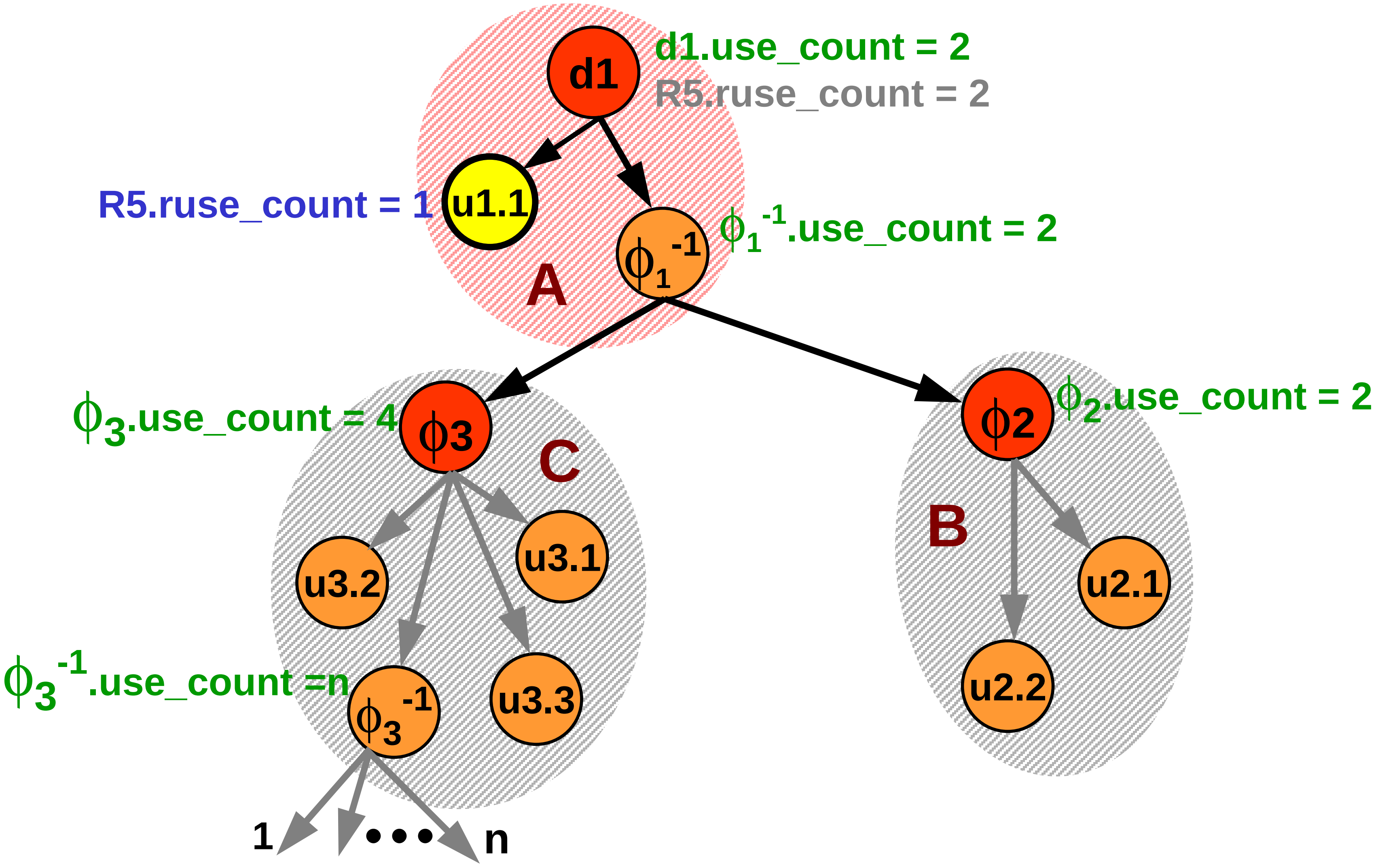
use_count computed in pre-CARS pass

On-the-fly register allocation



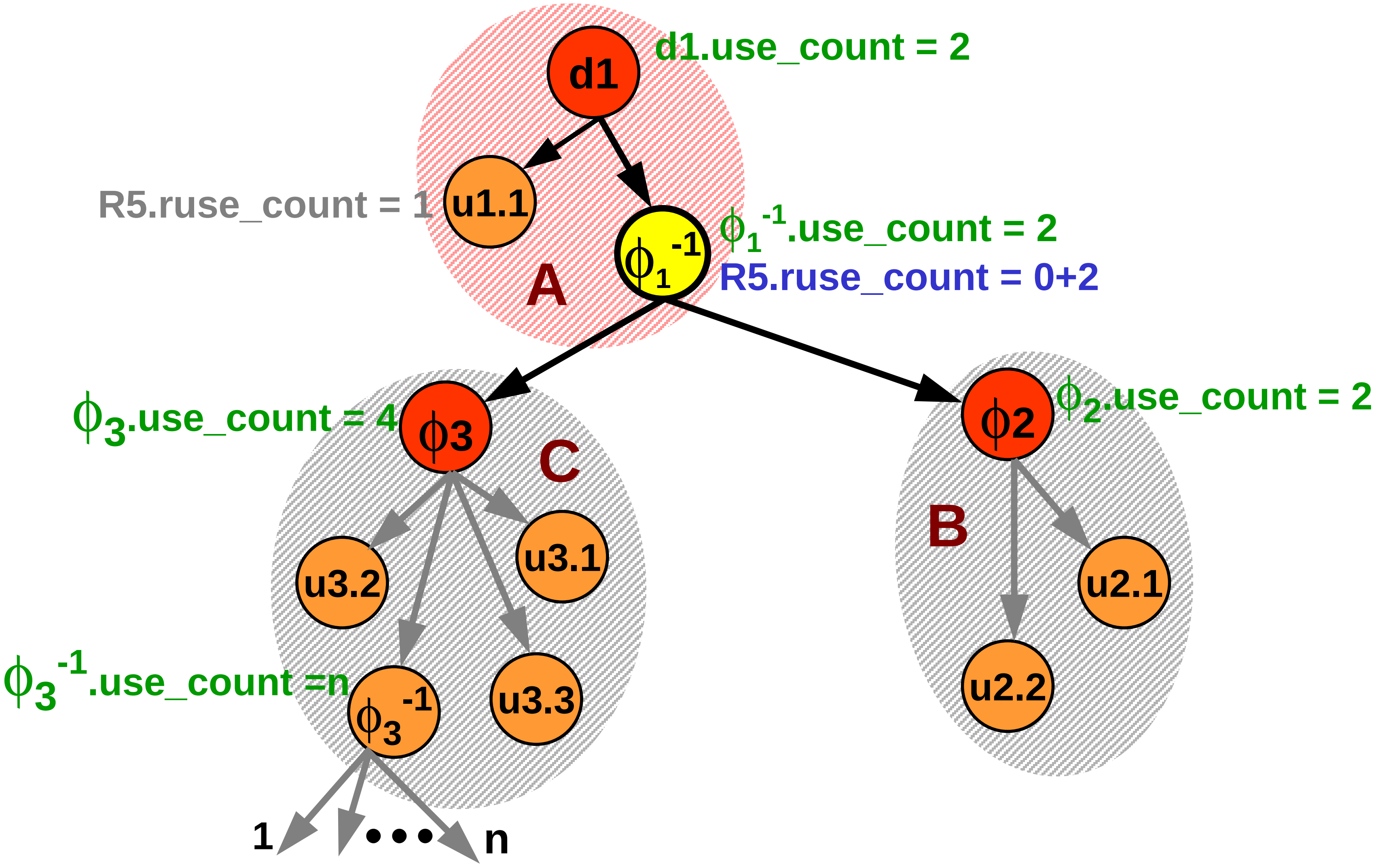
allocating region A

On-the-fly register allocation



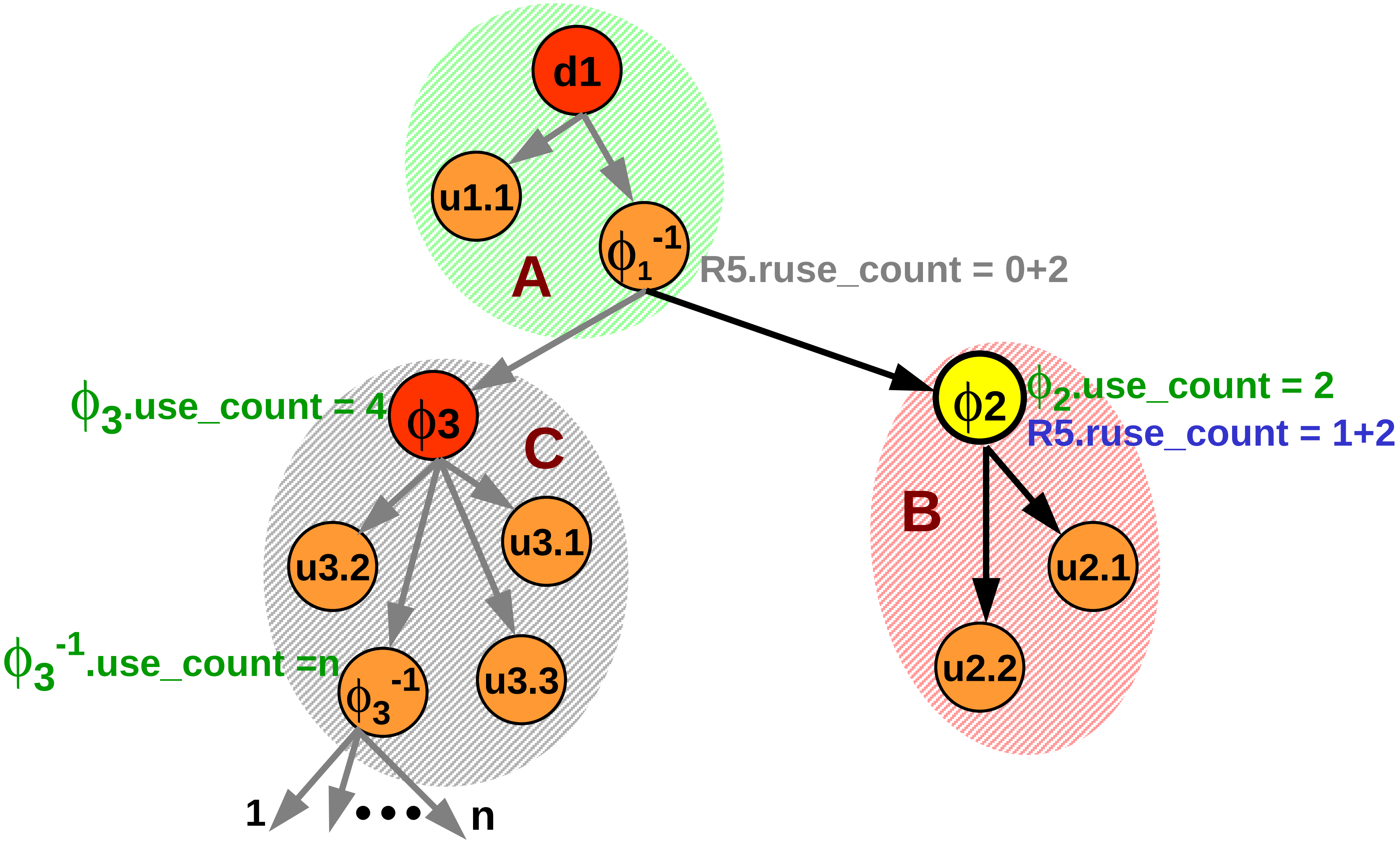
allocating region A

On-the-fly register allocation



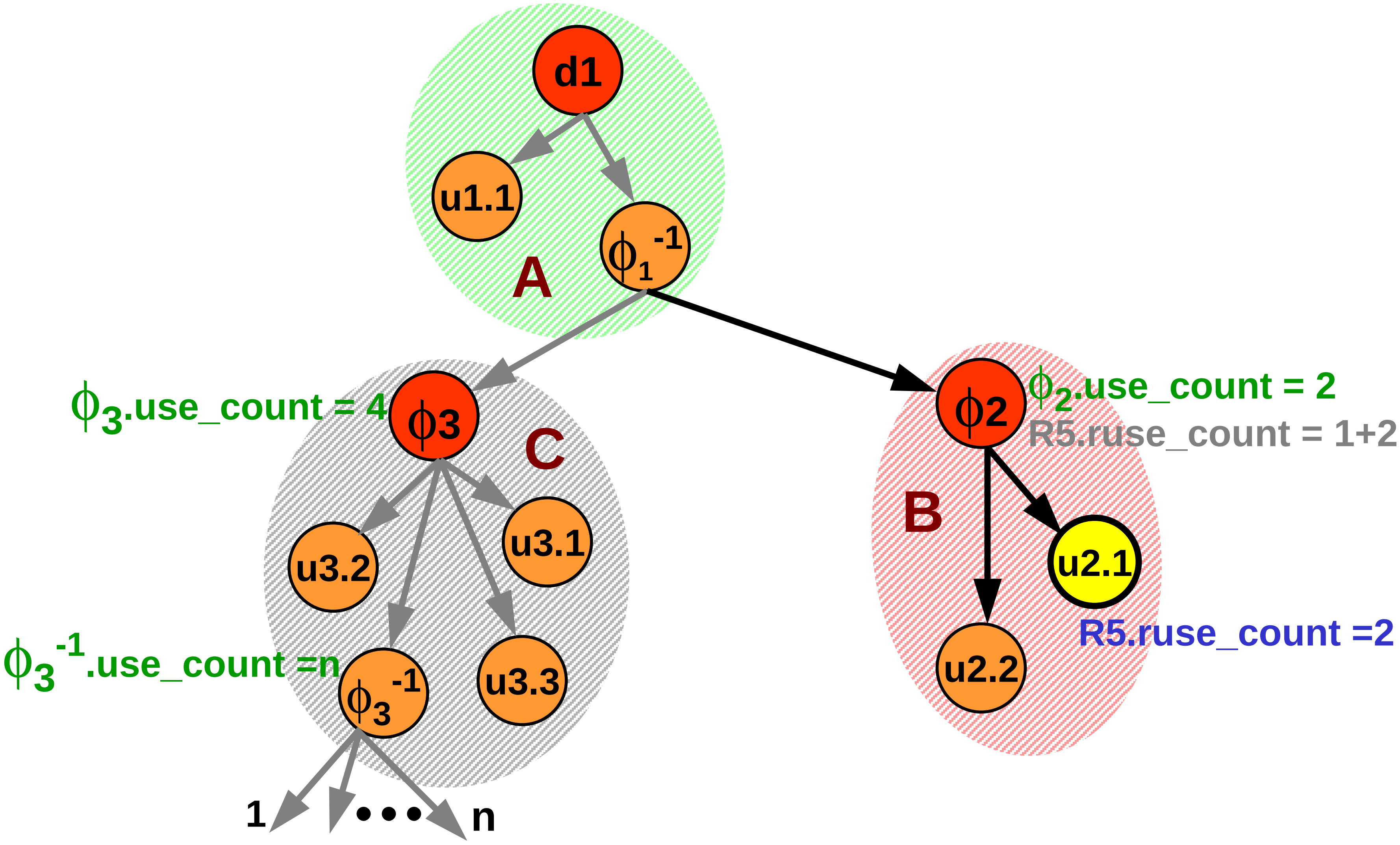
allocating region A

On-the-fly register allocation



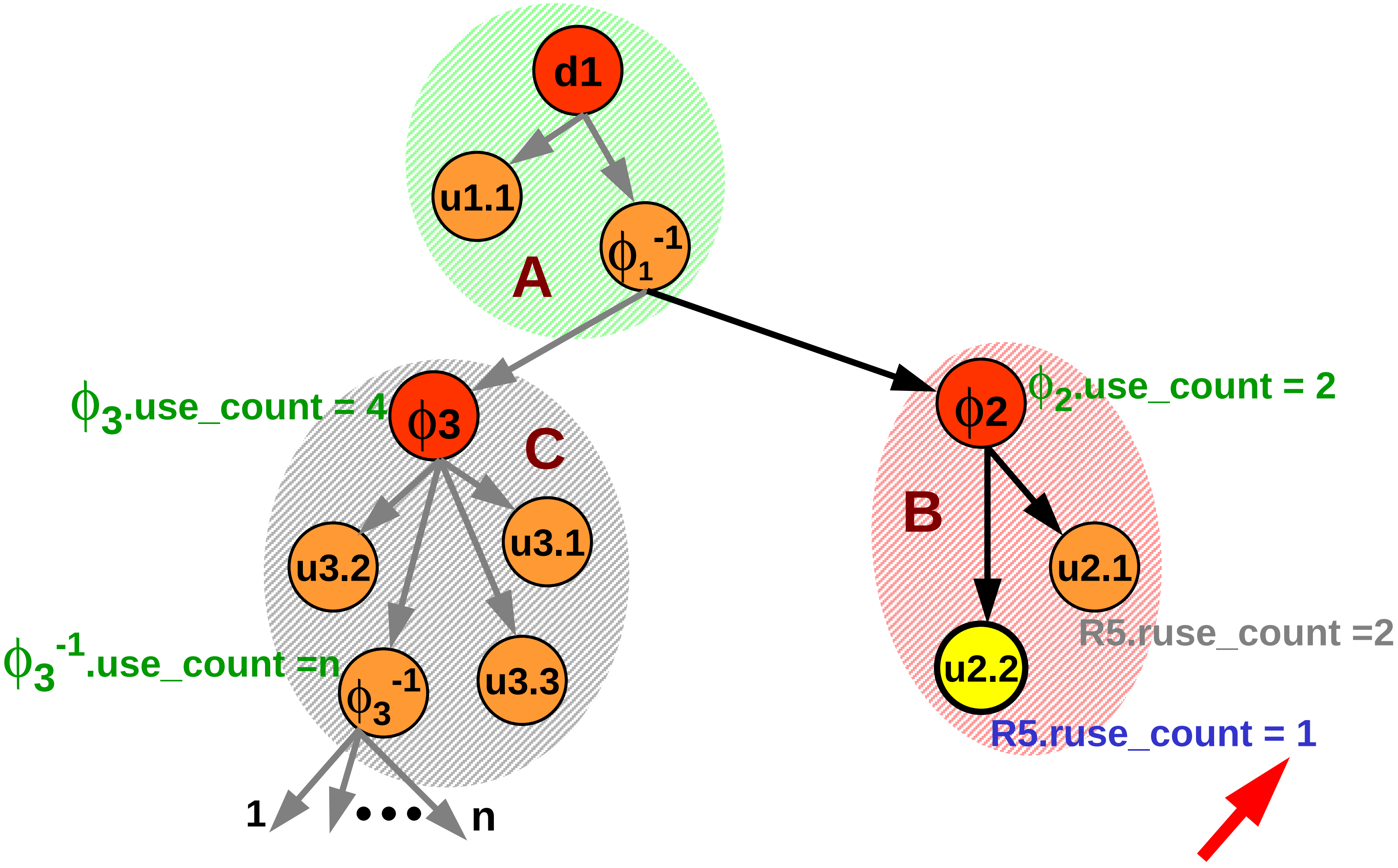
allocating region B

On-the-fly register allocation



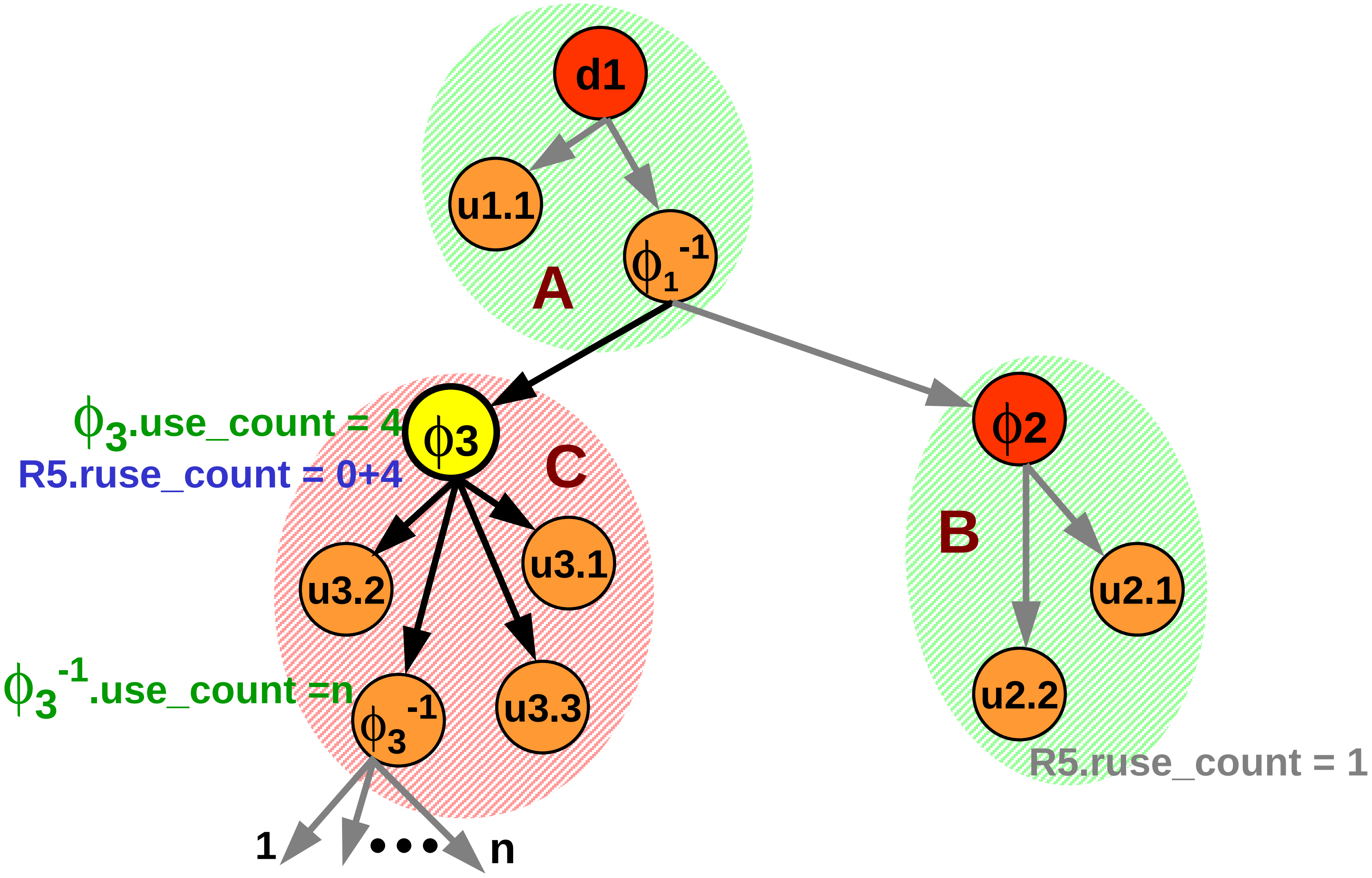
allocating region B

