Dynamic Instruction Reuse

Avinash Sodani

Guri Sohi

Computer Sciences Department

University of Wisconsin — Madison

Motivation

- Programs consist of static instructions
- Execution sees static instruction many times
  - often with same inputs

    ➔ produces same result

    ➔ no need to compute again

Exploited by

- Buffer results of instructions
- Reuse old result if input operands are same

Dynamic Instruction Reuse

Advantages

Instruction Reuse

+ permits dependent instructions to issue earlier

+ reduces resource contention

+ salvages useful work from misprediction squashes

+ completes chains of dependent instruction in single cycle

  ➔ potentially breaks dataflow limit

Outline

- ~~Motivation~~
- What enables reuse ?
- Implementing Reuse
- Three Reuse Schemes
- Some Results
- Summary

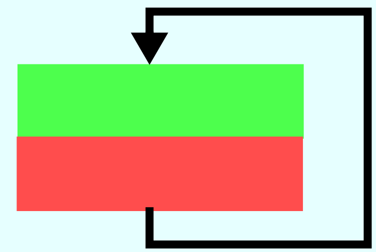
What enables reuse?

- General Reuse
  - due to the way computation is expressed

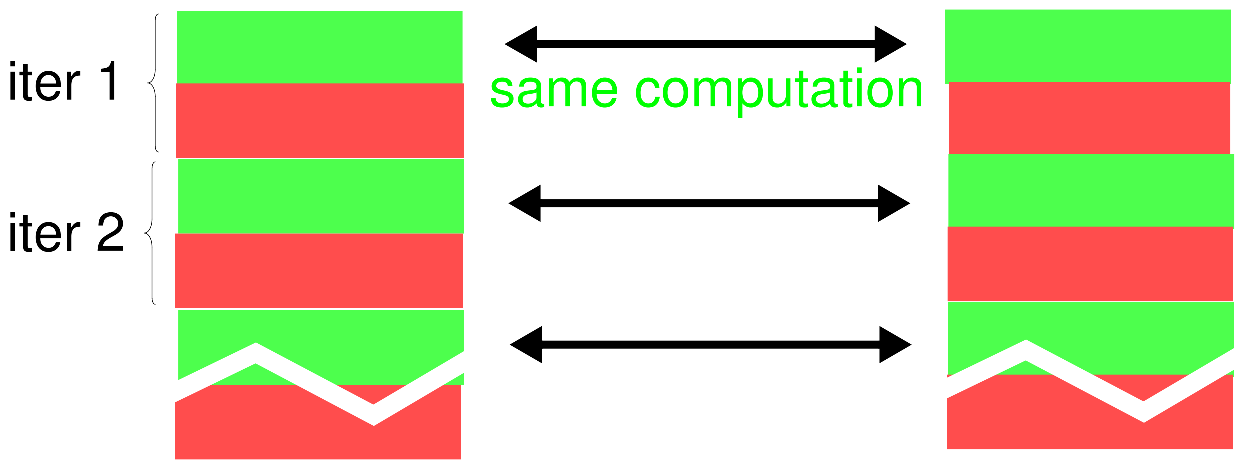
    ➔ same code visited with same data
- Squash Reuse
  - due to mis-speculation

General Reuse

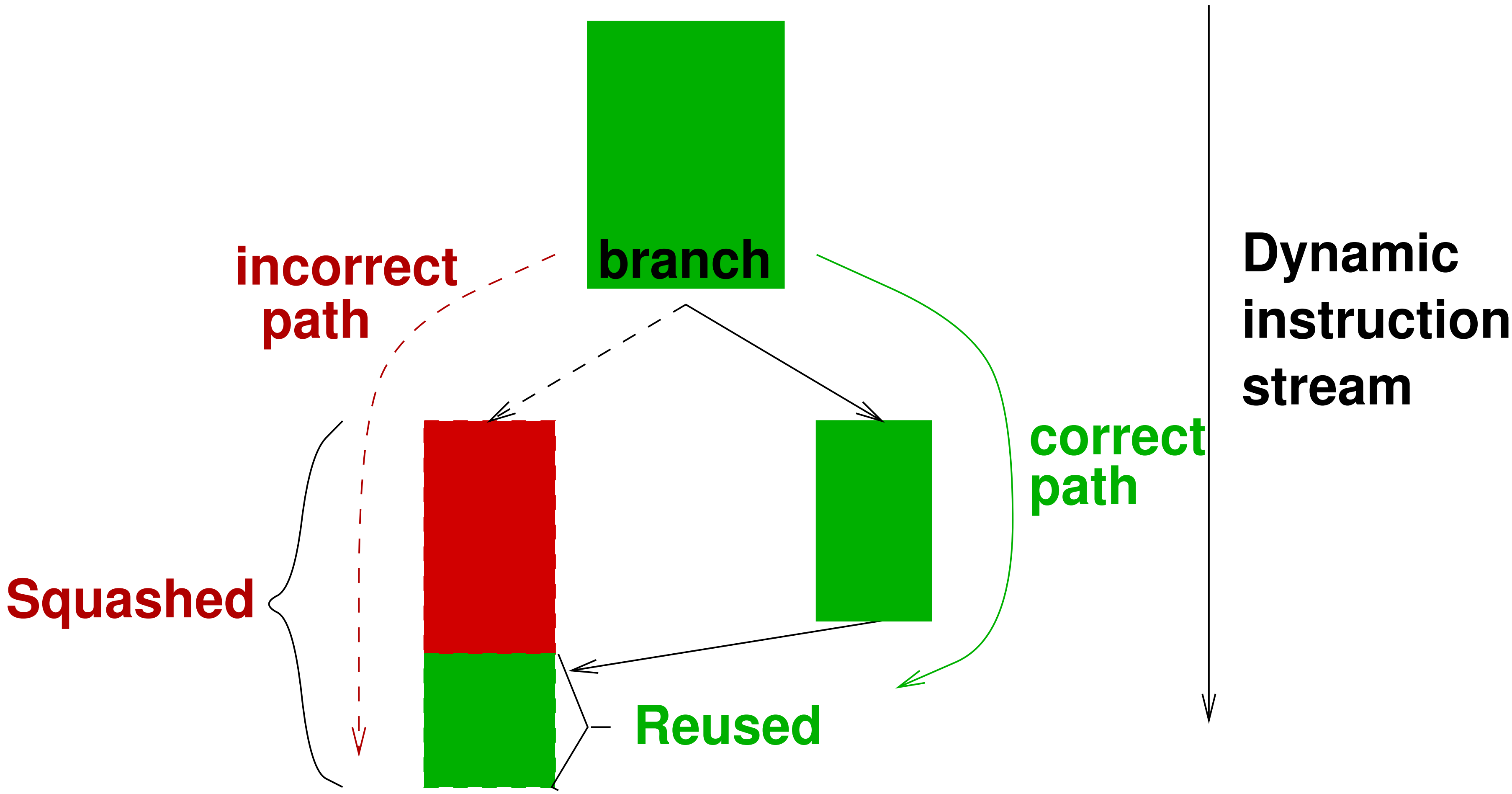
```
find (key, list)
  foreach element in list
    access element
    if (key == element) found
  not found
```



```
find (a, list)          find(b, list)
```



