Use-Based Register Caching with Decoupled Indexing

J. Adam Butts and Guri Sohi

University of Wisconsin–Madison

`{butts,sohi}@cs.wisc.edu`

ISCA-31

München, Germany

June 23, 2004

Motivation

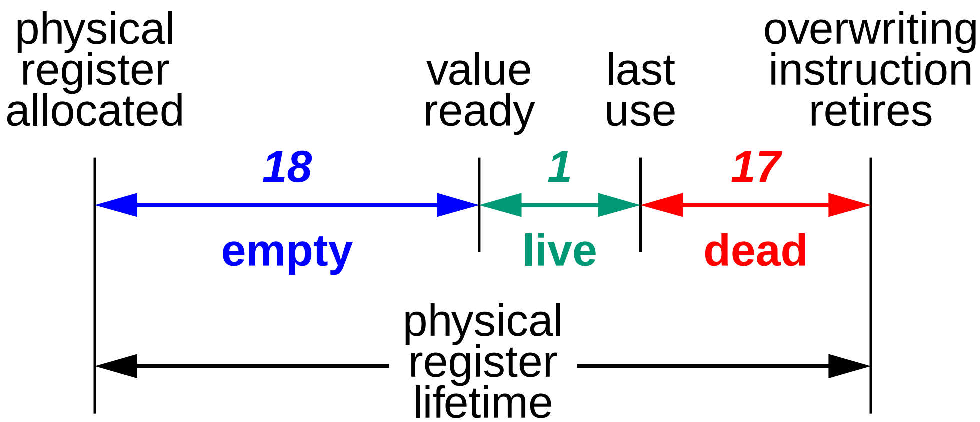
Need **large** register file

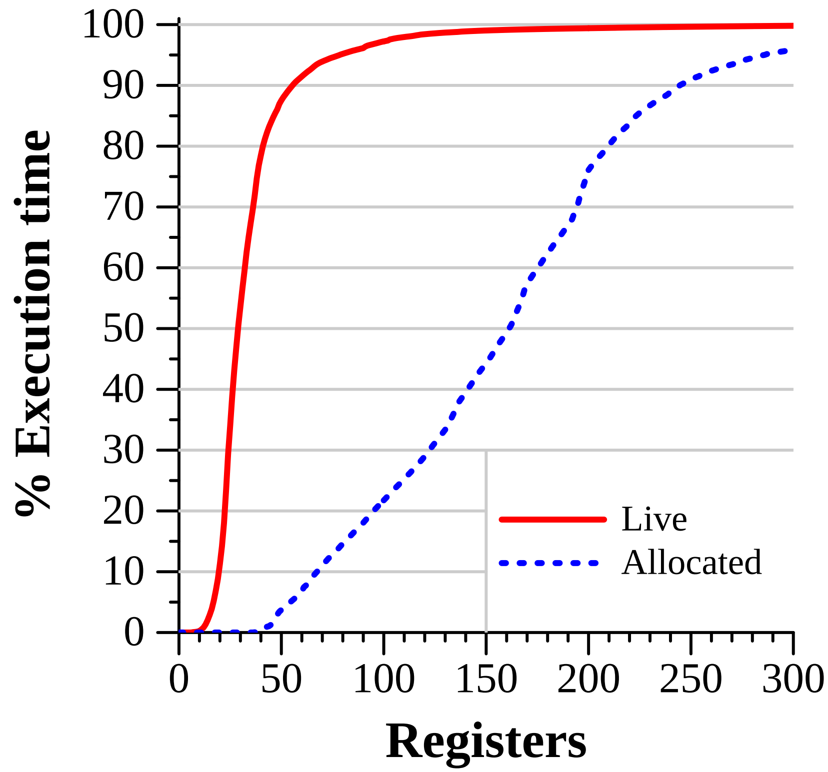
- **Deep**, **wide** pipelines
- In-flight instructions: O(**depth** × **width**)
- Many read and write ports: O(**width**)

Need **fast** register file

- Deep pipeline ⇒ **high clock frequency**
- Multi-cycle latency hurts IPC
- Performance penalty increases unless fully-bypassed
  - Complex bypass network: O(**depth** × **width**^{2})
  - Bypass network is **wire-dominated**

Motivation





Register values needed for small fraction of overall lifetime

- Long overall lifetime ⇒ many physical registers
- Fewer registers can hold just **live values**

Leverage small working set by caching registers

Overview

What values should be present in the register cache?