On-the-fly register allocation



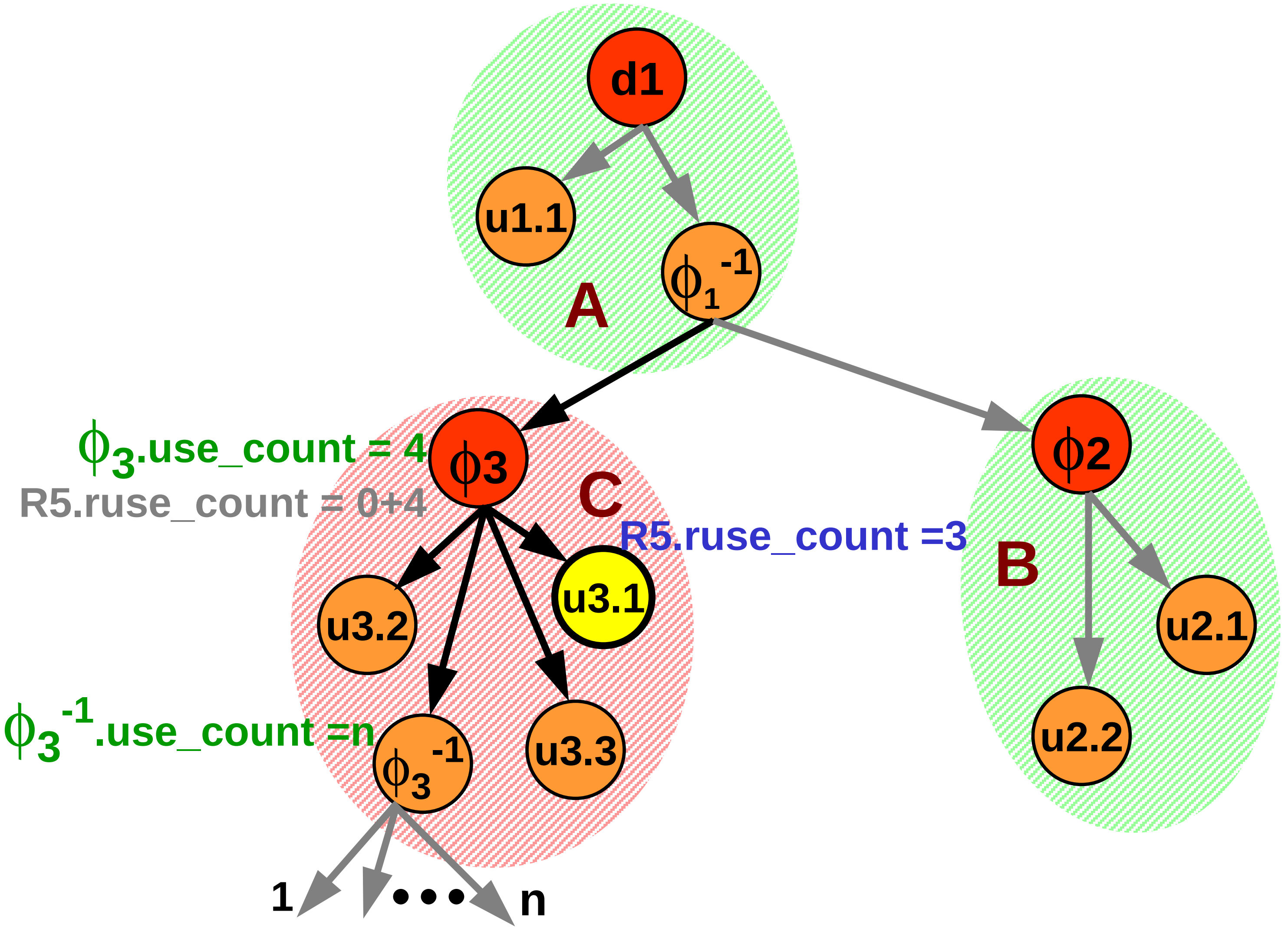
allocating region B

On-the-fly register allocation



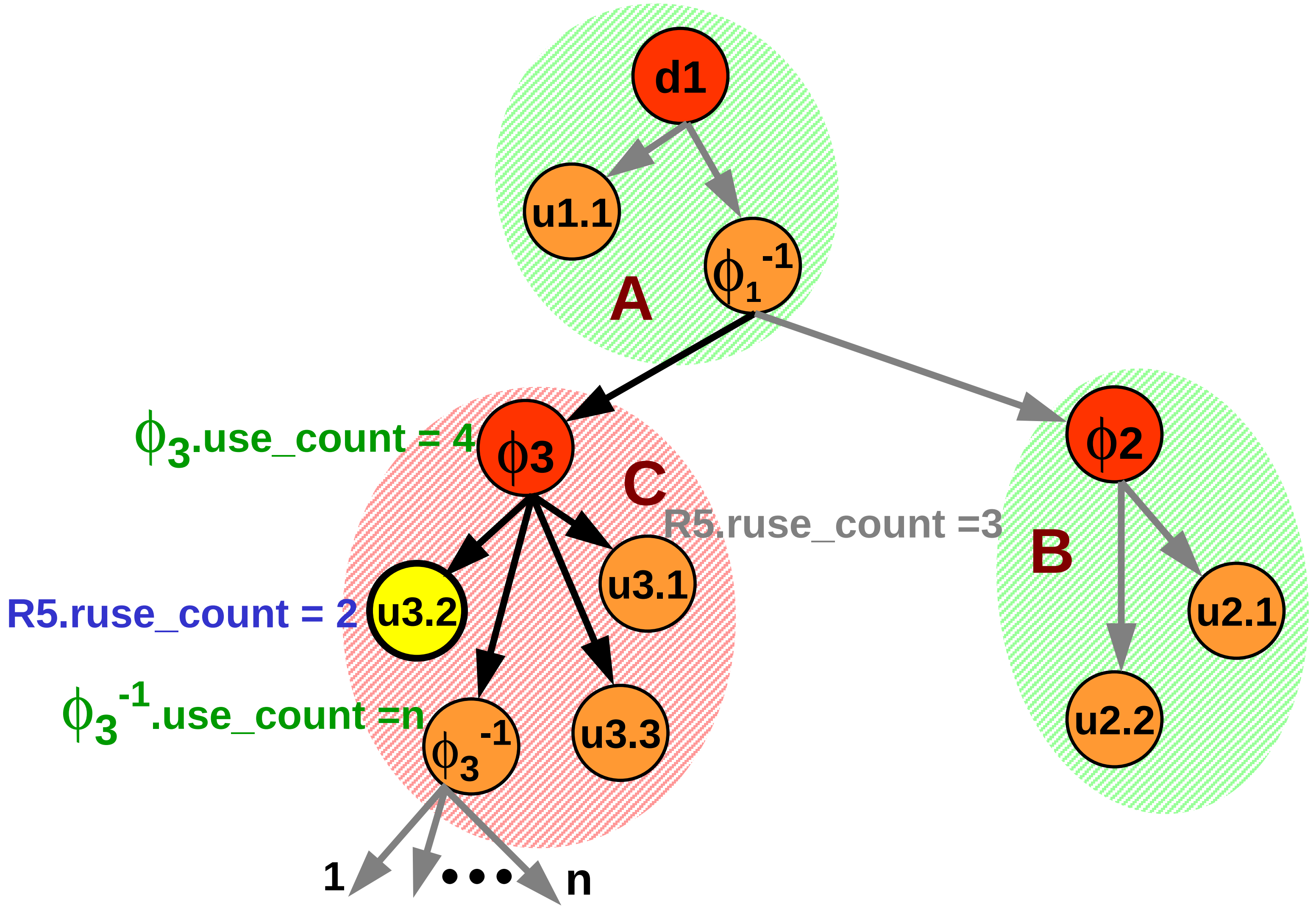
allocating region C

On-the-fly register allocation



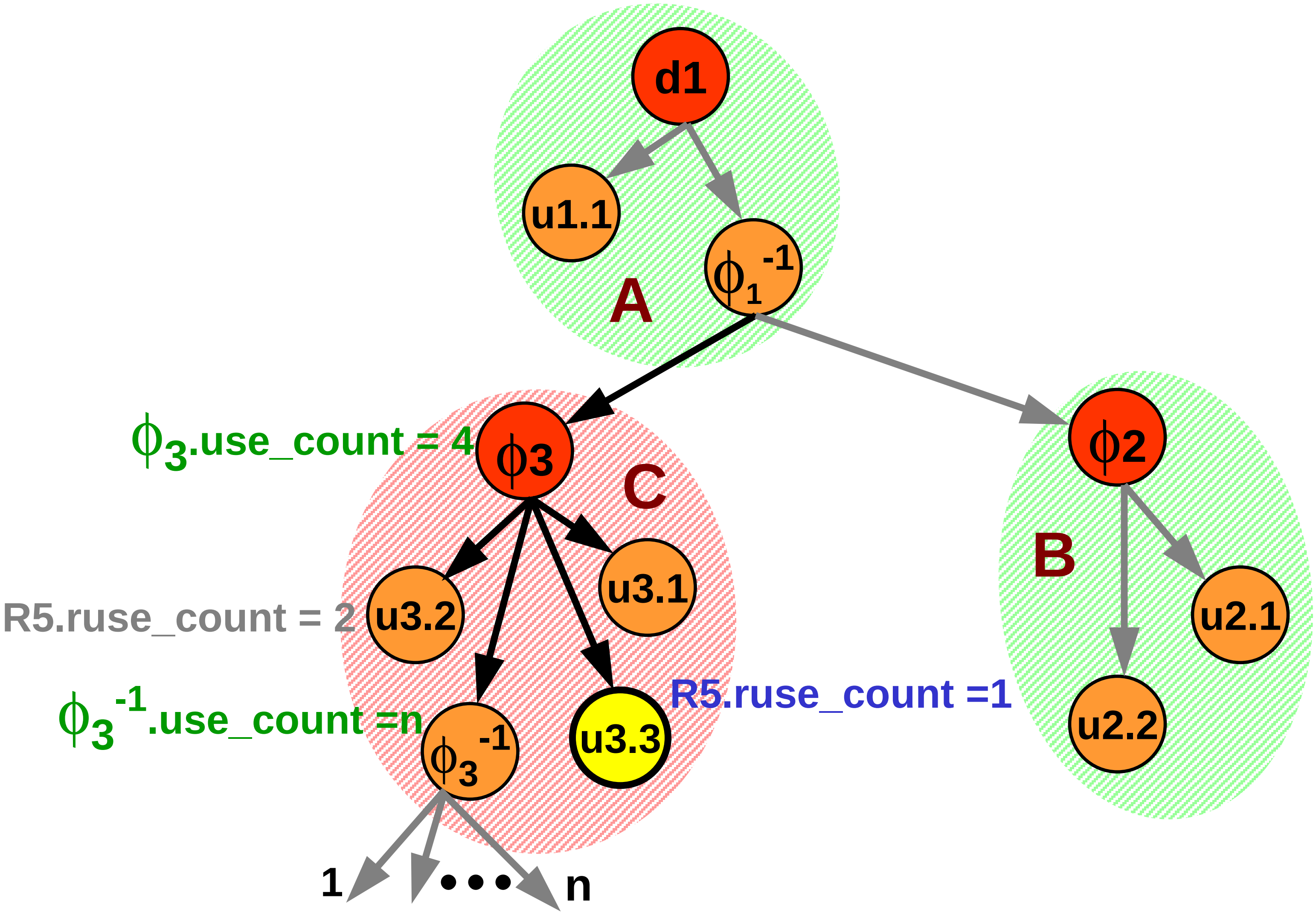
allocating region C

On-the-fly register allocation



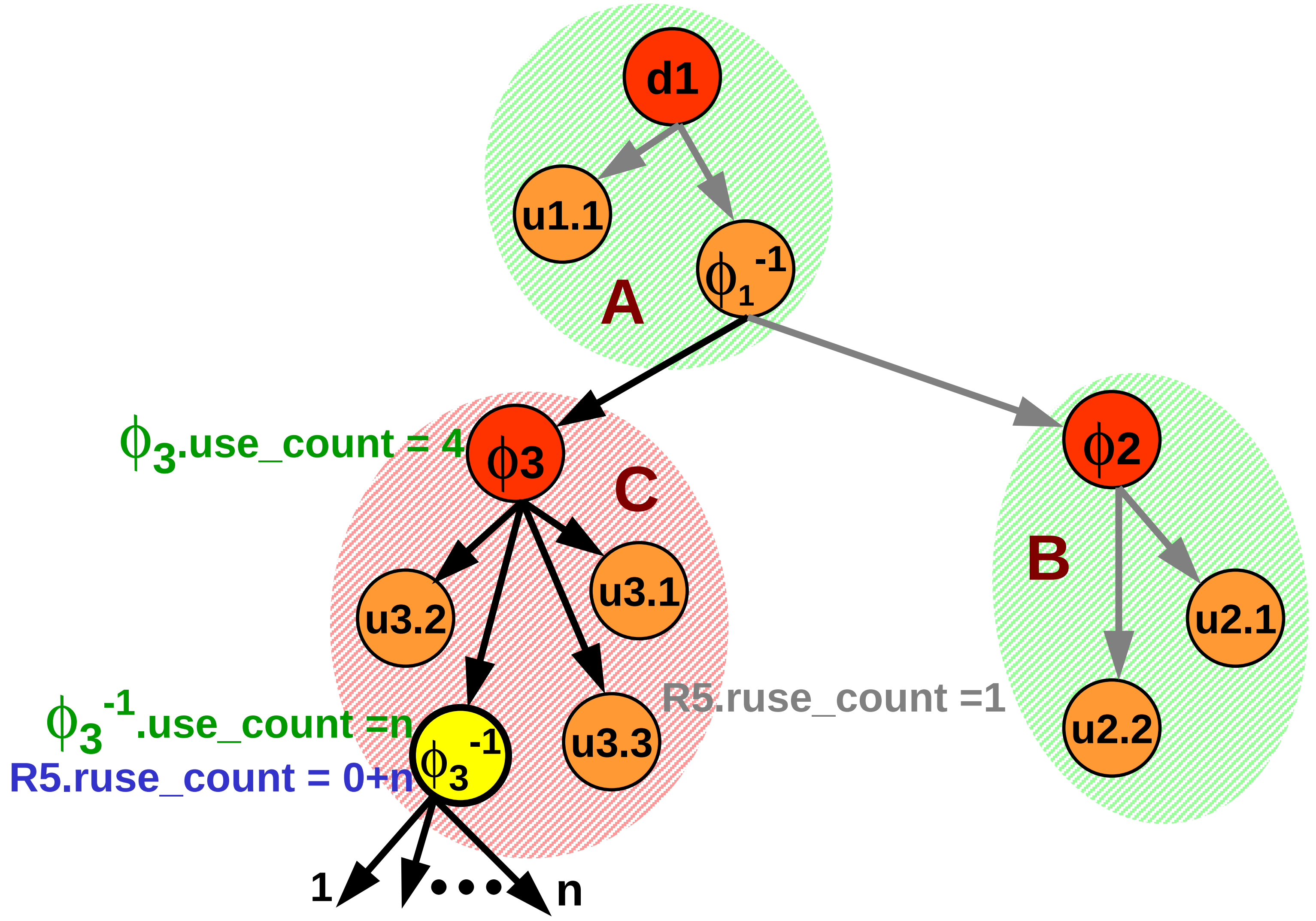
allocating region C

On-the-fly register allocation



allocating region C

On-the-fly register allocation



allocating region C

DEF→register mapping

- **DEFs can have fixed mapping due to:**
  - **calling conventions (static)**