Squash Reuse



Outline

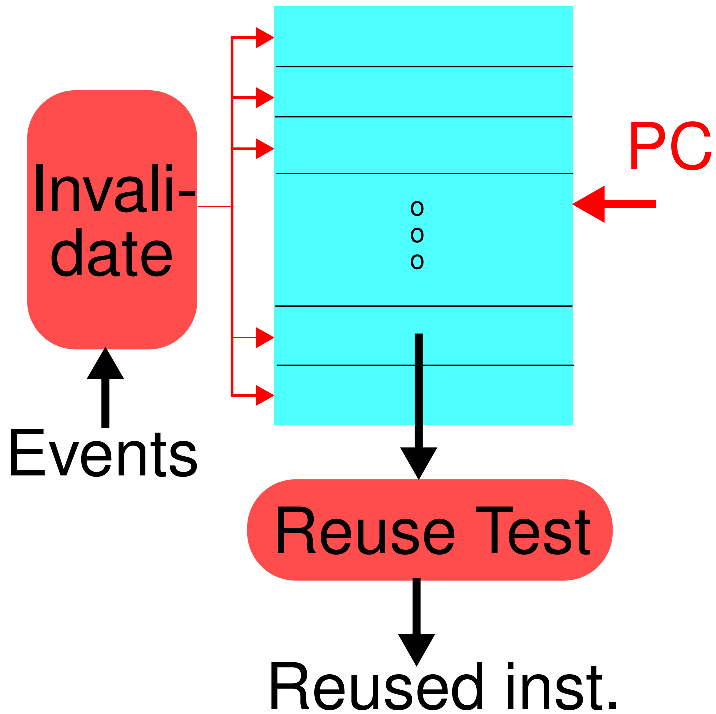
- ~~Motivation~~
- ~~What enables reuse?~~
- Implementing Reuse
- Three Reuse Schemes
- Some Results
- Summary

Implementing Reuse

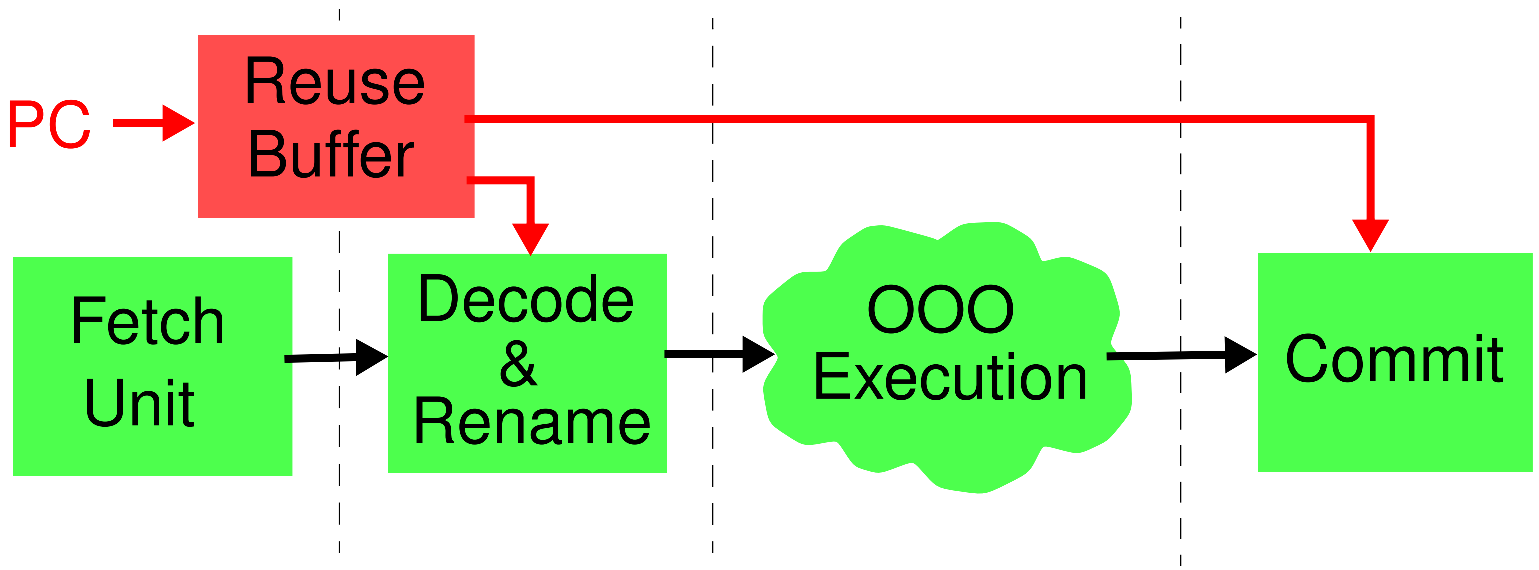
- Buffer results of instructions: Reuse Buffer (RB)
- Reuse if operands same as in previous execution: Reuse Test

RB

- Indexed by PC
- Reuse Test
- Selective invalidation



Integrating RB in pipeline



- RB access begins in fetch stage
- Reuse happens in decode stage

Issues

- What information stored in RB?
- How is Reuse Test done?
- How is the information kept consistent?

Reuse Schemes

- Scheme S_v : operand values
  - \+ most aggressive
  - \- lot of bits
- Scheme S_n : operand names
  - \+ few bits
  - \- too conservative
- Scheme S_{n+d} : operand names + dependences
  - \+ improvement on S_n
  - \+ S_v performance at (near) S_n cost

Scheme S_v

What to store in RB?

