

Use-Based Register Caching

J. Adam Butts and Guri Sohi

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

University of Wisconsin–Madison

Architecture Affiliates Meeting

October 9, 2003

Motivation

Need **large** register file

- Deep, wide pipelines
- Many instructions in flight
- Many read and write ports

Need **fast** register file

- High clock frequency
- >1 cycle latency hurts IPC
- Complex bypass network

Motivation

Need **large** register file

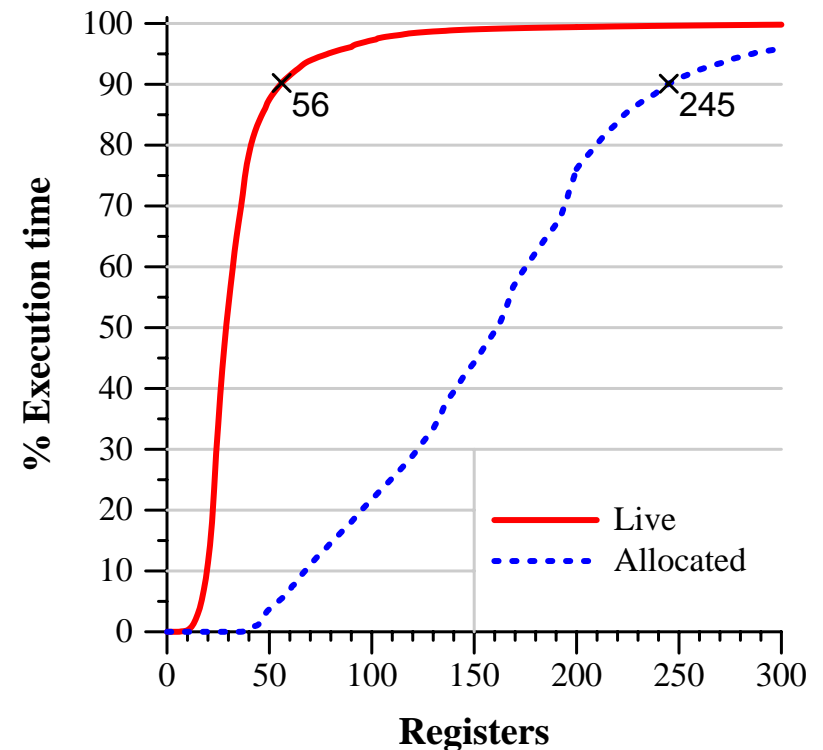
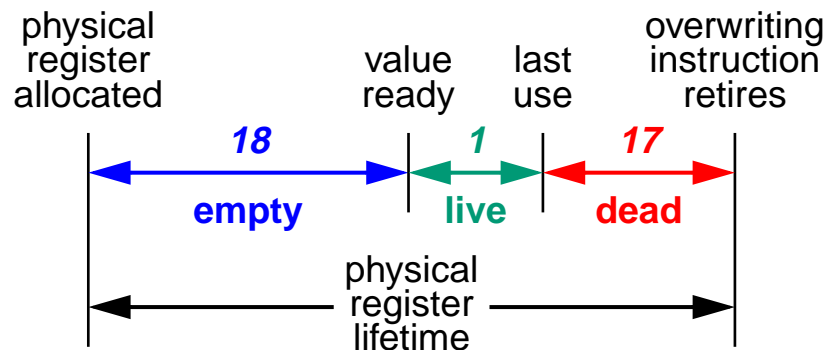
- Deep, wide pipelines
- Many instructions in flight
- Many read and write ports

Need **fast** register file

- High clock frequency
- >1 cycle latency hurts IPC
- Complex bypass network

Register values needed for small fraction of lifetime

- Few registers contain live values
- **Use a cache**



Overview

What register values should be present in the cache?

Values that have live consumers will be read in the future

- Keep these values close, others available
- **Degree of use** indicates **total** number of consumers
- Count uses as they occur to determine *future usefulness* of value
- Use *future usefulness* to make (re)placement decisions

How should values be placed within the cache?