  - **register allocation (dynamic)**

➢ **mapping mismatch problem**

Mapping mismatch problem

```
if(...){
   ...
   y = x + z;
}
else {
    ...
    y = x – z;
}
z = ... y;
```

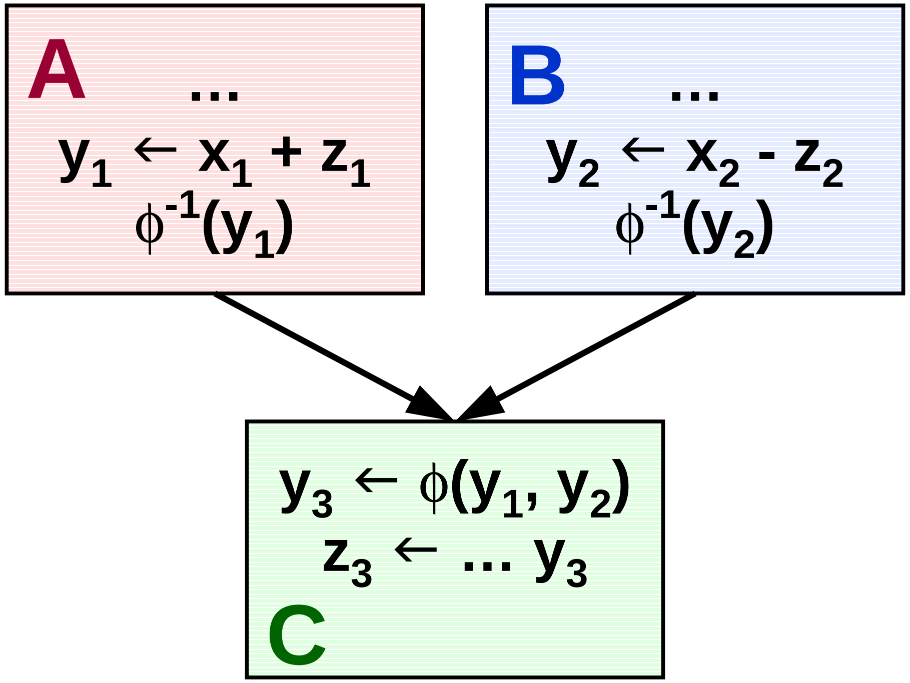
A

B

C

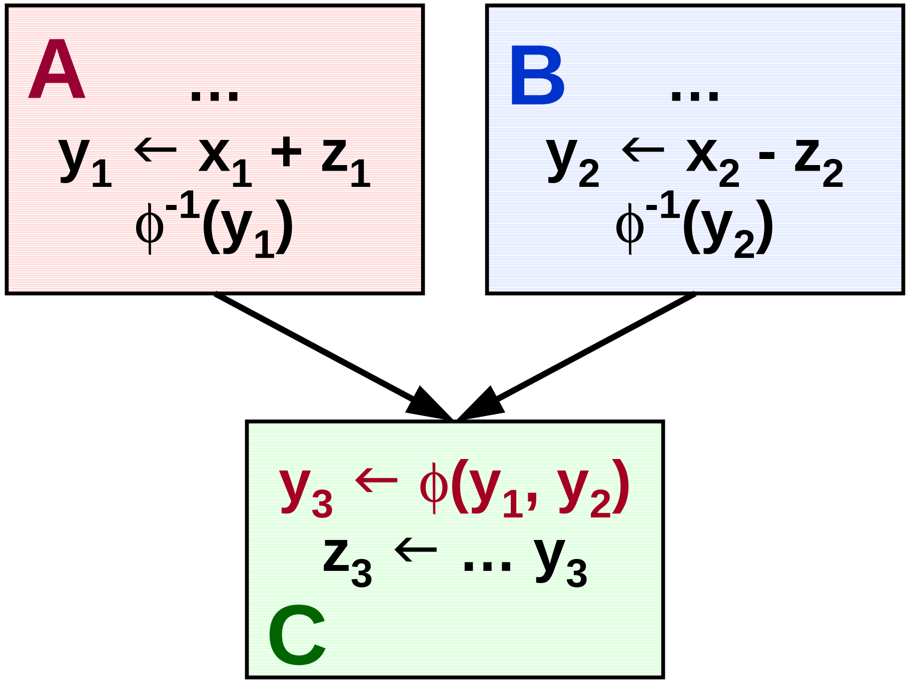
Mapping mismatch problem

```
if(…){
   …
   y = x + z;          A
}
else {
    …
    y = x – z;         B
}
z = … y;               C
```



Mapping mismatch problem

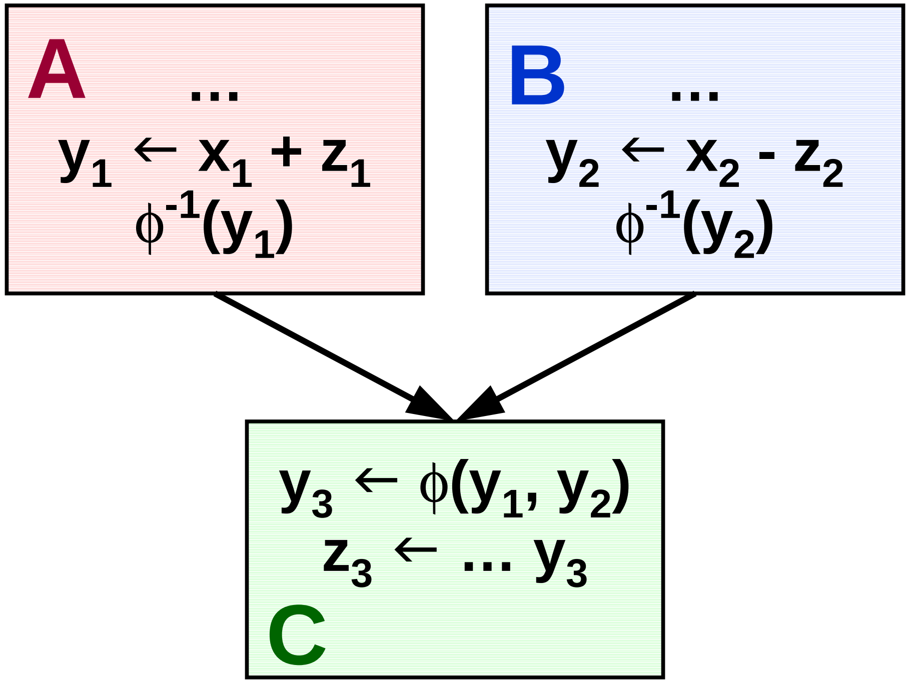
```
if(...){
   ...                 A
   y = x + z;
}
else {