- Store results and operand values

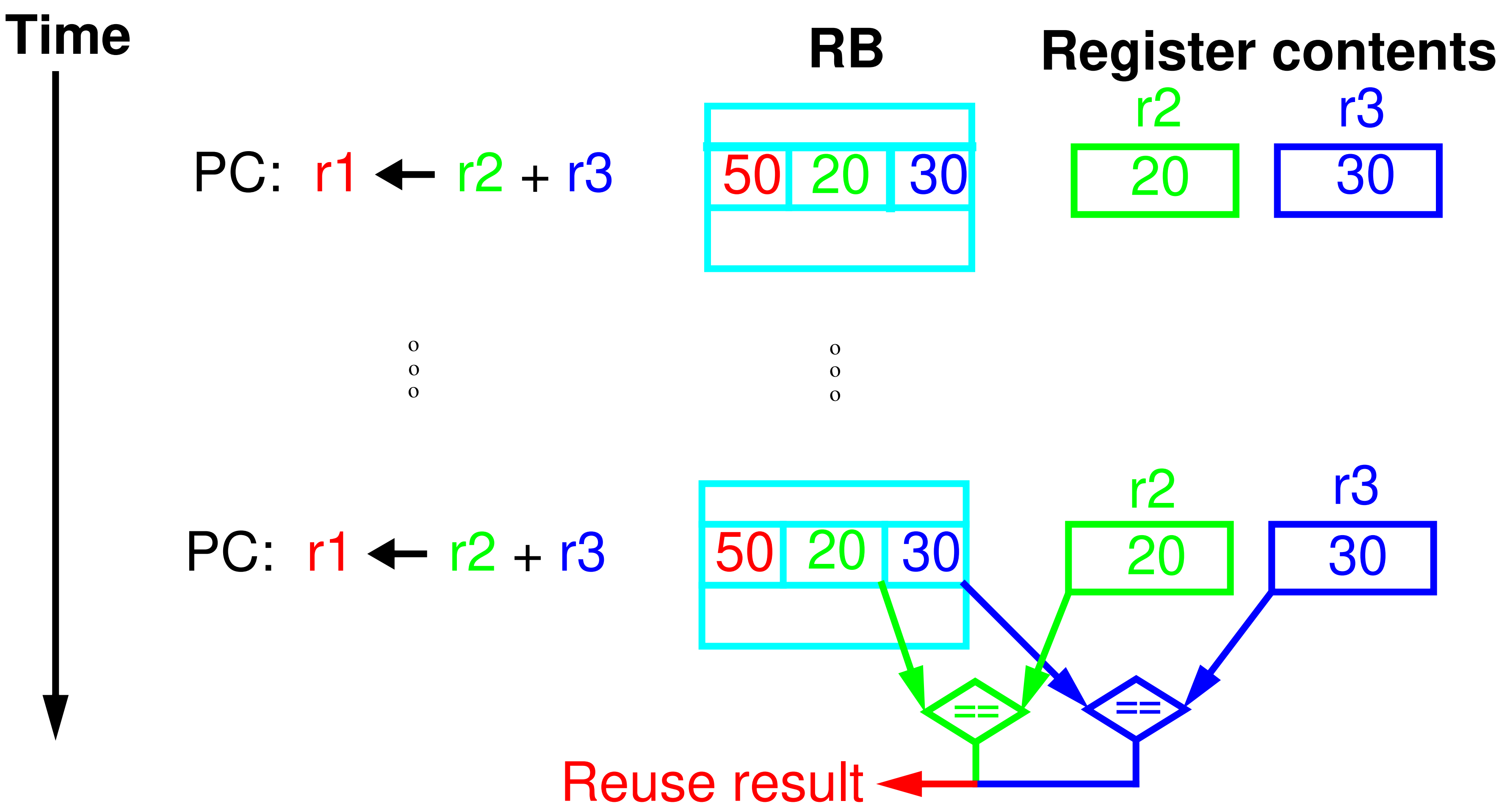
Reuse Test

- Reuse result if operand values are same

How to keep RB consistent?

- loads invalidated when memory location overwritten.
- other instructions not invalidated

Scheme S_V (cont'd)



Avinash Sodani
U. of Wisconsin-Madison

Scheme S_n

What to store in RB?