Assign cache sets to minimize conflicts

- No meaning in physical register tags
- Map register tags to cache indices intelligently

Outline

Motivation and Overview

Register Caching

- Prior work
- Shortcomings

Use-based Register Cache Management

Decoupled Indexing

Evaluation

Conclusion

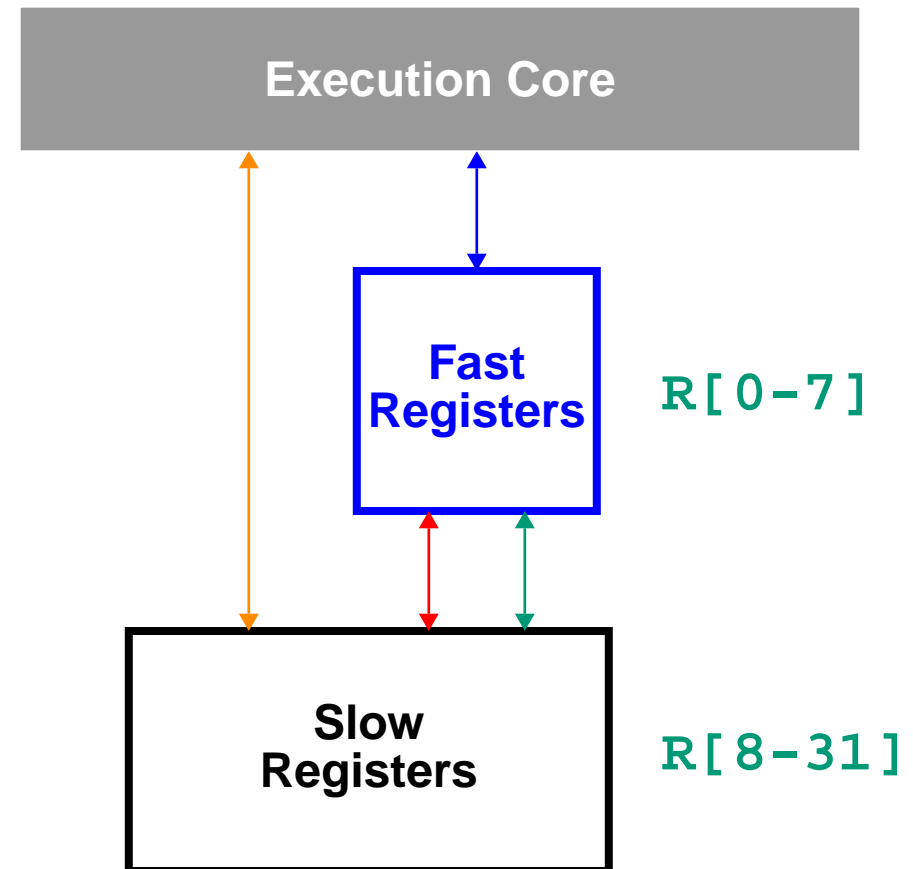
Register Caching

Reduce average access latency

- Like cache hierarchy
- Small, fast, high-level file
- Large, slow low-level file

Many variations

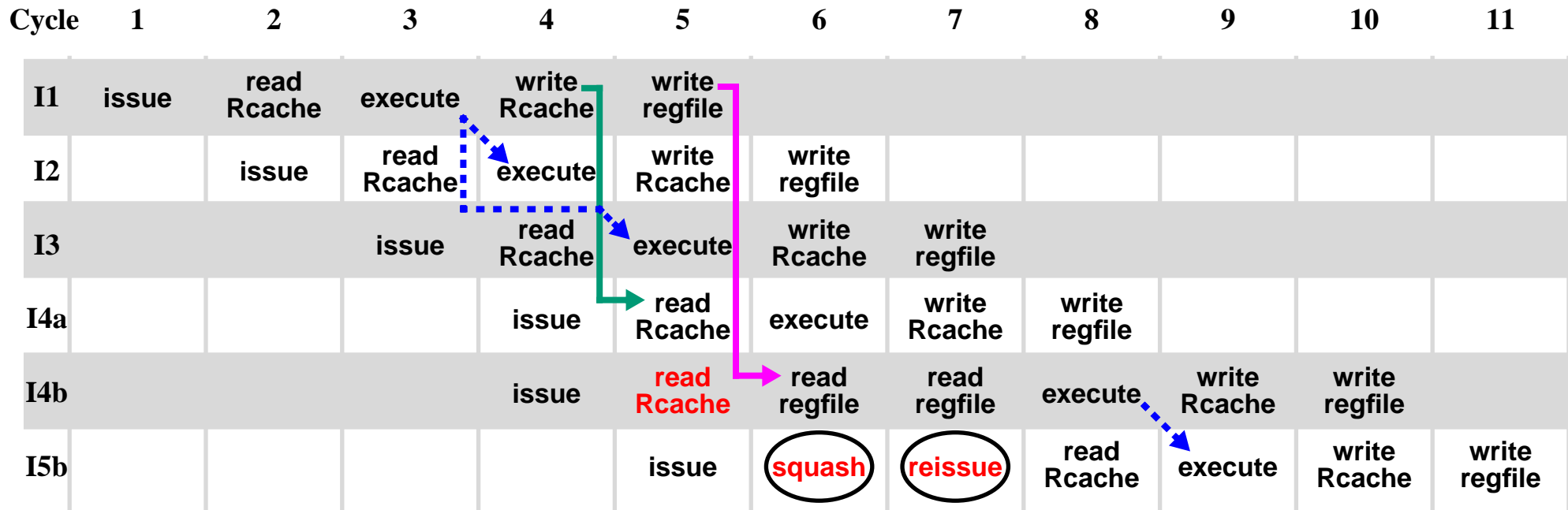
- Visibility to ISA
- Software vs. hardware management
- Supply values from both levels or only high-level register file
- Inclusion policy