    ...                B
    y = x – z;
}
z = ... y;             C
```



- y_1 mapped to r10 and y_2 mapped to r11
- $y_3 \leftarrow \phi(y_1, y_2)$ not identity assignment
  - needs a copy OP r10 $\leftarrow$ r11 in block B

Mapping mismatch problem

```
if(...){
   ...                A
   y = x + z;
}
else {
    ...               B
    y = x – z;
}
z = ... y;            C
```

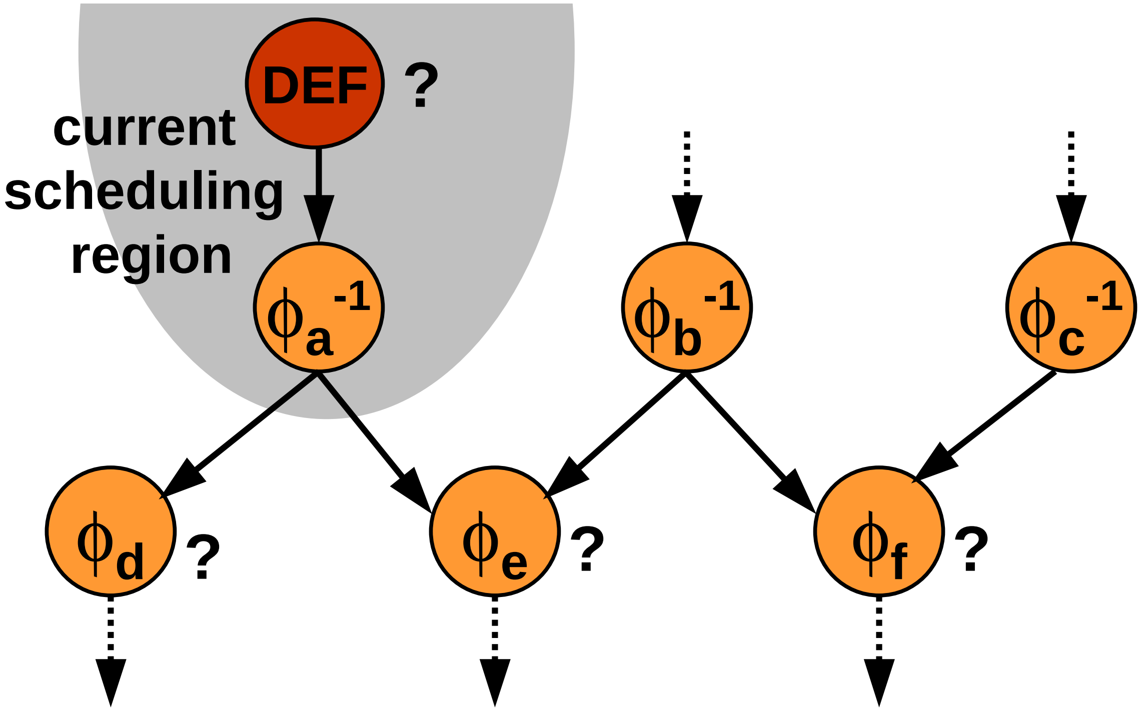


- y_1 mapped to r10 and y_2 mapped to r11
- $y_3 \leftarrow \phi(y_1, y_2)$ not identity assignment
  - needs a copy OP r10 $\leftarrow$ r11 in block B