- Store result and operand names

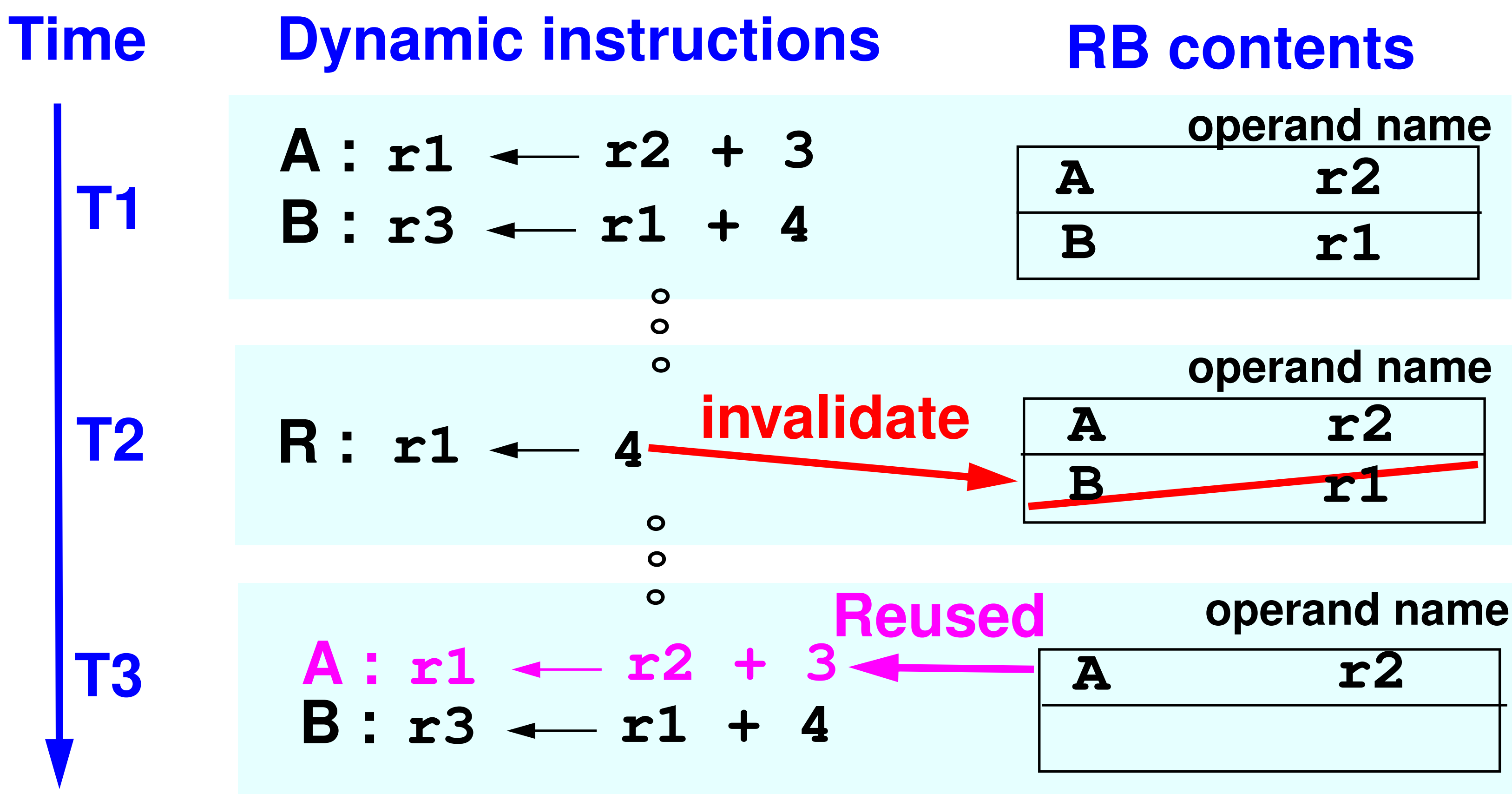
Reuse Test

- Reuse if result valid : valid bit

How to keep RB consistent?

- invalidate result when operand name overwritten

Scheme S_n (cont'd)



B performs same computation — but not reused by S_n

Scheme S_{n+d}

What to store in RB?

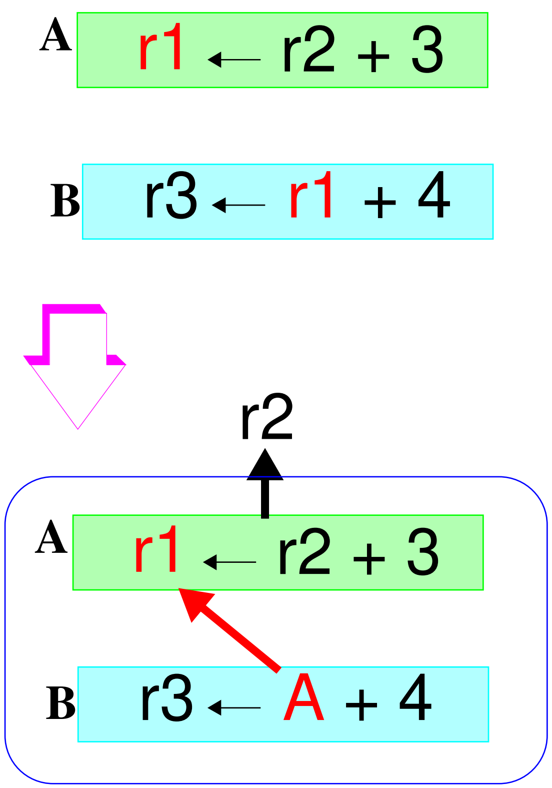
- Store result, name and dependences

Reuse Test

- A reused if valid : valid bit
- B reused if A is latest producer of r1

How to keep RB consistent?

- Invalidate chain when inputs overwritten



Scheme S_{n+d} (cont'd)

Time | **Dynamic instructions** | **RB contents**

T1

A : r1 ← r2 + 3
B : r3 ← r1 + 4

	operand name
A	r2
B	A

T2

R : r1 ← 4
(B not invalidated)