Register Cache Pipeline

Like Yung and Wilhelm or Cruz et al.

- Hardware managed
- Cache fill on miss
- Values assumed to be in cache
- All values written to register file



Problems with Register Caching

Fully-associative caches

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

Poor content management

- LRU replacement
- Leads to frequent misses

Implementation **complexity**

- Expensive recovery mechanisms
- Many additional datapaths

Optimistic evaluations

- Cheap misses
- Unrealistic baselines

Outline

Motivation and Overview

Register Caching

Use-based Register Cache Management

- Insertion policy
- Replacement policies

Decoupled Indexing

Evaluation

Conclusion

Use-Based Cache Insertion

Observation: a subset of values *bypass* to all their consumers

- Avoid placing in the register cache values already communicated through the bypass network

Bypass counting

- Write to cache only if *number of bypasses* < *predicted degree of use*
- Store remaining uses with each value in cache
- Monitor subsequent uses (for use-based replacement)

Compare with *non-bypass* proposed by Cruz et al. [ISCA 27]

- Write to cache if value is *not bypassed*
- Assumes single-use values
 - Def-first use distance largely independent of degree of use
 - Subsequent consumers experience higher latency

Use-Based Victim Selection

Observation: LRU is **poor**

- Does not accurately capture the behavior of register values

Use-based replacement

- Use remaining uses stored in cache to select victim

Handling **unknown** numbers of remaining uses

- **Unknown default** when initial prediction unavailable
 - During **training** of degree of use predictor
 - Unknown default of **1** works well; **2** for larger cache sizes
- **Fill default** after register cache miss
 - Fill default of **0** performs best
 - Still need to fill!

Outline

Motivation and Overview

Register Caching

Use-based Register Cache Management

Decoupled Indexing

- Register cache set assignment
- Round-robin indexing
- Performance

Evaluation

Conclusion

Decoupled Indexing

Problem: Conflict misses

- Standard cache index equals **register tag** modulo number of sets
- No spatial locality in physical register tag references

Solution: Assign set index intelligently

- Augment rename map to hold **register cache index**
- Allocate set index with physical register using some algorithm
- Provide set index to consumers along with physical register tag

Algorithm considerations

- Avoid assigning **long-lived values** to same cache set
- Information available
 - Predicted number of uses
 - Set assignment history
 - Current front end status, performance

Round-Robin Indexing

Simple scheme to avoid conflicts

- Single state variable: **last assigned set**
- Assumes execution order resembles rename order