Values that have outstanding consumers

- Keep these few values in the small, low latency cache
- Manage cache contents via **use-based insertion, replacement**

How should values be placed within the register cache?

Assign cache sets to minimize conflicts

- Map register tags to cache indices intelligently
- Enables reasonable performance using a set-associative cache

Outline

Motivation and Overview

Register Caching

- Prior work: register hierarchies
- Register cache operation
- Shortcomings

Use-based Register Cache Management

Decoupled Indexing

Evaluation

Summary

Register Caches

Reduce average access latency

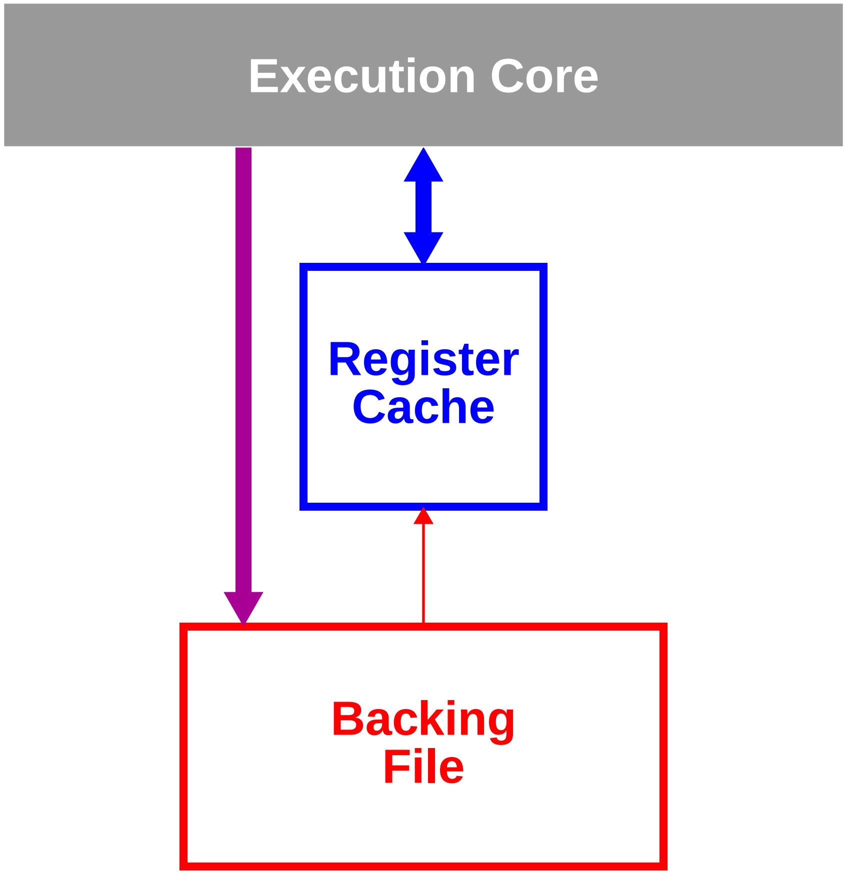
- Small, fast register cache
- Large, slow backing file

Interface to execution core

- Cache handles full core BW
- All values written to backing file
- Go to backing file on miss

Previous implementations

- Yung and Wilhelm [ICCD-’95]
- Cruz et al. [ISCA-’00]