➢ solution: dynamically assign and propagate regmap info

Dynamic DEF-register map propagation

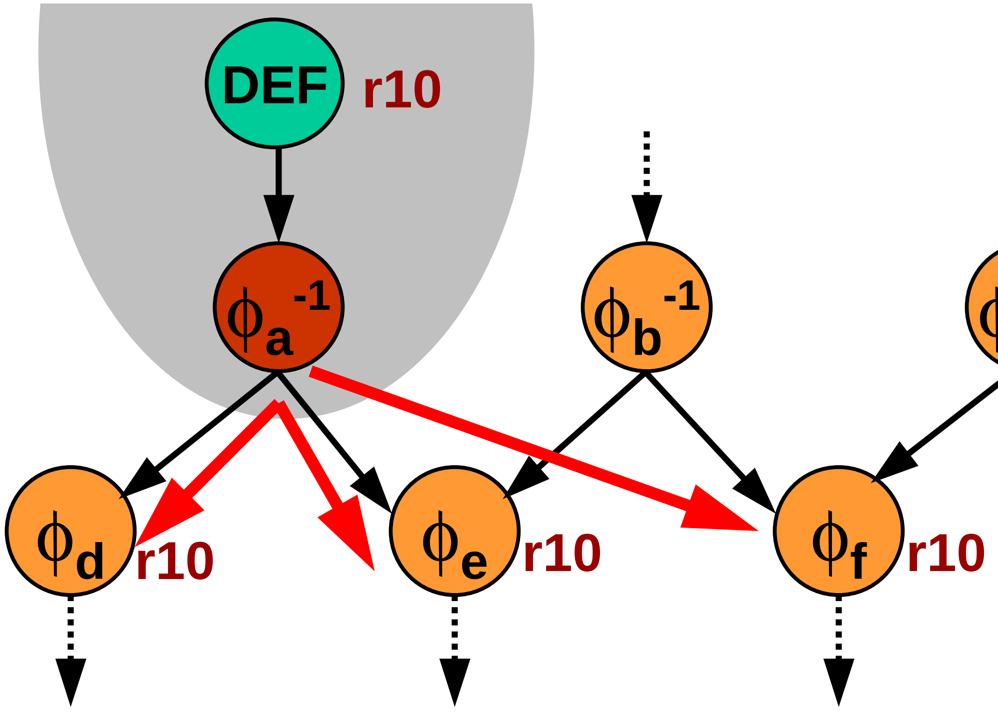
- **preferred regmap** for each DEFs



- similar ideas…
  - (static) value location maps [Multiflow compiler]
  - 2-pass register map propagation [LaTTe compiler]

Dynamic DEF-register map propagation

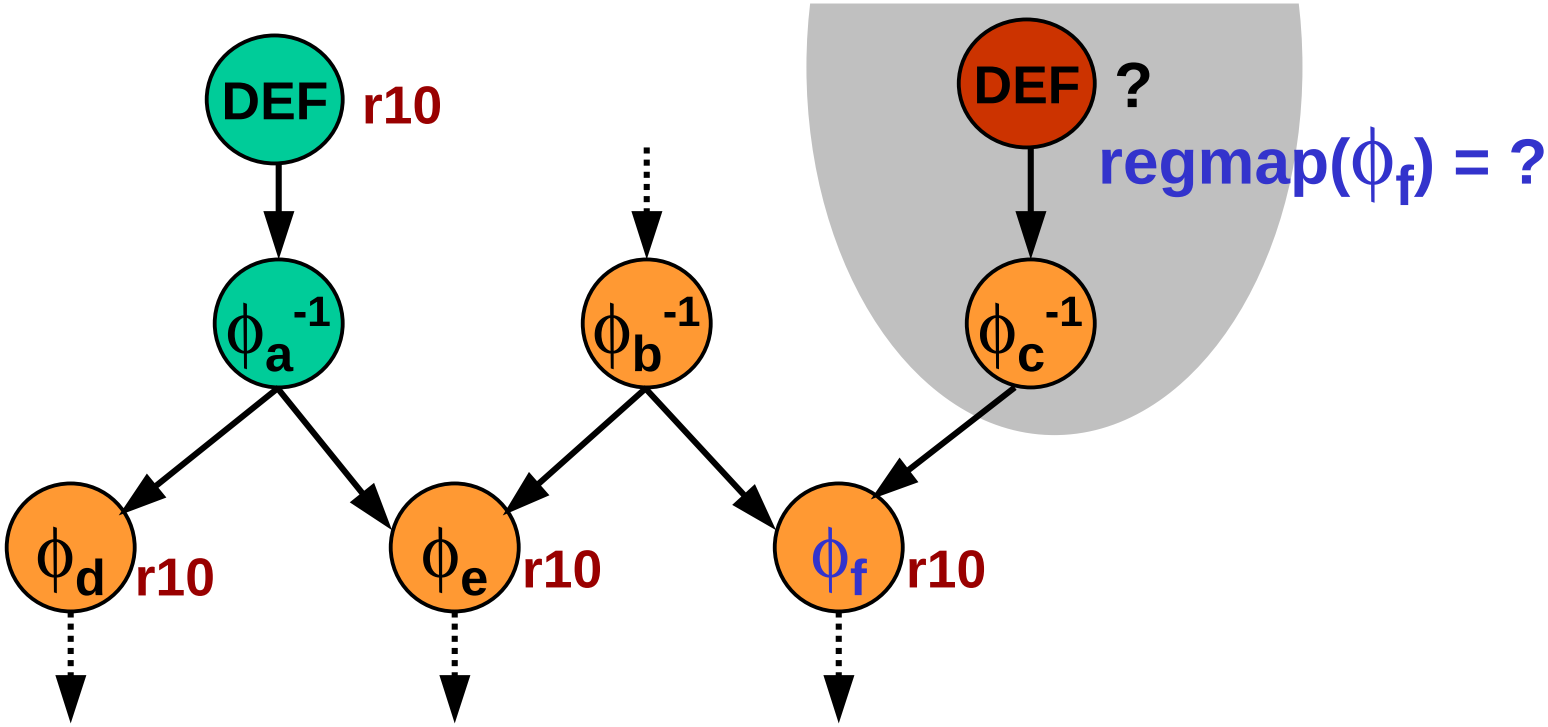
- **preferred regmap for each DEFs**



- **if 1st assignment: propagate regmaps to all "connected" ϕs**
- **if DEF has fork-USE, use existing mappings (if any)**

Dynamic DEF-register map propagation

- preferred regmap for each DEFs



- if 1st assignment: propagate regmaps to all "connected" ϕs
- if DEF has fork-USE, use existing mappings (if any)

on-the-fly register allocation

- **mask bitvector for each DEF**

  eg: mask out caller-save regs for

  DEFs live across call
- **two bitvectors to keep live/dead info**

on-the-fly register allocation

- **mask bitvector** for each DEF

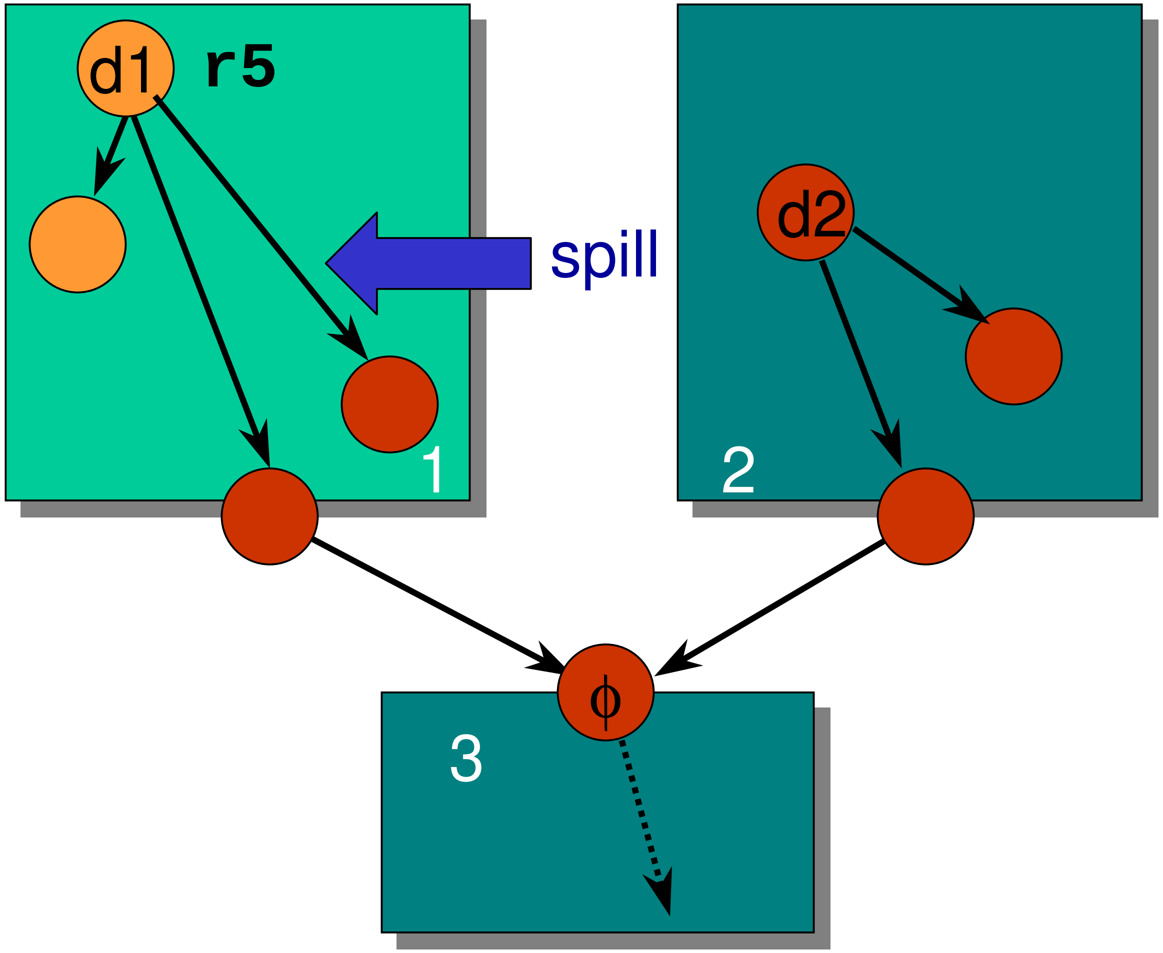
  eg: mask out caller-save regs for

  DEFs live across call
- two bitvectors to keep live/dead info
  1. global-reg-bitvector
     - status of all regs used in scheduled regions
  2. local-reg-bitvector
     - all unused regs in current regions are marked dead
     - can allocate such “dead regs" to certain DEFs
  - provides "interference info"