Round-Robin Indexing

Simple scheme to avoid conflicts

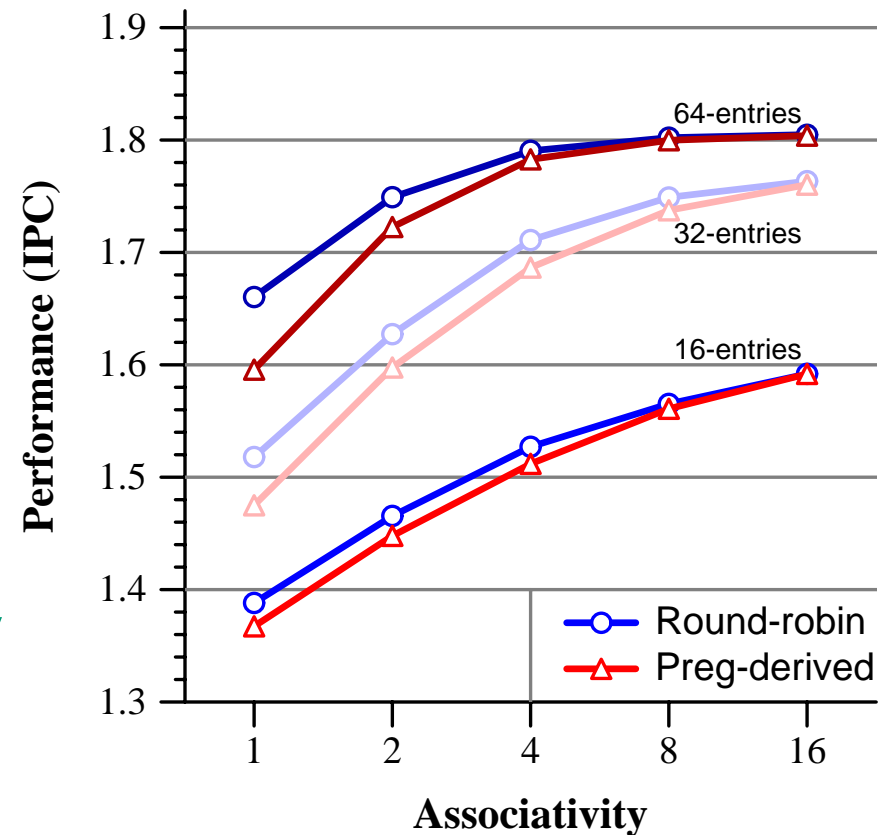
- Single state variable: **last assigned set**
- Assumes execution order resembles rename order

Advantage dependent on cache organization

- Helps more with **less associativity**
- **More sets** helps to a point

Room for improvement

- Data still indicates **25%** of misses due to conflicts
- Use-based set assignment?



Outline

Motivation and Overview

Register Caching

Use-based Register Cache Management

Decoupled Indexing

Evaluation

- Methodology
- Cache parameters
- Performance

Conclusion

Methodology

Simulator

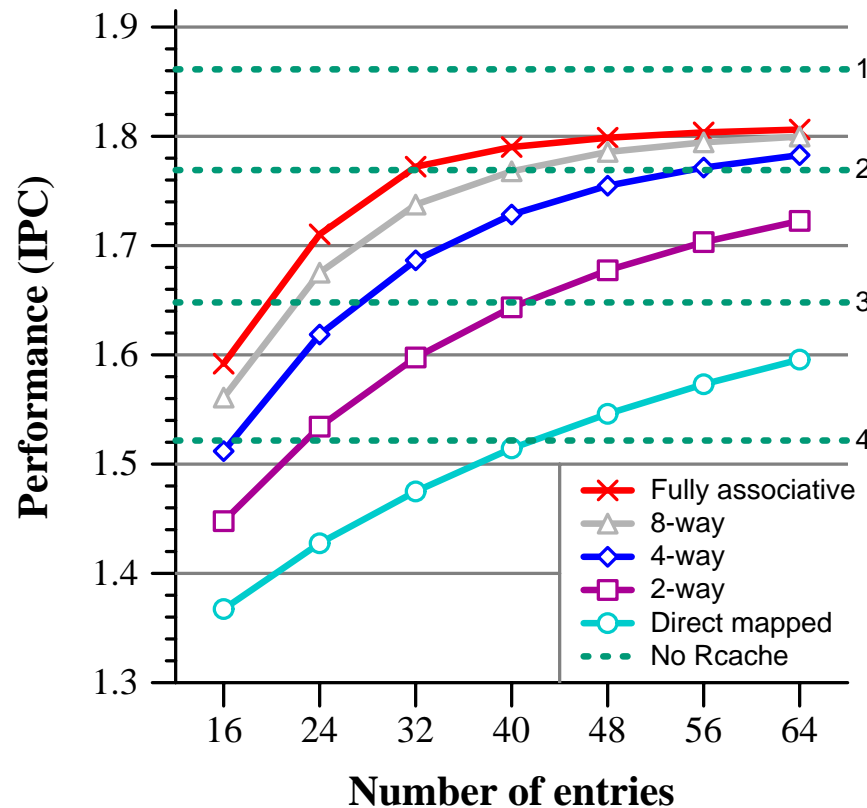
- Execution driven, SimpleScalar syscalls (trap to OS)
- 512 instructions in-flight, 128-instruction window, 8-wide issue
- 15-cycle minimum fetch redirect, 12 KB YAGS, 9KB DOU predictor
- 32 KB 2-way L1 (4), 1MB 4-way L2 (12), 180 cycles to memory