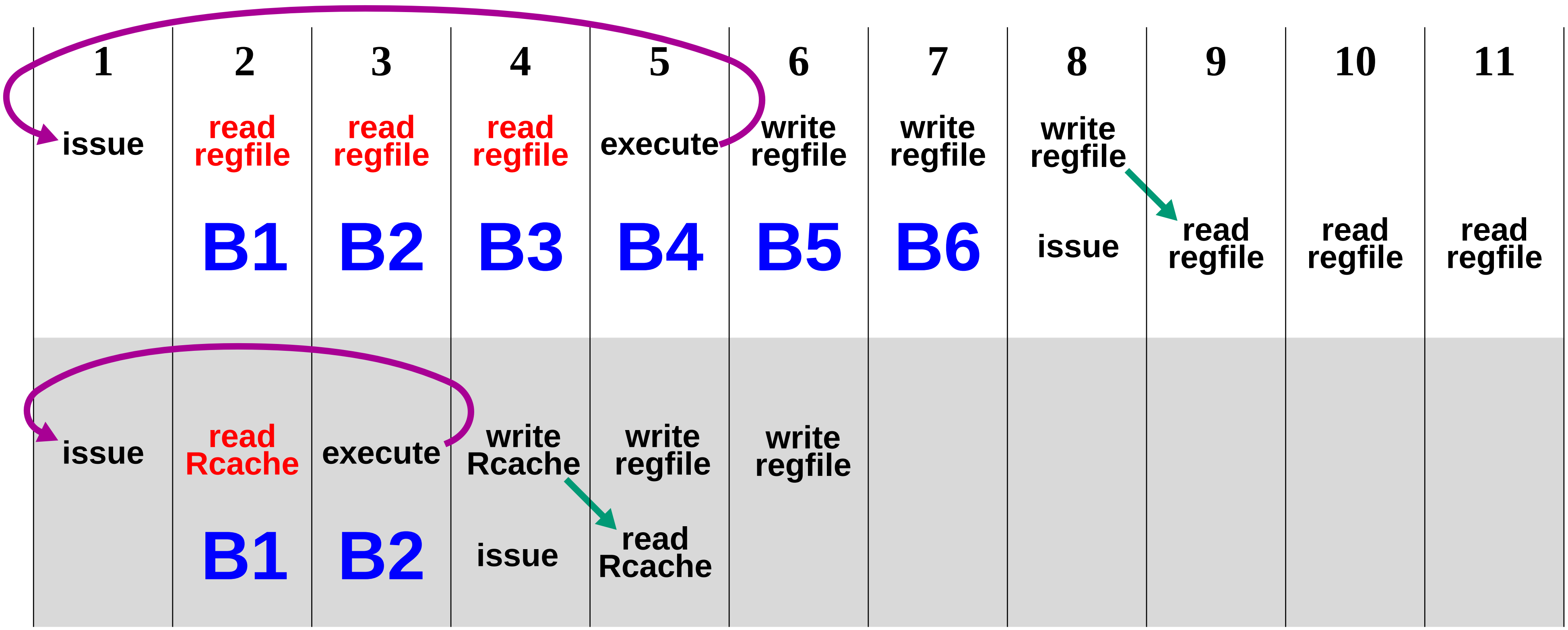


Register Cache Advantages

Lower issue to execute loop latency

- Data cache misses
- Branch mispredictions
- Load dependence replays

Smaller bypass network



Problems with Register Caching

Bad content management

- Poor insertion policies
- LRU replacement
- Leads to **frequent misses**

Fully-associative caches

- Required to obtain reasonable performance (**conflict misses**)
- Need many ports ⇒ **slow**

Outline

Motivation and Overview

Register Caching

Use-based Register Cache Management

- Ideal cache contents
- Insertion policy
- Replacement policy

Decoupled Indexing

Evaluation

Summary

Register Cache Contents

Ideally, cache values only during their live time

- Reads will only occur for these values
- But, **how to determine whether a value is live?**

Live values have remaining uses

- Begin with total number of **expected uses**
- Subtract uses as they occur to find ***remaining usefulness***
- Cache values with ***remaining usefulness***

Get expected uses using degree of use prediction

Degree of Use Prediction

Degree of use = # of consumers of a dynamic value

Degree of use prediction [MICRO-’02]

- History-based prediction
- Maintain a PC-indexed table of observed degree of use
- Associate path information with entries to improve accuracy

❶ **97% average accuracy with 9KB predictor**

❷ **Predictions available early: at rename of value producer**

Use-Based Filtering

Observation: some values bypass to all their consumers

- Avoid placing these values in the register cache

Insertion policy: bypass counting

- Write to cache only if number of bypasses < predicted degree of use
- **Filters** values from the cache, reducing capacity pressure

Compare with *non-bypass* proposed by Cruz et al. [ISCA-'00]

- Write to cache if value is not bypassed
- Assumes single-use values

Use-Based Victim Selection

Observation: LRU is poor

- Does not capture the behavior of register values
- Consider compiler **register allocation**