	operand name
A	r2
B	A

T3

A : r1 ← r2 + 3
B : r3 ← r1 + 4

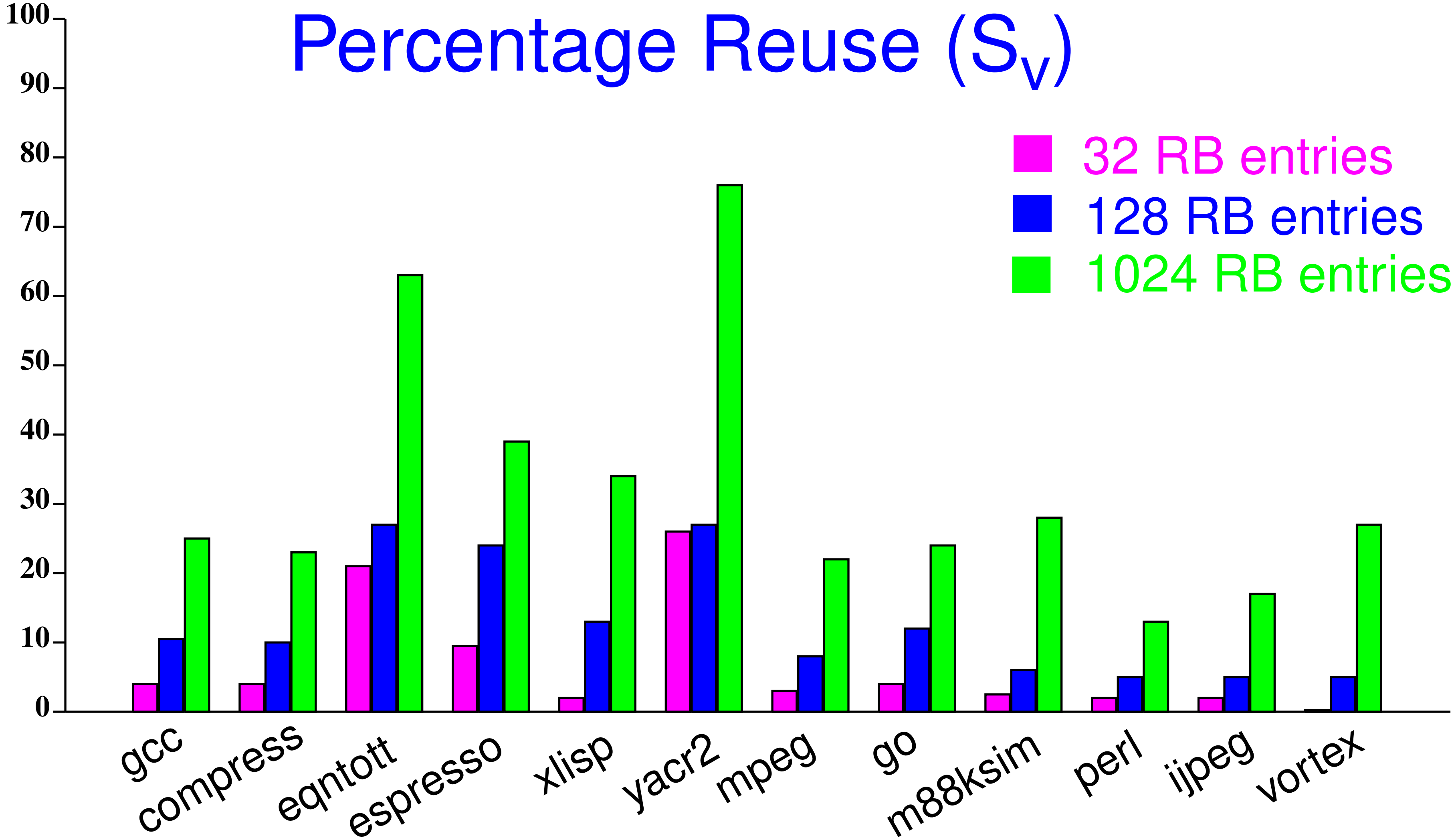
Reused

	operand name
A	r2
B	A

Dependent chain reused — possibly in the same cycle

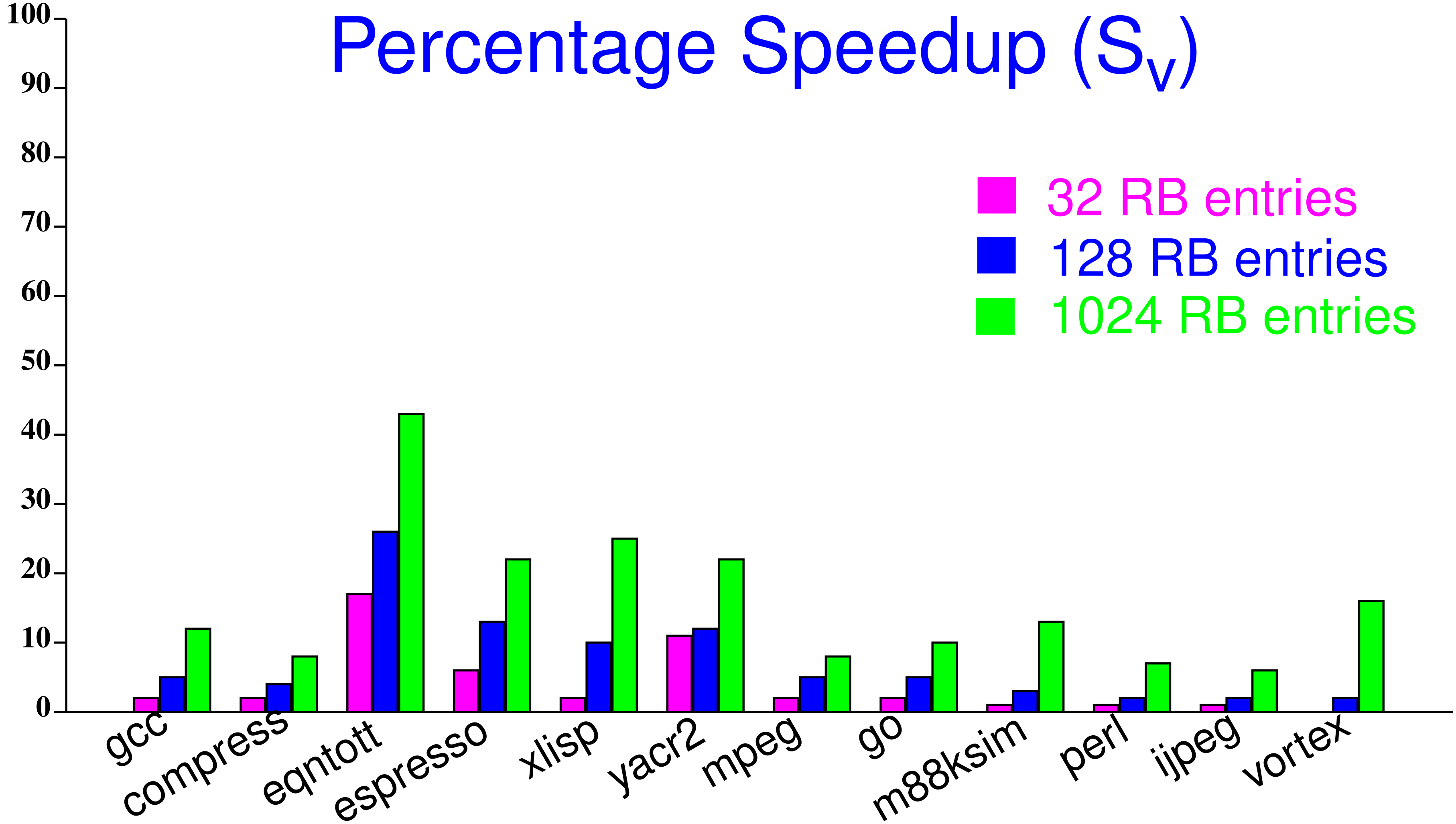
Experimental Evaluation

- OOO execution: window of 32 inst.: 4-way superscalar
- BTB: 2048 entries with 2-bit counters
- I-cache: 16K direct mapped, 32 byte line
- D-cache: 16K 2-way set assoc., 32 byte line
- Reuse Buffer
  - Size: 32, 128 and 1024 entries
  - Fully assoc. and 4-way set assoc. with FIFO replacement
  - 4 reads, 4 writes and 4 invalidations per cycle
- Benchmarks: Spec95 Int, Spec92 Int and others