SPECInt 2000, training inputs, 1 billion instructions

Register cache miss model

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

Register Cache Tuning



Associativity is important

- 4-way minimum
- Capacity can compensate
- **Conflicts**

Larger caches than prior work

- 48-64 entries vs. 16
- Due to wider, deeper pipeline
- **Use 48-entry, 6-way**

Register Cache Miss Breakdown

LRU is bad

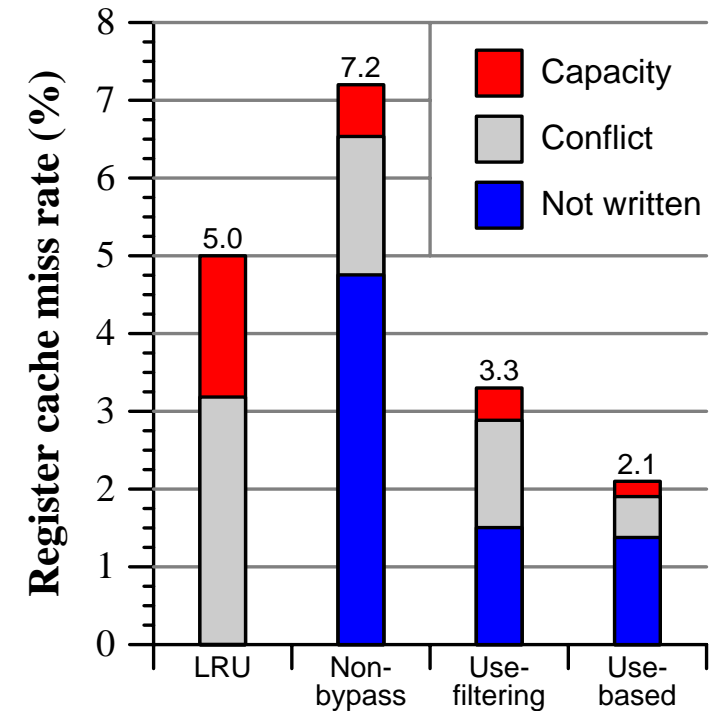
- No write-filtering (75% never read)
- Many capacity & conflict misses

Non-bypass is worse (!)

- Reduces capacity and conflict misses
- But, larger increase in misses from write filtering

Use-based scheme is superior

- Insertion policy reduces capacity and conflict misses
- Only with small increase in misses from write filtering
- Replacement policy reduces misses from premature evictions of useful values