Replacement policy: Fewest-use replacement

- Store **remaining uses** with each value in cache
- Monitor subsequent uses, **update remaining use counts**
- Select victim with **fewest remaining uses**
  - Minimizes potential for future misses on victim
  - Replaced value frequently has **zero** remaining uses

Outline

Motivation and Overview

Register Caching

Use-based Register Cache Management

Decoupled Indexing

- Reducing conflict misses
- Set assignment algorithms

Evaluation

Summary

Decoupled Indexing

Fully-associative structures are slow

- So many ports, so little time...
- Fine, set-associative caches are faster

Problem: Conflict misses

- Standard cache index equals register tag modulo number of sets
- No spatial locality in physical register tag references
- ➜ Many live values can get mapped to the same set

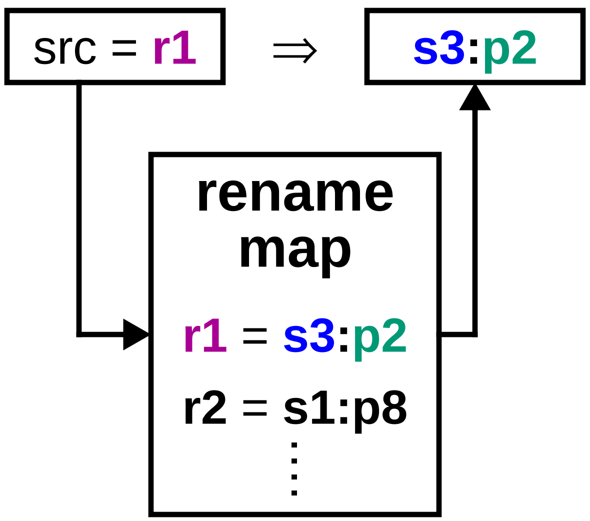
Solution: Decoupled indexing

- Assign set index intelligently to minimize conflicts
- Store full physical register tag in cache for hit detection

Decoupled Indexing

Rename source as before

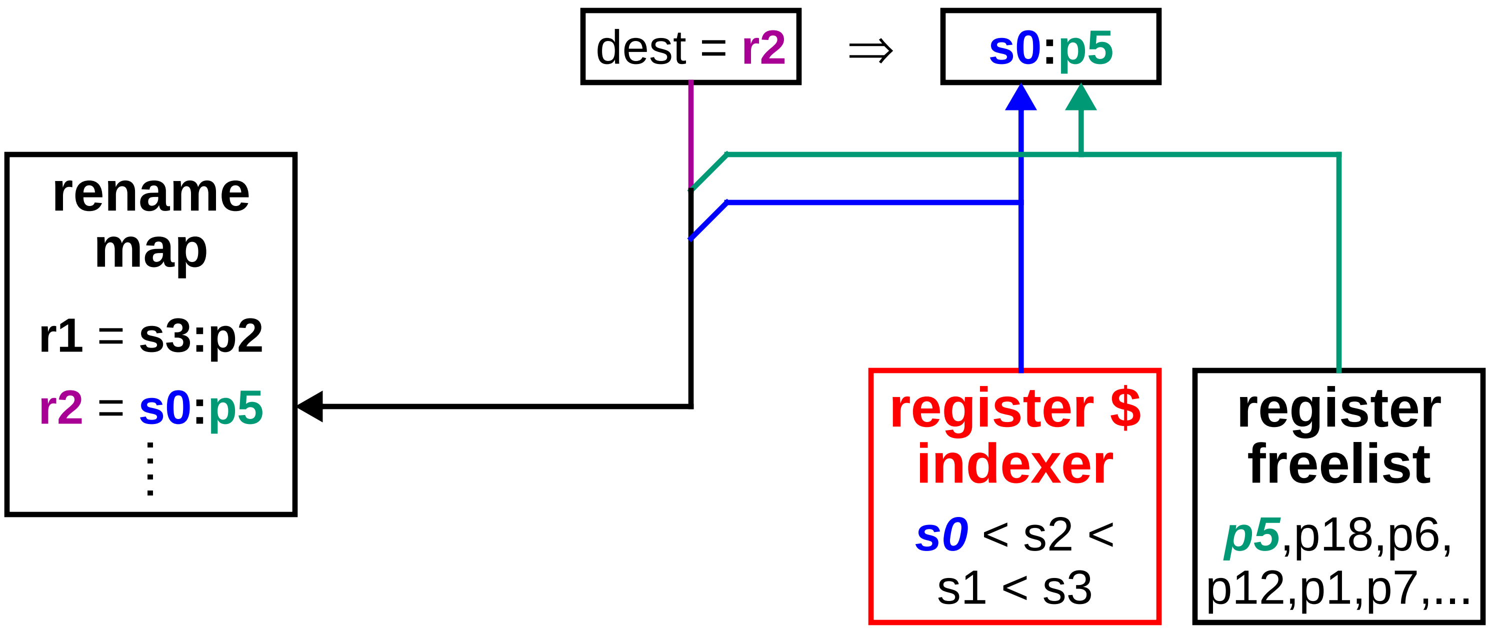
- Index into **augmented** rename map using **architectural register**
- Return **physical register** + **set index** to consumers