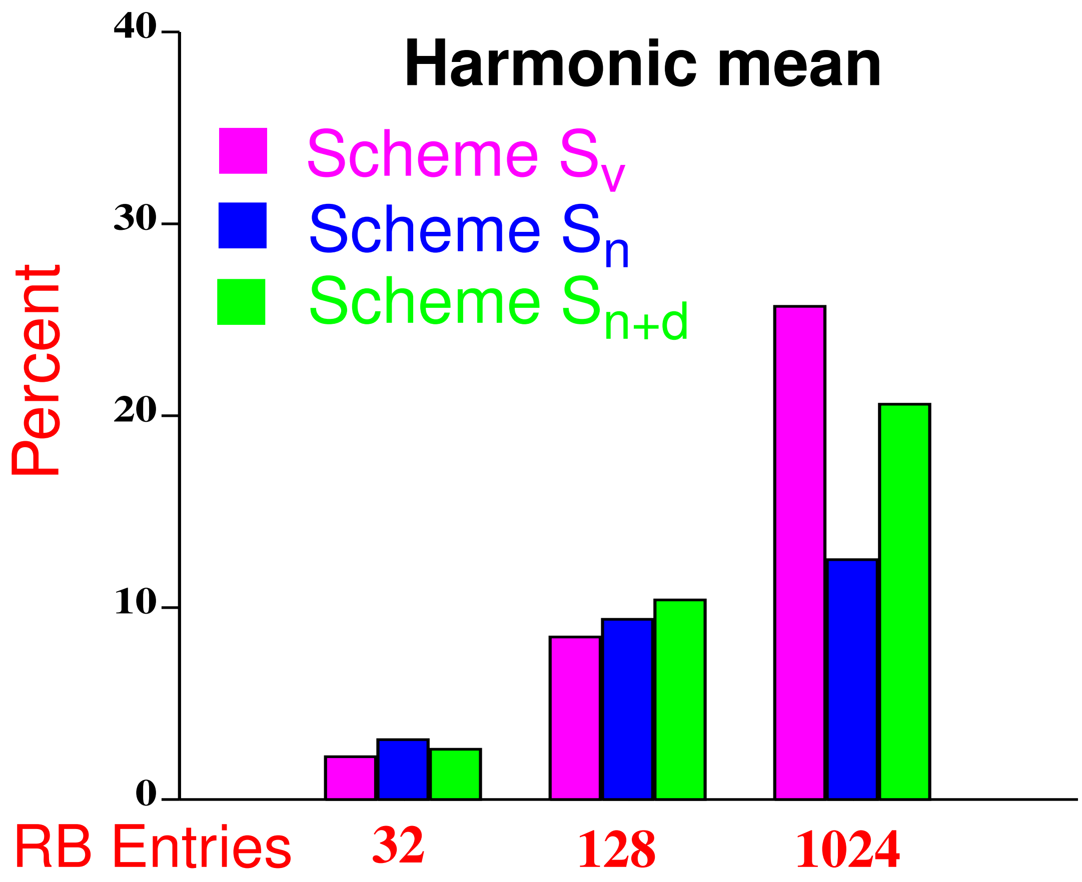
Significant reuse observed

Percentage Speedup (S_v)



Speedups significant too

Comparing Schemes



Summary

Instruction Reuse

- reduces critical path of the computation
- reduces contention for resources
- reduces mis-prediction penalty