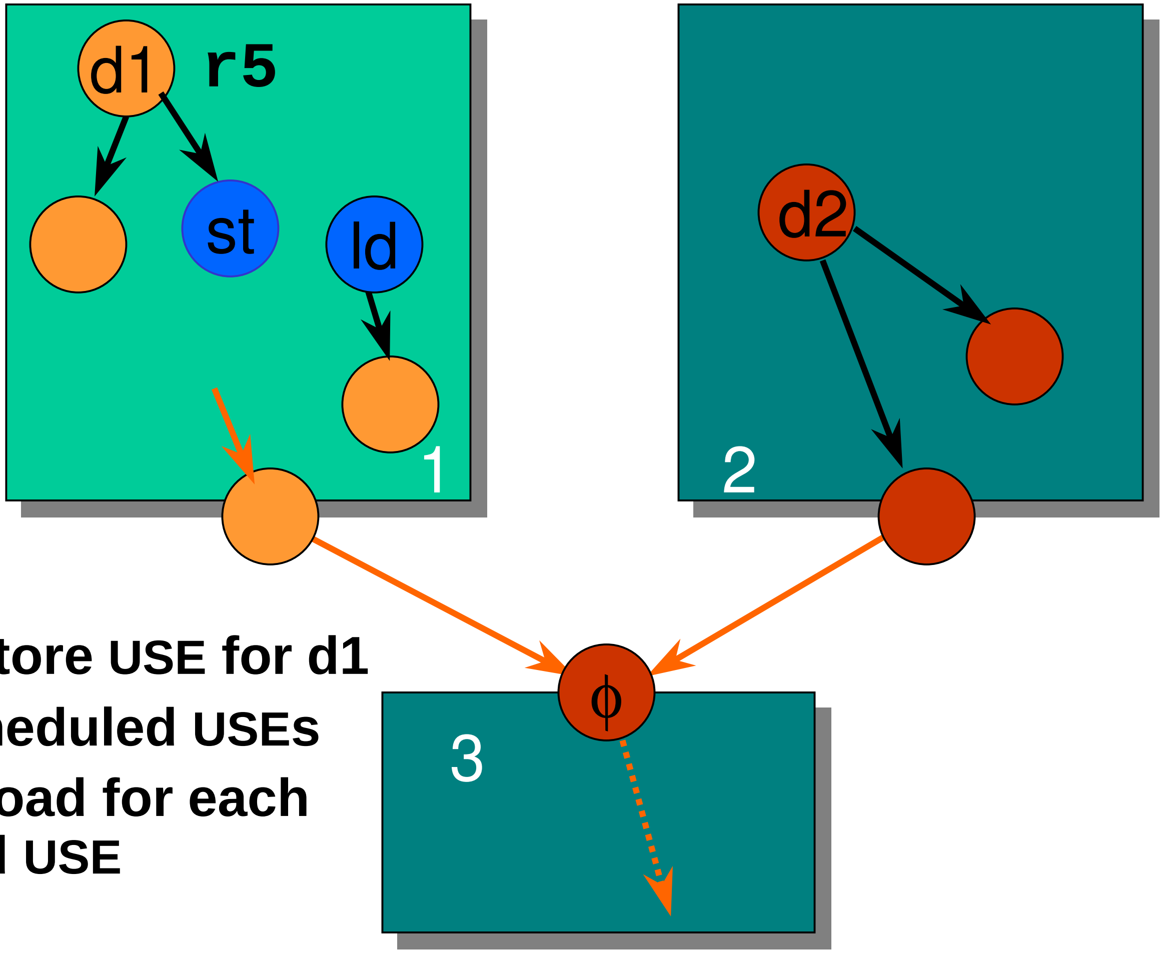
Spilling

- select a live register for spilling
- spill point splits "life" selected for spilling
  - no need to update scheduled code
- assign "unique register ID" to spilled value
- propagate this ID as new preferred regmap of spilled DEF
  - override existing mapping (if any)
- spilling is on-the-fly too

Spilling example



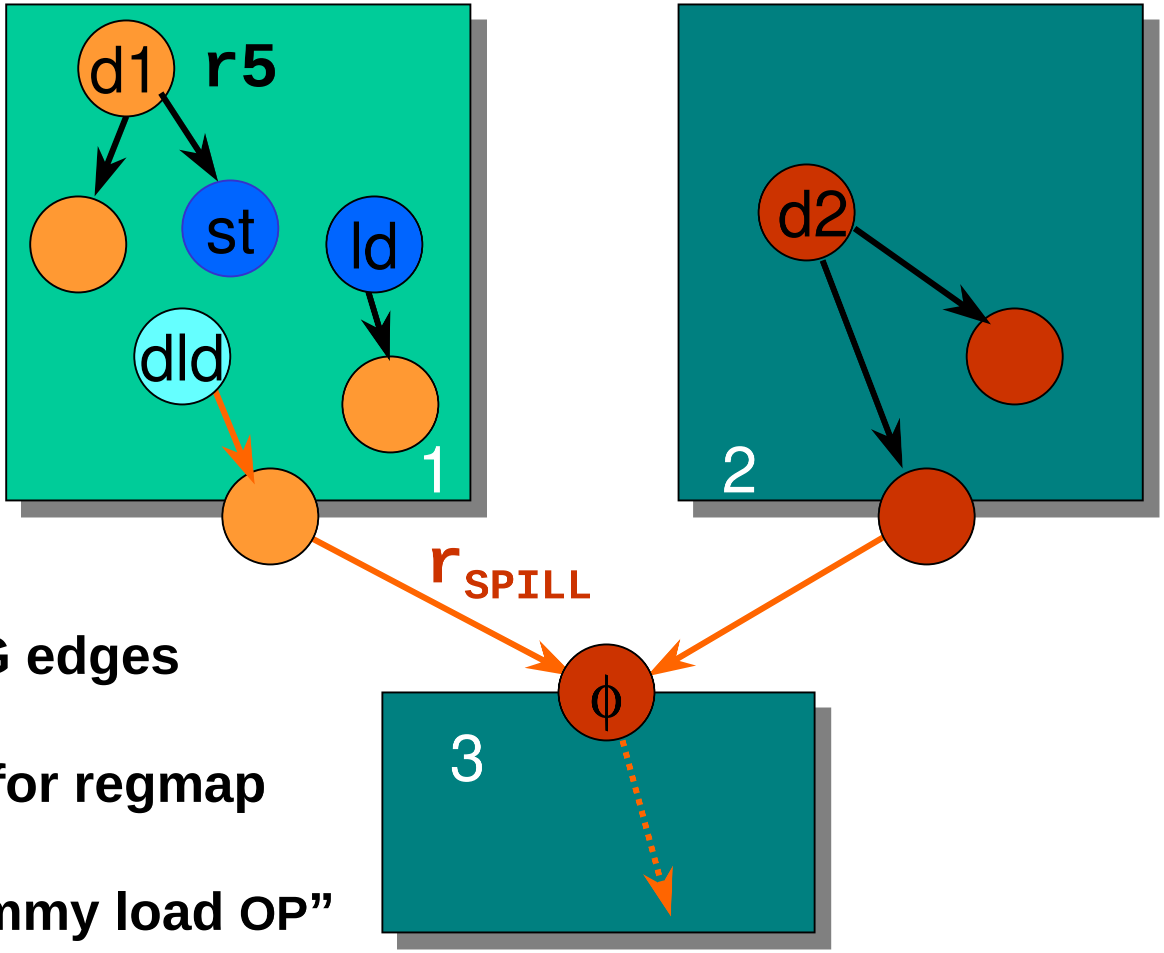
Spilling example



- insert spill-store USE for d1
- delete unscheduled USEs
- insert spill-load for each unscheduled USE

splitting “live range”

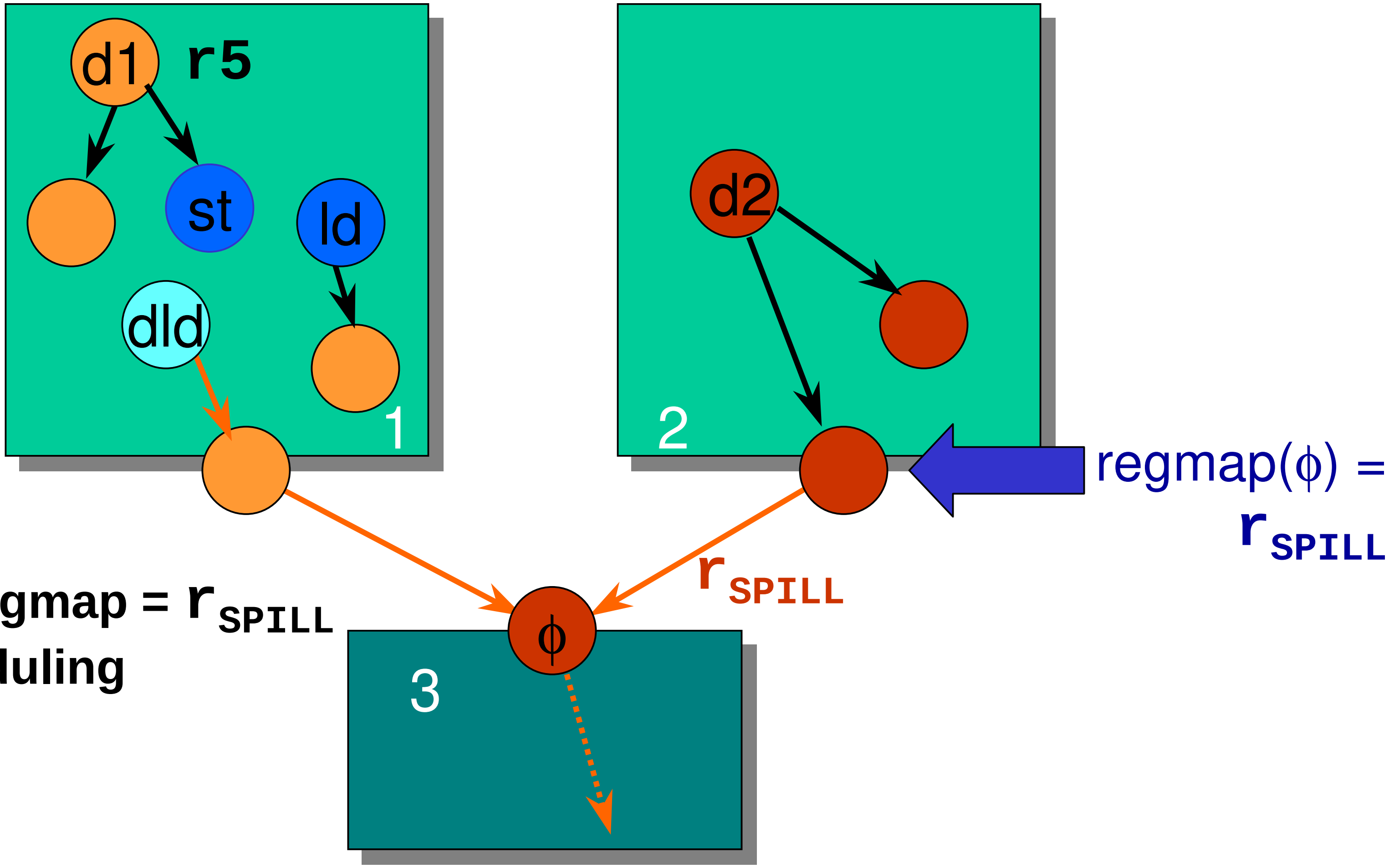
Spilling example



- deleting DFG edges expensive…
- need edges for regmap propagation
- insert a “dummy load OP” for fork-USE