Decoupled Indexing

Allocate register cache set when renaming destination

- Obtain **free physical register** from freelist as before
- Also get **register cache set** from **indexer**
- Store both **physical register** and **cache set** in rename map



Set-Assignment Algorithms

Avoid assigning long-lived values to same cache set

Avoid excessive complexity

Can use predicted degree of use, set assignment history

Three algorithms tested

- Round-robin: assign values to sets sequentially
- Minimum sum: assign to set with fewest predicted total uses
- Filtered round-robin: RR, but skip sets with high-use values

Outline

Motivation and Overview

Register Caching

Use-based Register Cache Management

Decoupled Indexing

Evaluation

- Methodology
- Cache parameters
- Performance

Summary

Methodology

Execution driven simulation, SimpleScalar syscalls (trap to OS)

SPECInt 2000, training inputs, first 2 billion instructions

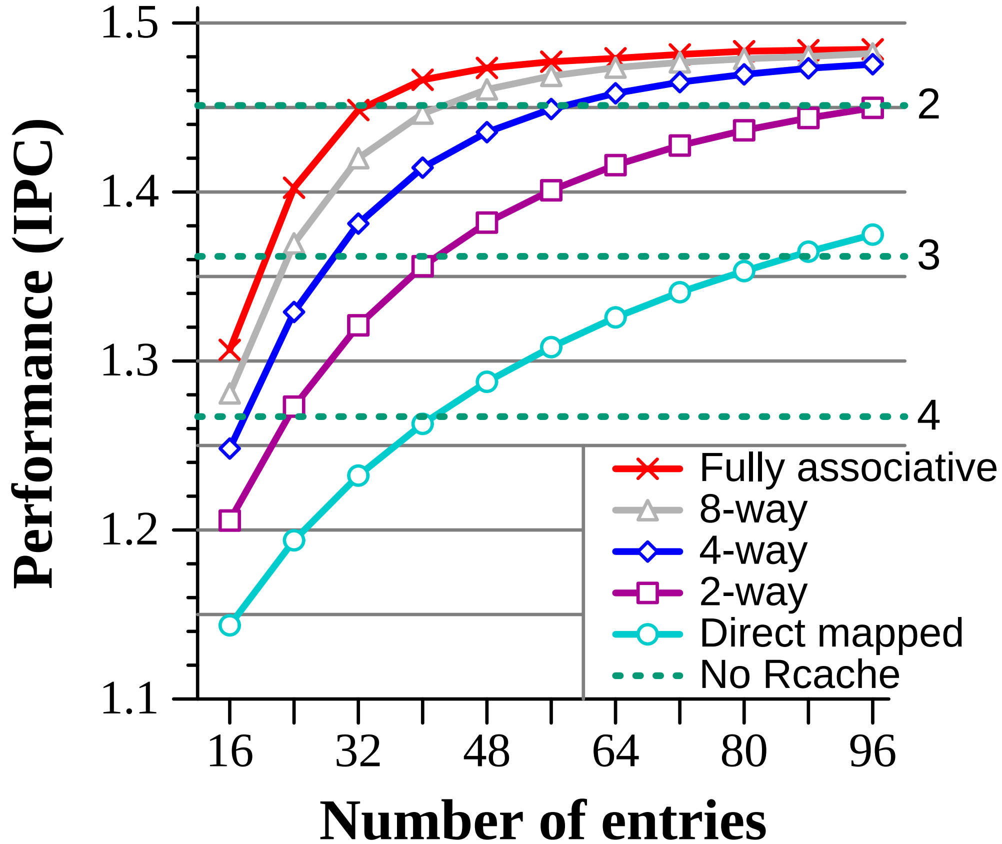
Aggressive baseline