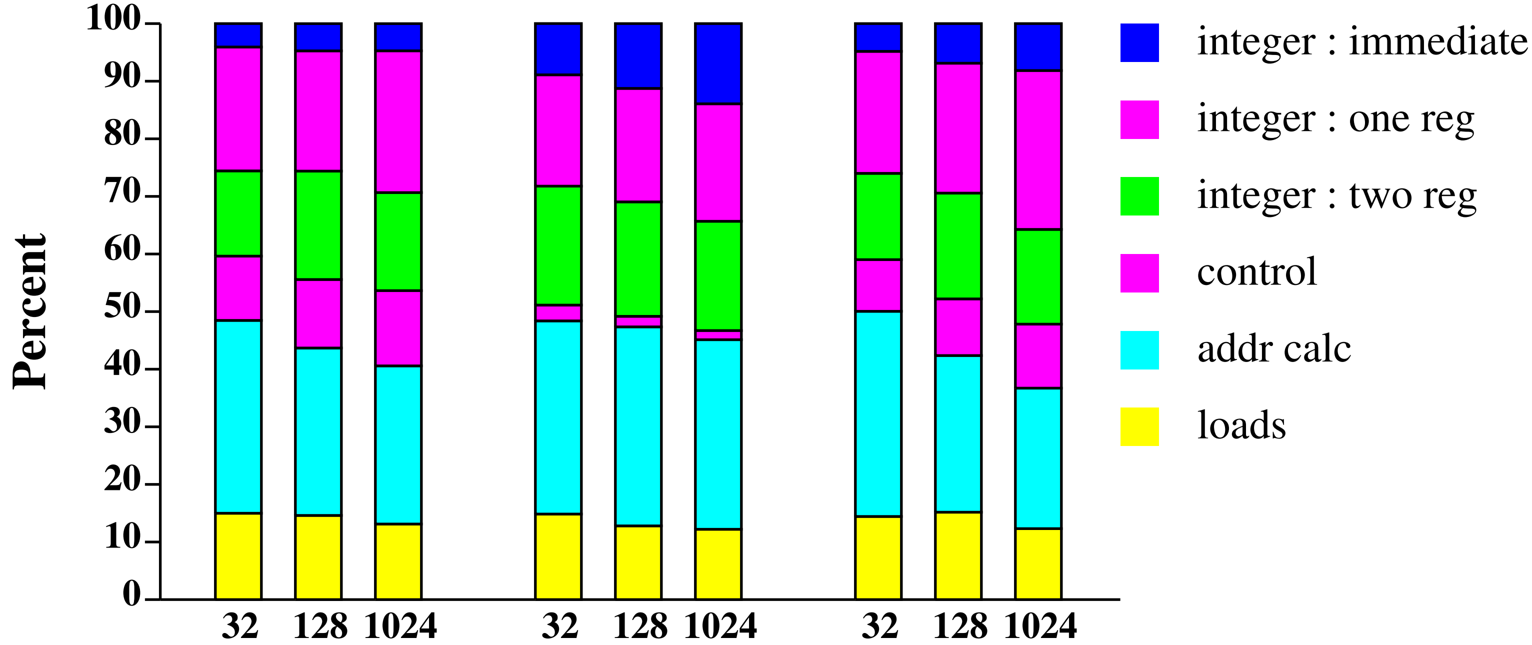
Significant instruction reuse

- in some cases > 50% : typically ~ 20%

Speedups also significant

- in several cases > 20% : typically ~ 10%

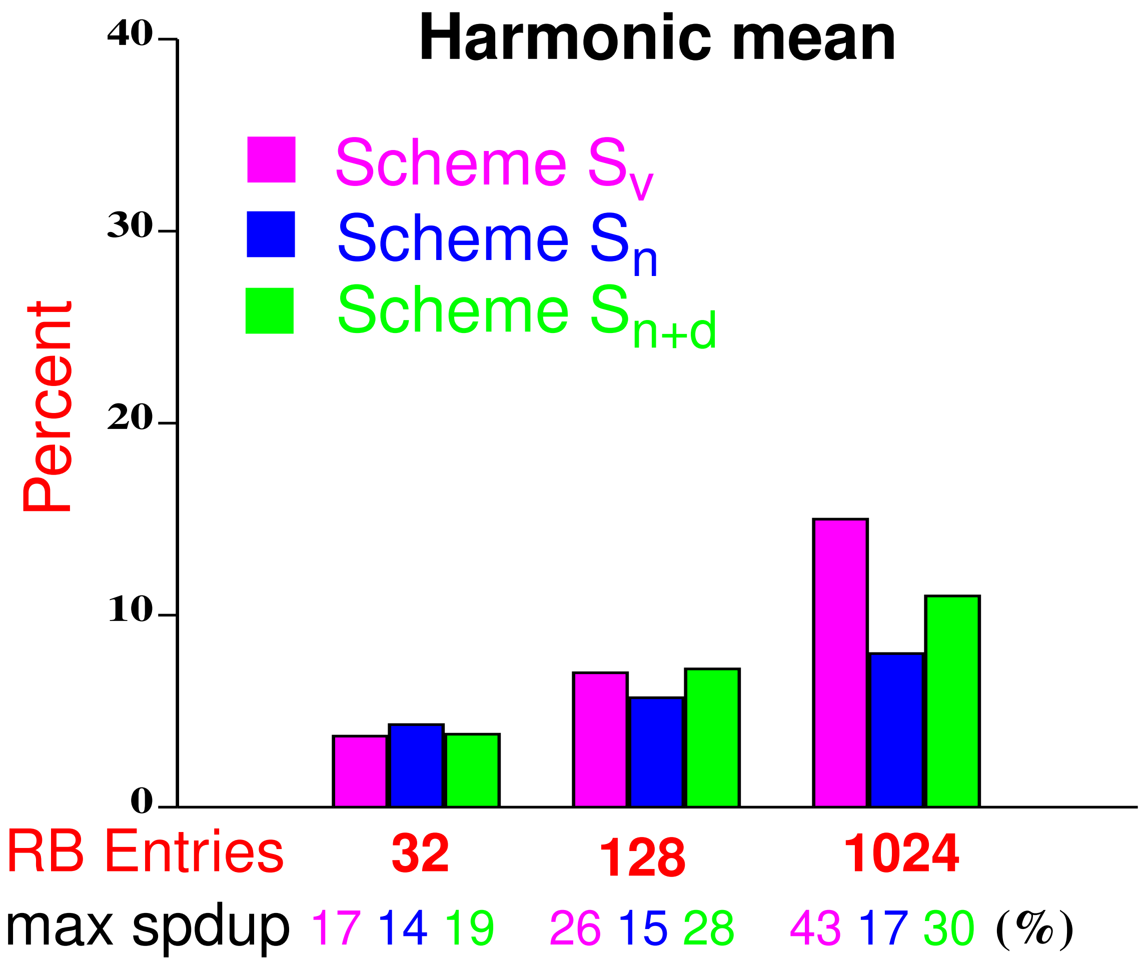
Reuse per Inst. Category



- Reuse prevalent among all instruction categories
- About 15% from load values
- About 25-35% from address calculation