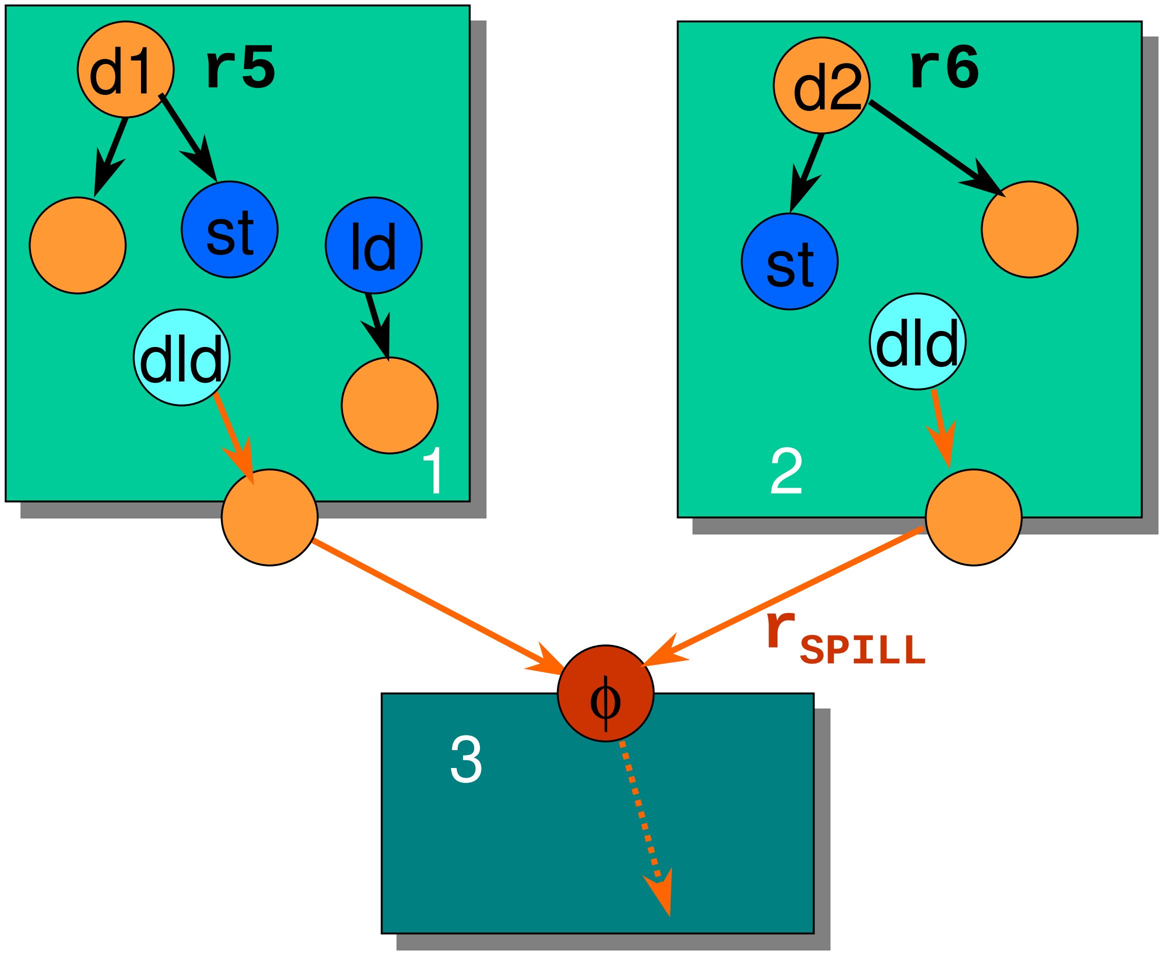
splitting “live range”

Spilling example



- spill d2 w/ regmap = $r_{\texttt{SPILL}}$ before scheduling

Spilling example



Spilling

- selecting an interfering “life”
  - local-reg-bitvector gives interference info
  - use local-reg-bitvector to pick a DEF
- spilling heuristics:
  - prefer longer live ranges
  - prefer DEFs that do not have dynamically assigned fixed regmaps

our on-the-fly register alloc vs. previous work

- prior work
  - graph coloring based (Chaitin/Briggs/Chow-Hennessy)
  - bin-packing based (GEM compiler/DECompaq)
  - Linear scan (Smith et al./Poletto-Sarkar)

  ...

- all of them depend on:
  - prior ordering of OPs
    - interference graphs
    - live range info
  - restricts aggressive scheduling of OPs after allocation
  - scheduling before allocation increase spilling

our on-the-fly register allocation

- does not require nor depend on prior ordering of OPs
- no interference graph computations
- no live range computations

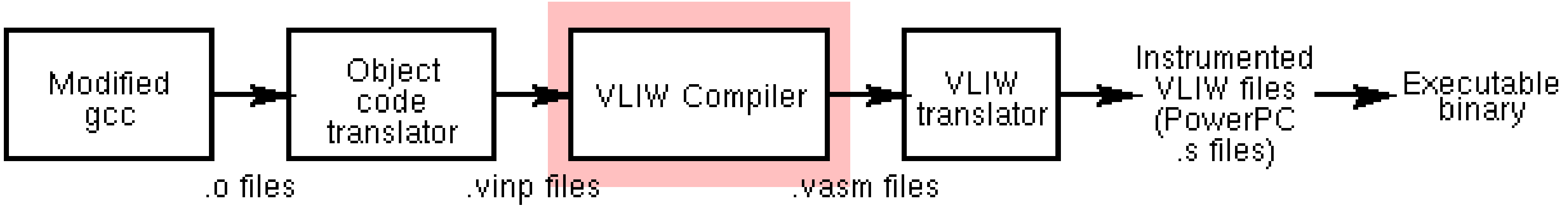
- fast -- complexity: O(nr)

our on-the-fly register allocation

- does not require nor depend on prior ordering of OPs
- no interference graph computations
- no live range computations
  - allows a new class of aggressive out-of-order scheduling schemes
  - allows CARS to prioritize scheduling of OPs in critical path
- fast -- complexity: O(nr)
  - combining w/ cluster scheduling does not increase complexity significantly

Experimental evaluation of CARS

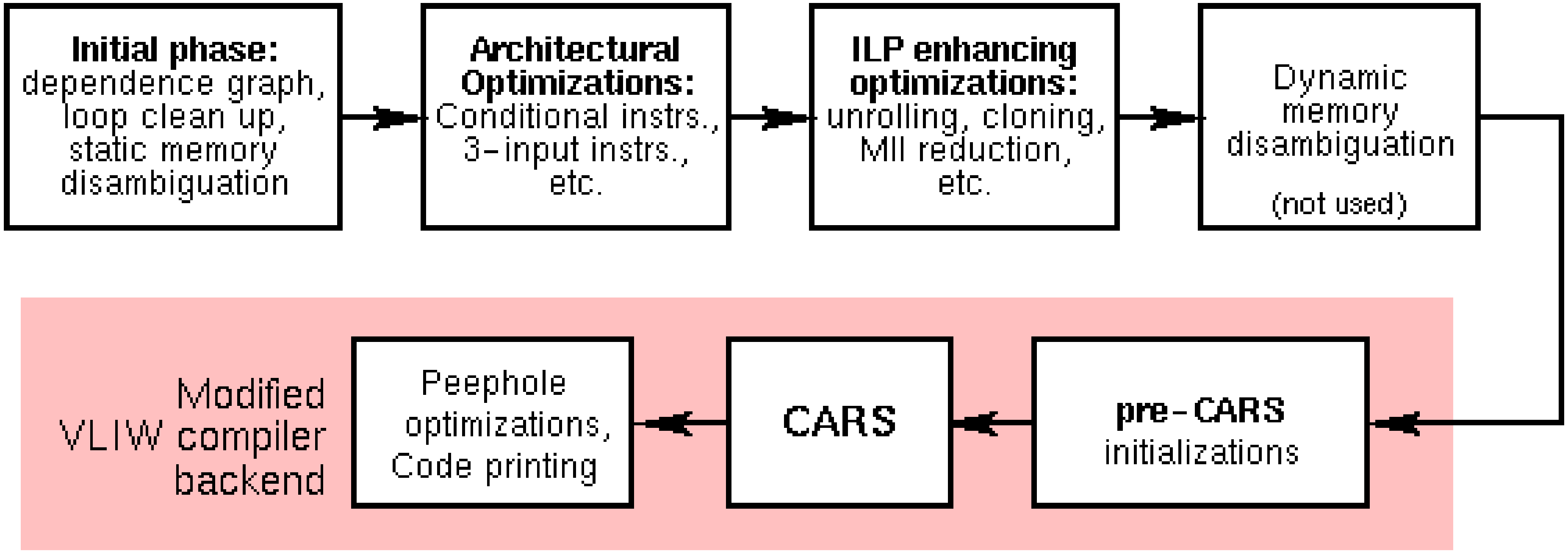
- **used a real compiler…**
- **generates executable code…**
- **compiled-simulation approach**



CHAMELEON -- VLIW research testbed from IBM T. J. Watson Research Center

Experimental evaluation of CARS

VLIW compiler with new backend



- **parametric machine model**

Benchmarks

SPEC-cint95	MediaBench	SPEC-cint2000
099.go	rawcaudio	181.mcf
124.m88ksim	rawdaudio	197.parser
126.gcc	g721.encode	
129.compress	g721.decode	
130.li	epic	
132.ijpeg	unepic	
134.perl		
147.vortex		

- **total 17 benchmark programs**

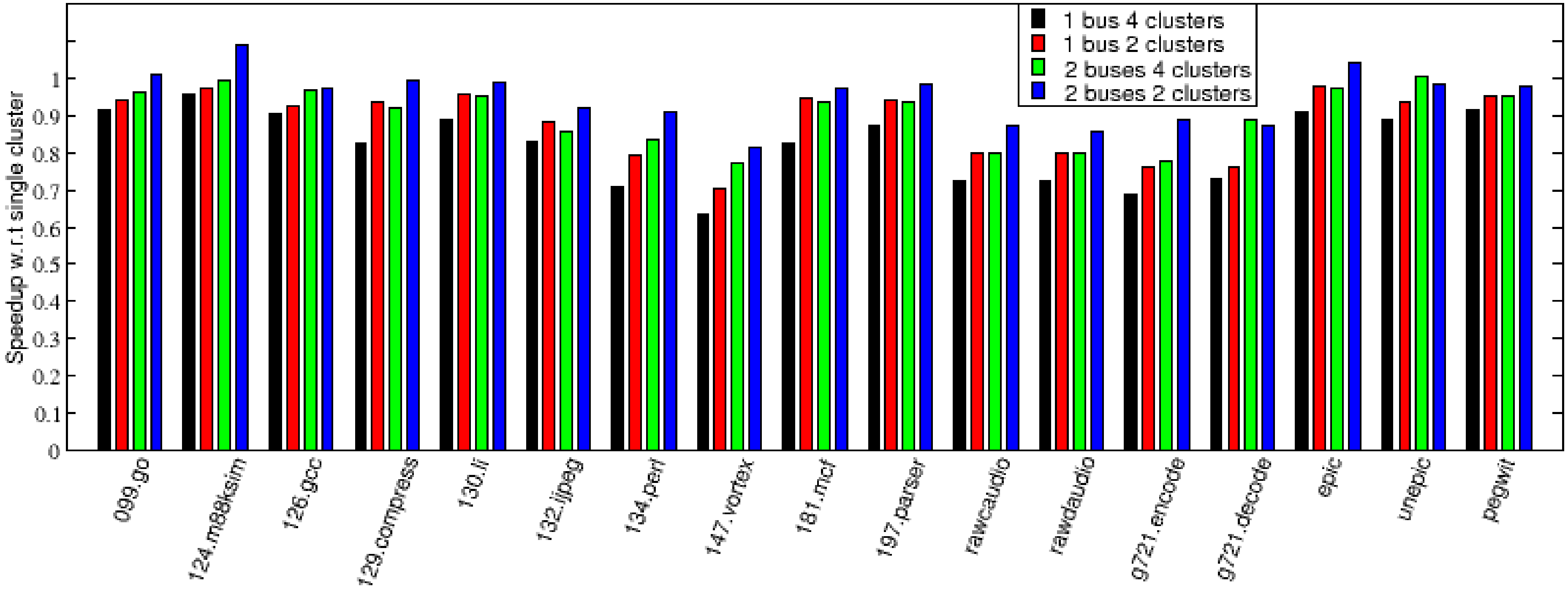
Clustered processor configurations

- base machine:
  - 8 and 16-issue single cluster processor
  - register file: 64 INT, 64 FP and 16 CC-bits
- 8 basic clustered processor configurations:
  - 8- and 16-issue versions of:
    - 1b4c: 1 bus, 4 clusters
    - 2b2c: 2 bus, 2 clusters
    - 2b4c: 2 bus, 4 clusters
    - 1b2c: 1 bus, 2 clusters
- function units and register file of base m/c are partitioned and evenly distributed to each cluster