- 8-wide issue, 512 instructions in-flight
- 15-cycle minimum fetch redirect

Register cache miss model

- Replay **all** operations issuing within one cycle (Alpha 21264-style)
- Block issue port for duration of miss resolution
- Re-issue delay to ensure complete writeback
- Contention for single read port

Register Cache Parameters



Larger caches than prior work

- 48-64 entries vs. 16
- Due to wider, deeper pipeline

Associativity is important

- Conflict misses
- Complicates victim selection

Decoupled Indexing

Use-based schemes work best

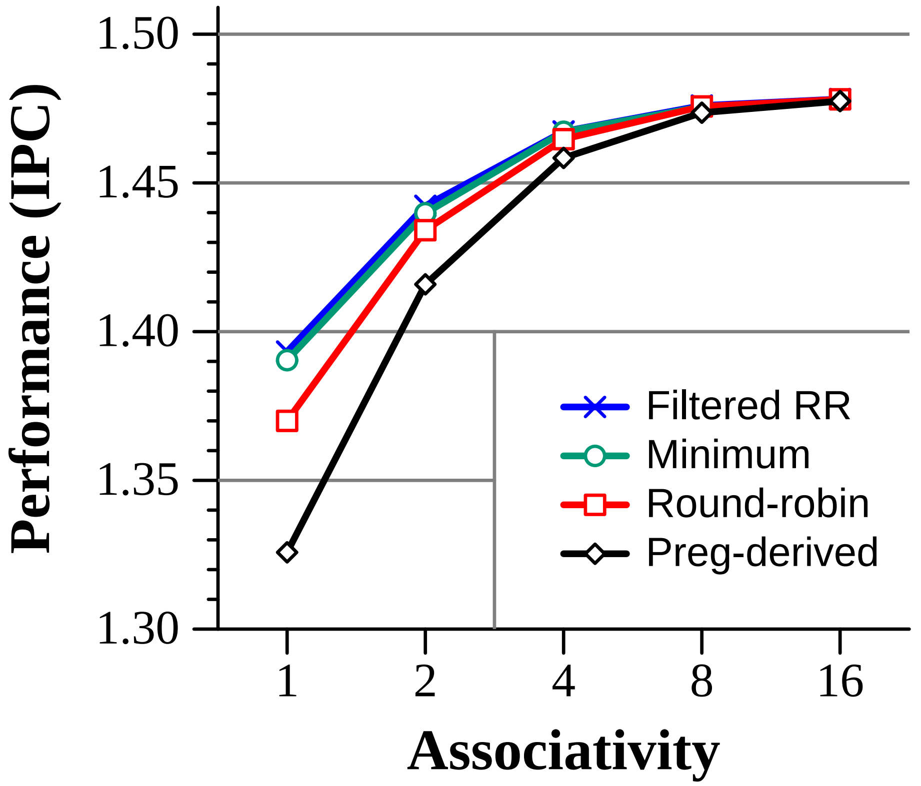
- Minimum, filtered round-robin
- +5% performance for DM cache

Simple round-robin does well

- Rename order ≈ execution order

Room for improvement

- 2-way still 2.5% short of FA performance
- Still many conflict misses...



Register Cache Miss Breakdown

Capacity misses unimportant

Decoupled indexing ⇒ 1/3 fewer conflicts

- (Filtered) round-robin
- Conflicts still **50% of misses**

LRU is not so good...