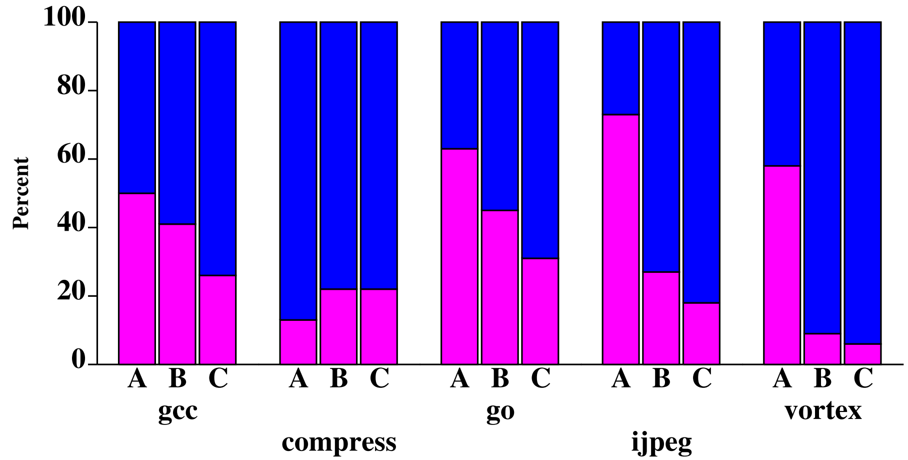
40-50% of Reuse

Mean Speedups

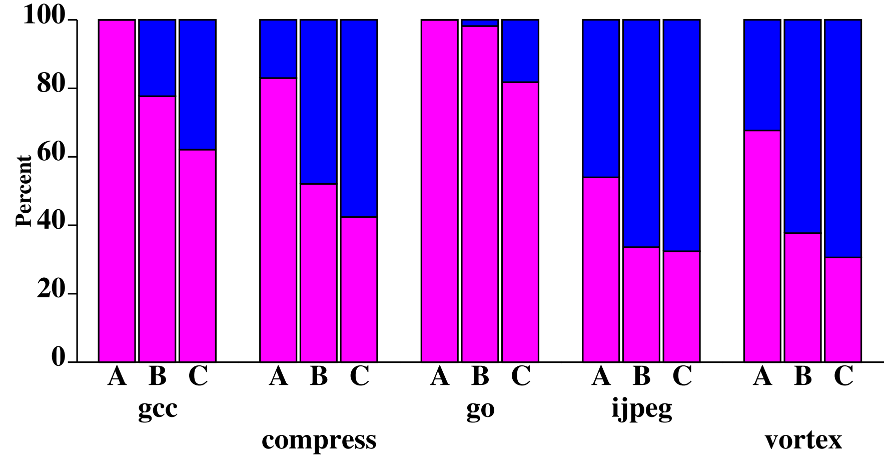


Squash Vs. General Reuse

Reuse



Performance



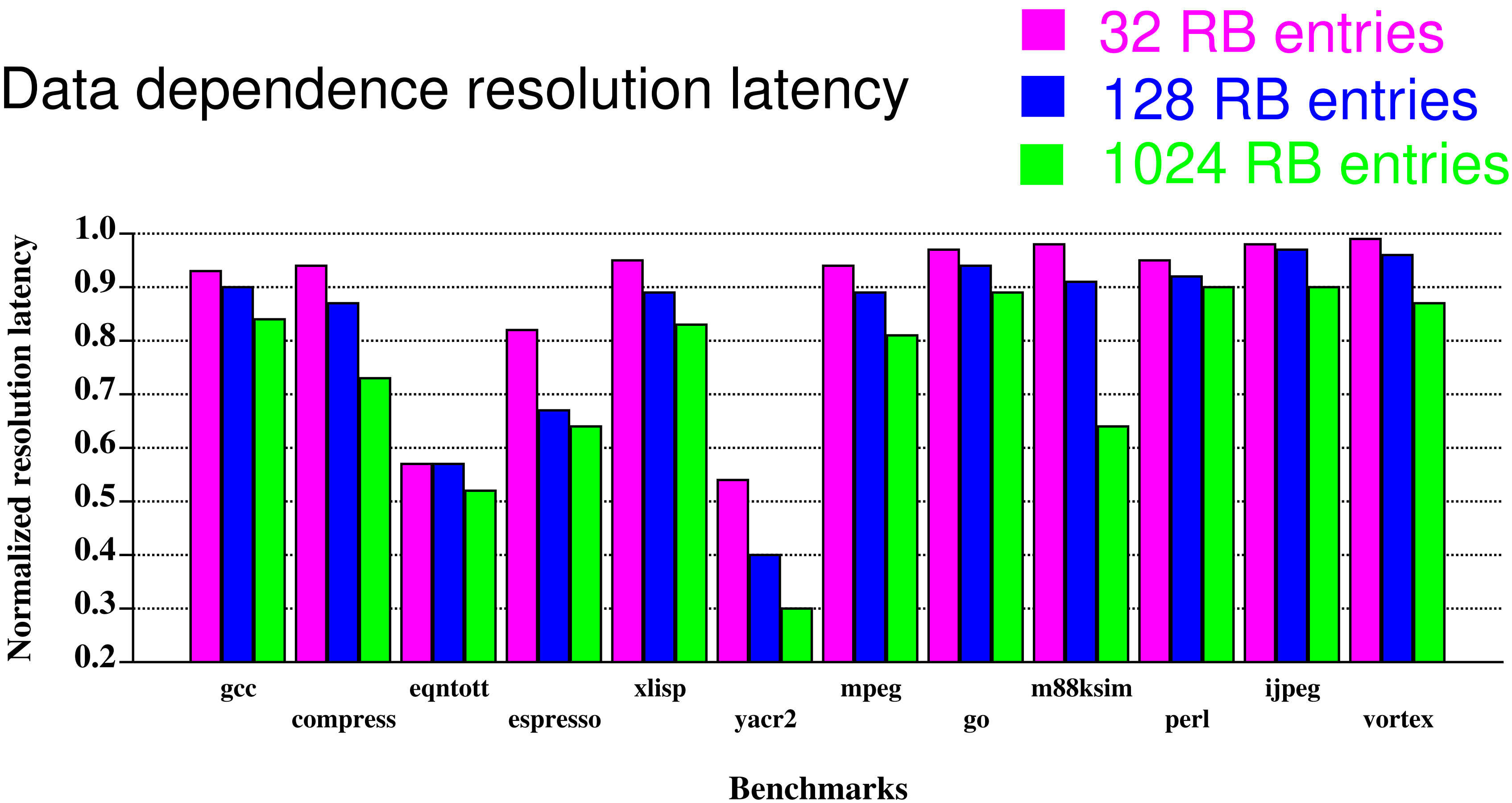
A = 32 entries **B = 128 entries** **C = 1024 entries**

squash reuse **general reuse**

- Recovering squashed work gives significant reuse.
- Squash reuse buys more — but general reuse important too

Collapsing true dependences

Data dependence resolution latency



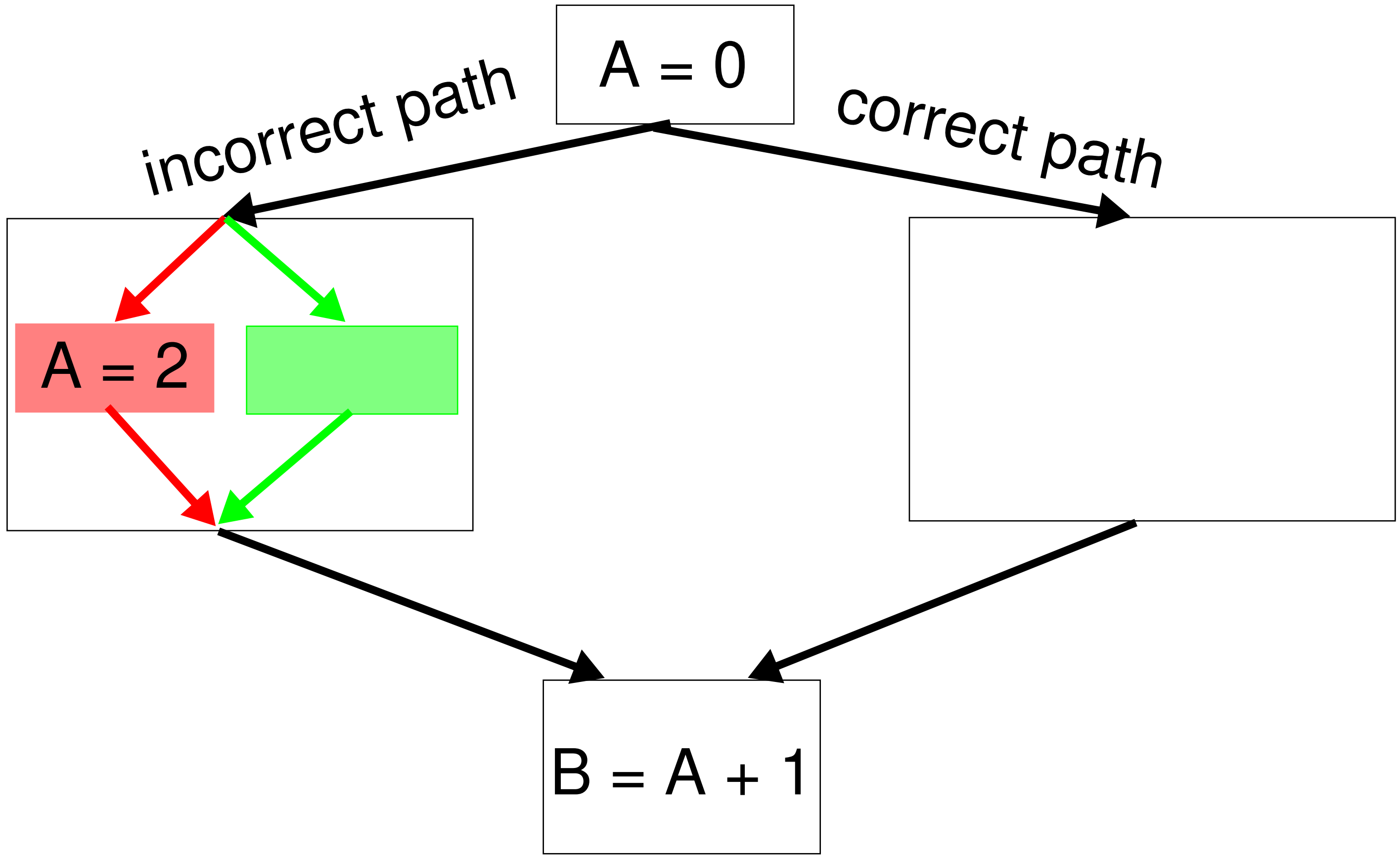
Related Work

- Harbison's value cache (*Tree machine*)
- Richardson's result cache.
- Oberman and Flynn's division and reciprocal caches

Key Difference

- They use address or operand values as index
  - limits the usefulness to long latency operations
  - Cannot reuse dependent chain in the same cycle

Squash Reuse



Control Flow Graph