Latencies

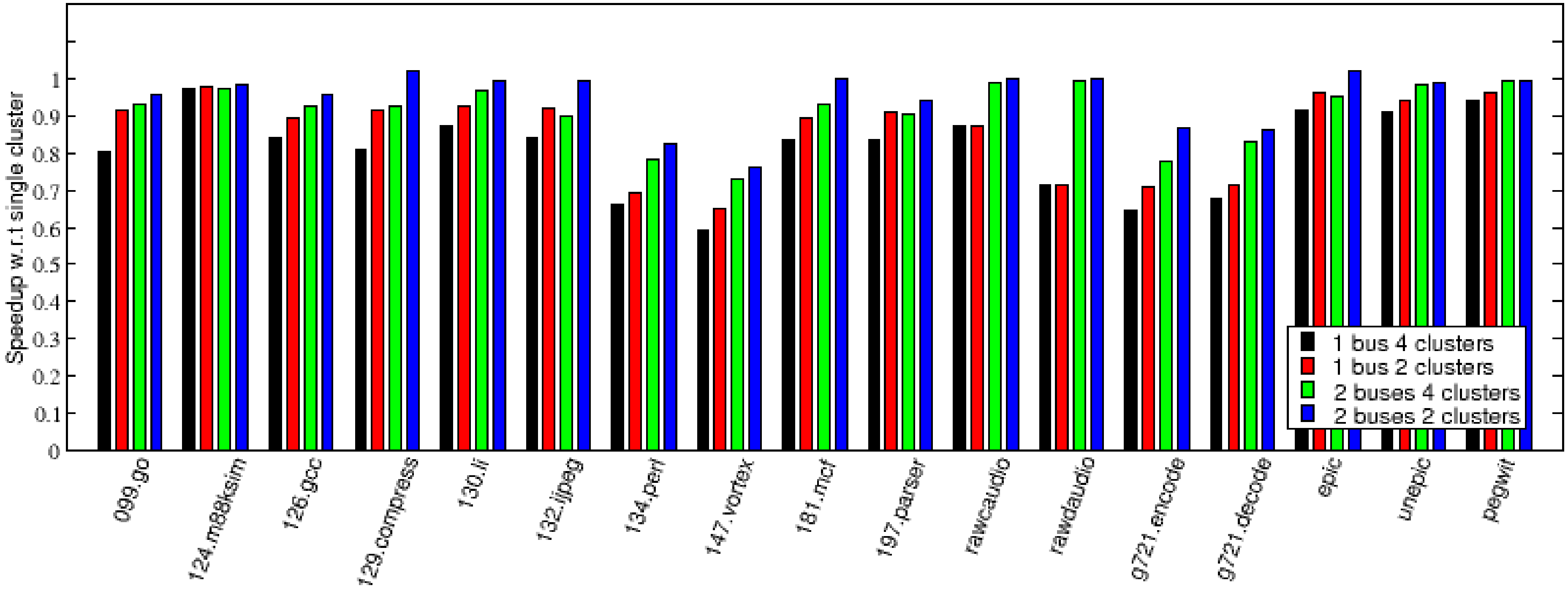
Function unit	Latency
Fix	1
FP	2
FP divide	9
FP sqrt	9
Load	2
Branch	1
Communication	1

Clustered vs. Single cluster processor performance



8 ALU configurations

Clustered vs. Single cluster processor performance



16 ALU configurations

Results

summary:

- **average number of additional cycles less than**

  10% (2-clust/16 or 8-issue)

  14% (4-clust/16 or 8-issue)

- **1 bus to 2 bus gives ~10% increase in speedup**

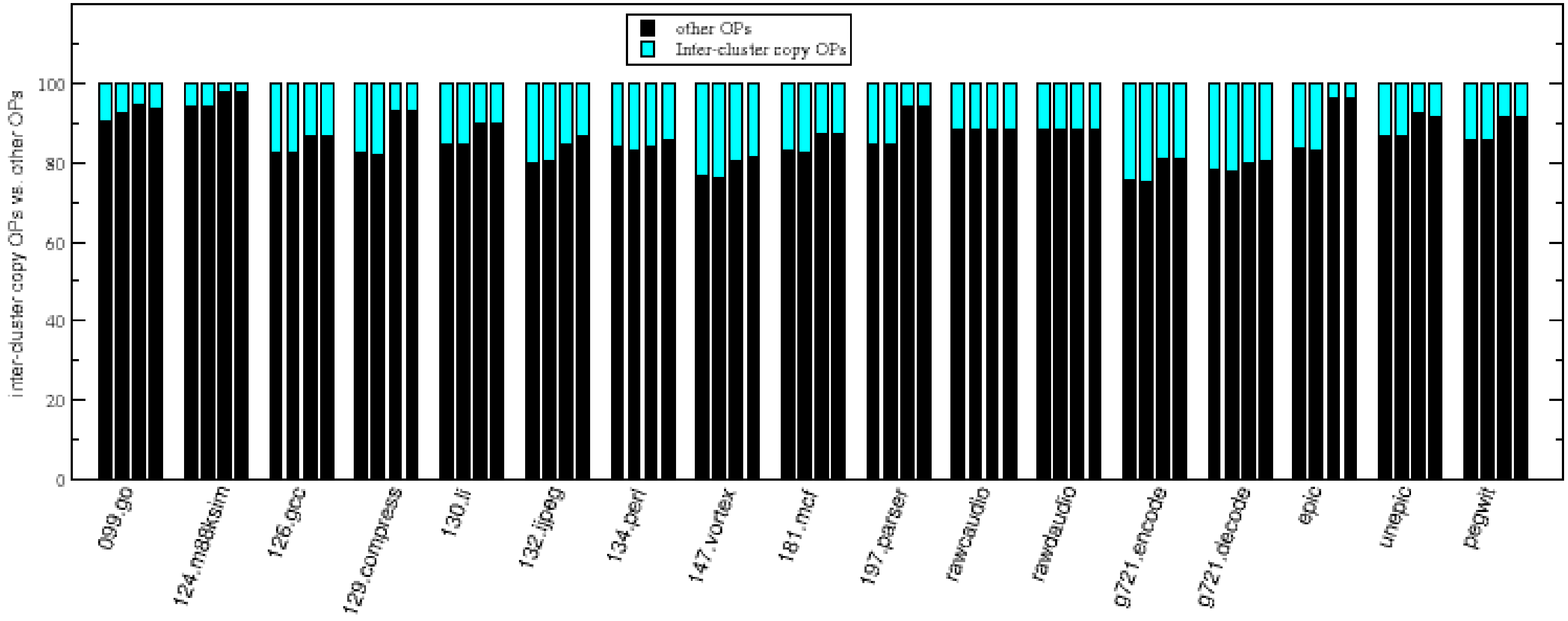
Results Comparison

summary:

- **average number of additional cycles less than**
  10% (2-clust/16 or 8-issue)
  14% (4-clust/16 or 8-issue)
- **other schemes: 15-25% performance penalty**
  [Faraboschi-Desoli-Fisher 1998]

- **1 bus to 2 bus gives ~10% increase in speedup**
- **other schemes: little speedup**
  [Ozer-Banerjia-Conte 1998, 2000]

`rcopy` vs. other OPs

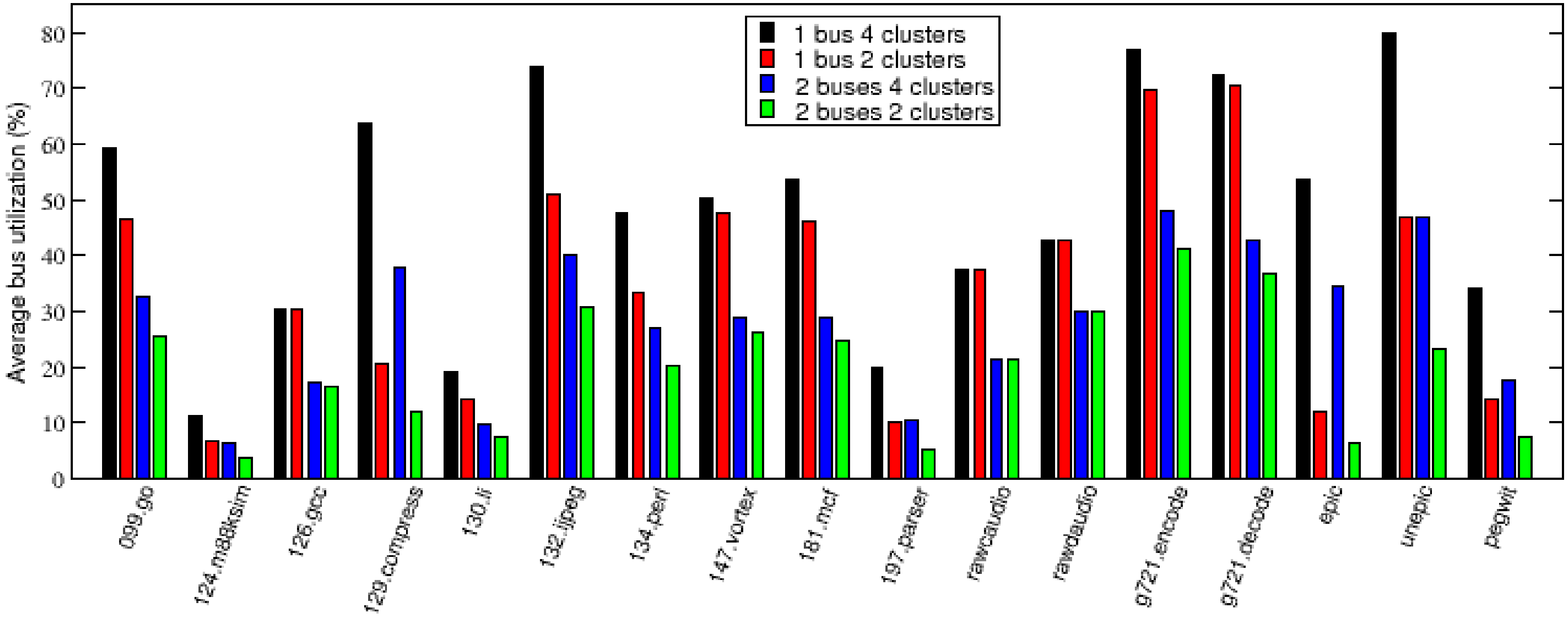


L-to-R: 1b4c, 2b4c, 2b2c, 1b2c (8 ALU processor)

- **`rcopy` OPs are 10-16% of total OPs**

➢ **CARS is a greedy algorithm…**

Average bus utilization



8 ALU processor

- **higher bus utilization for lower number of buses**
- **higher bus utilization for larger number of clusters**

Related work

- **Bulldog compiler/BUG** [Ellis-Fisher 1985]
- **Multiflow compiler** [Lowney etal. 1993]
- **UAS algorithm** [Ozer-Banerjia-Conte 1998]
- **LC-VLIW** [Capitanio-Dutt-Nicolau 1992]
- **PCC** [Desoli 1998]

- **Multiprocessor scheduling**

  [Sarkar 1989, Yang-Gerasoulis 1994]

Related work vs. CARS

 to  comparison is difficult…

Related work vs. CARS

 to  comparison is difficult…

code generation scheme	phases	executable code?	spilling ?	complexity	
				worst	observed
Bulldog/ BUG	1. A 2. S + R	✓	✗	$O(n^2 \log n)$	$O(n\ r)$
UAS	1. A + S	✗	n/a	$O(n^3)$	$O(n^2)$
CARS	1. A + S + R	✓	✓	$O(m\ n^2)$	$O(n^2)$

A = cluster Assign, S = Scheduling, R = Register allocation

- n = number of nodes, r = max. # of reg banks
- m = average retry factor
- measured worst-case value of m=1.49 for CARS

Not discussed today...

- new partitioned register file architecture
  - low interconnect bandwidth requirement
  - better utilization of architected name space
- distributed D-cache
  - no snoopy bus protocols
  - support for out-of-order stores
- DAISY processor for dynamic binary translation

  IBM T. J. Watson Research Center
  - clustered (wide) VLIW processor
  - replicated register files

Conclusion

- new code generation framework: CARS
  - resource-constrained, single-phase
  - better performance
  - out-of-order scheduling
  - avoids spilling

- fast on-the-fly register allocation scheme
  - does not depend on ordering of OPs
  - can be used in dynamic/JIT compilation

for more details:

- UMIACS technical report, July 2000.
- my Ph.D. dissertation (expected soon)
  `www.cs.umd.edu/~krish/publications.html`