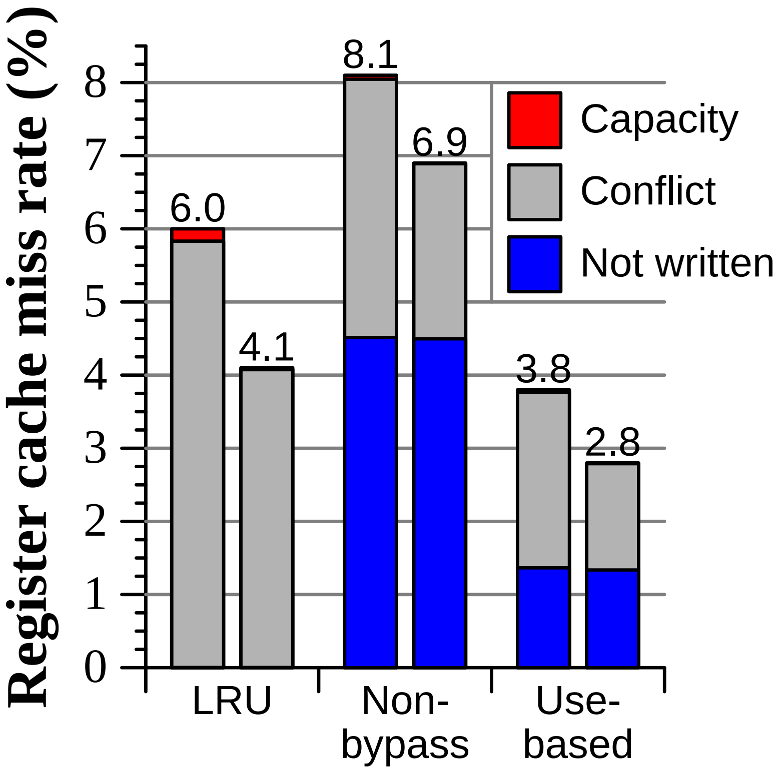
- **75%** cached values **never read**

Non-bypass is worse (!)

- **Reduces** capacity, conflict misses
- More **new misses** from write filtering

Use-based scheme

- Most benefit from insertion policy
- Nearly **60%** of values are **never cached**



Performance

Use-based caching superior

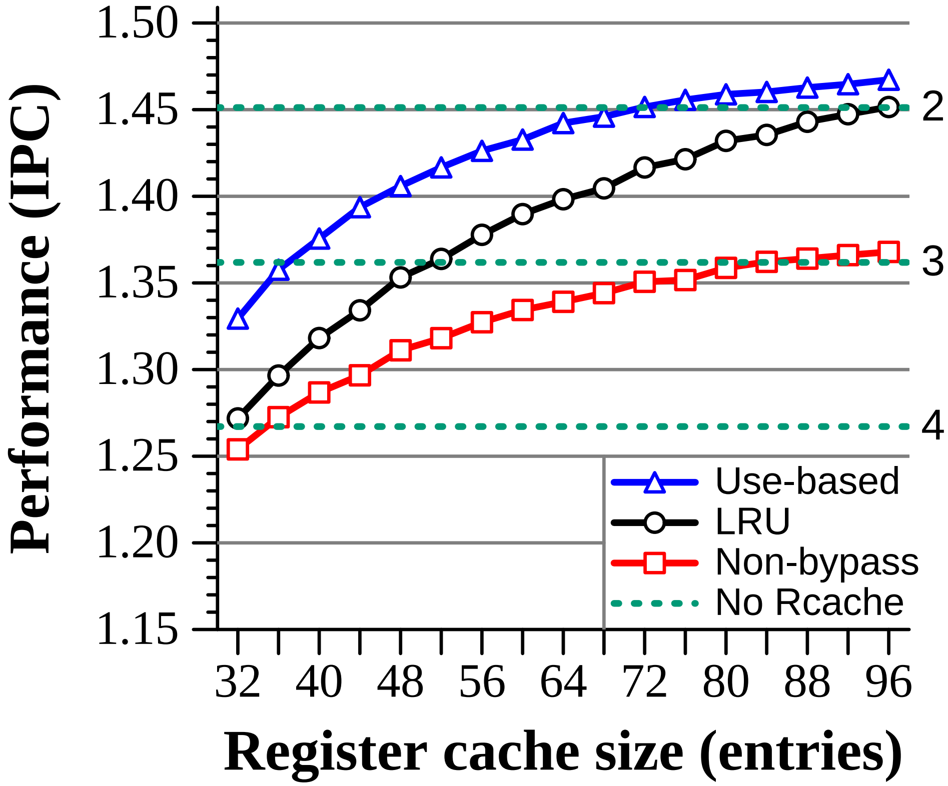
- Best perf. over size range
- **All** benefit from decoupled indexing

Small caches favor filtering

- Capacity misses dominate filtering misses
- **Non-bypass** surpasses **LRU** between 16-24 entries

Very large caches favor LRU

- Few capacity, conflict misses; **no** filtering misses
- Too large?



Outline

Motivation and Overview

Register Caching

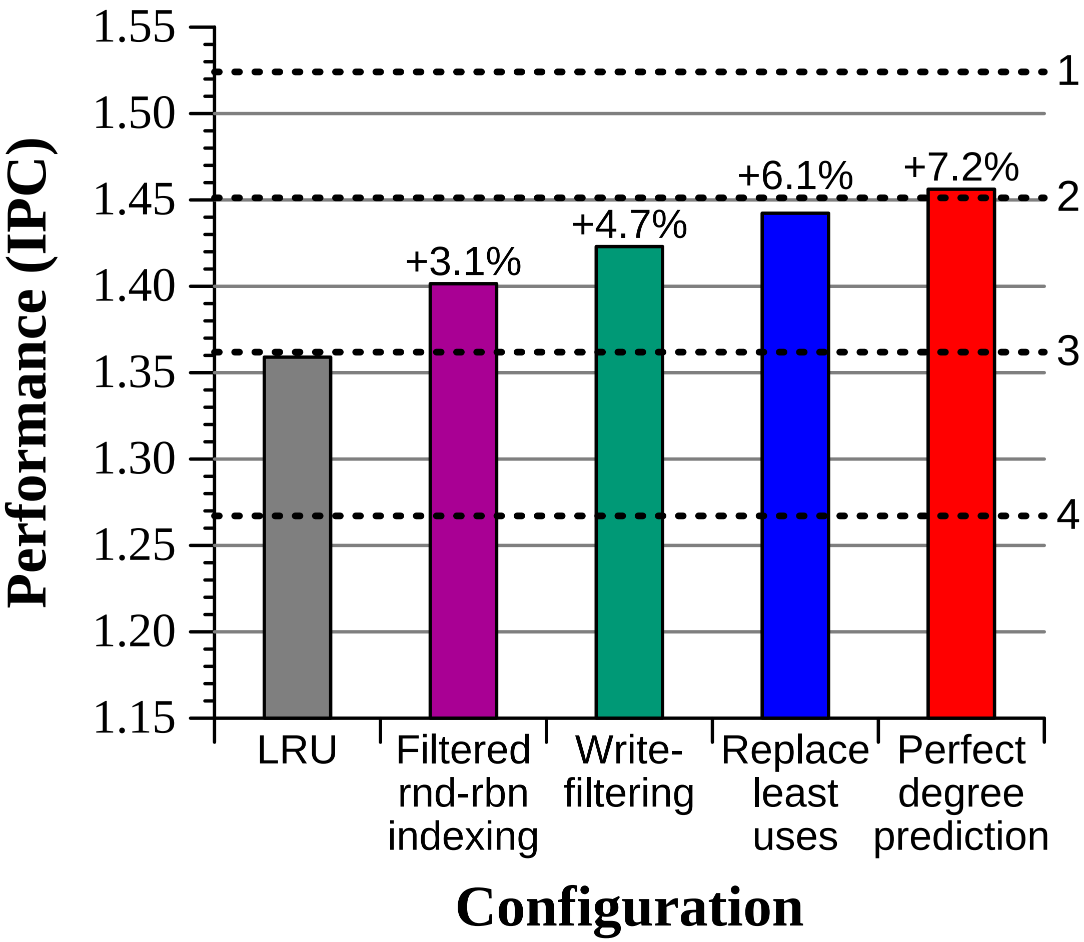
Use-based Register Cache Management

Decoupled Indexing

Evaluation

Summary

Incremental Performance Breakdown



Summary

Use-based register cache outperforms multi-cycle RFs

- Speculative use information enables good cache management
- Insertion policy filters needless writes
- Replacement policy chooses dead victims

Decoupled indexing facilitates low-associativity caches

Additional improvements possible

- **40%** of values cached are **never read**
- **50%** of misses from conflicts
- Other **50%** of misses due to bad write filtering