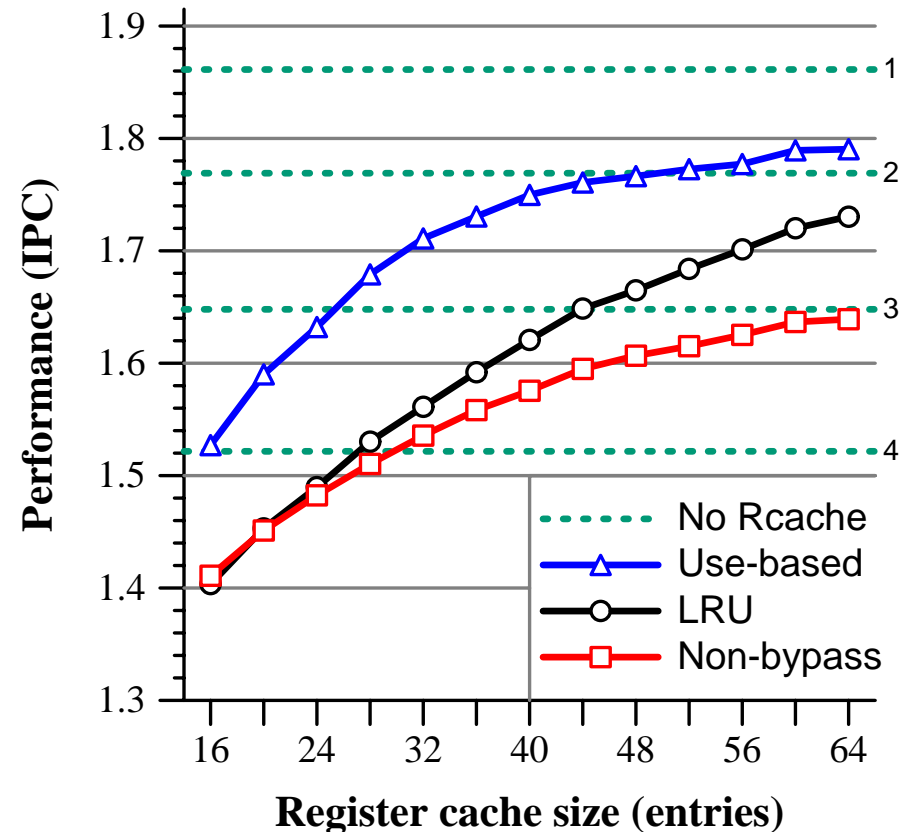
Performance vs. Cache Size

Small cache sizes favor filtering

- Net gain from filtering unneeded values
- Non-bypass surpasses LRU for caches with 16-24 entries

Very large cache sizes favor LRU

- **Not** due to replacement policy!
- No misses from incorrect filtering
- Large cache \Rightarrow low capacity/ conflict miss rate
- Too large to be much benefit as a cache



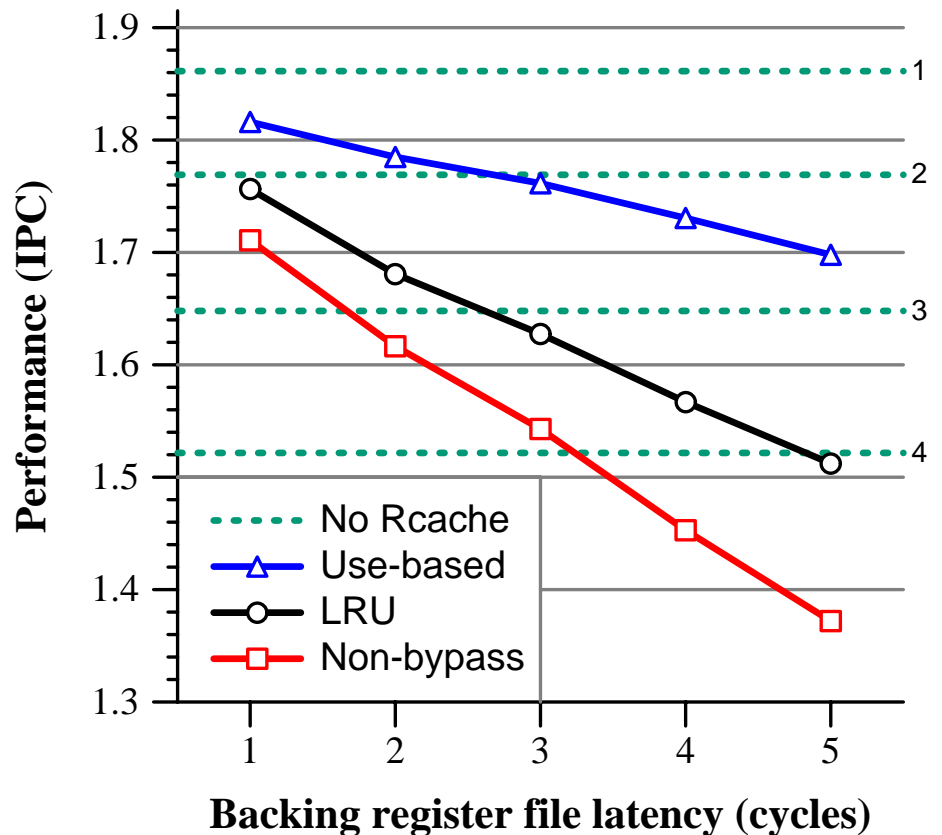
Sensitivity to Register File Latency

Use-based register cache
exhibits least sensitivity

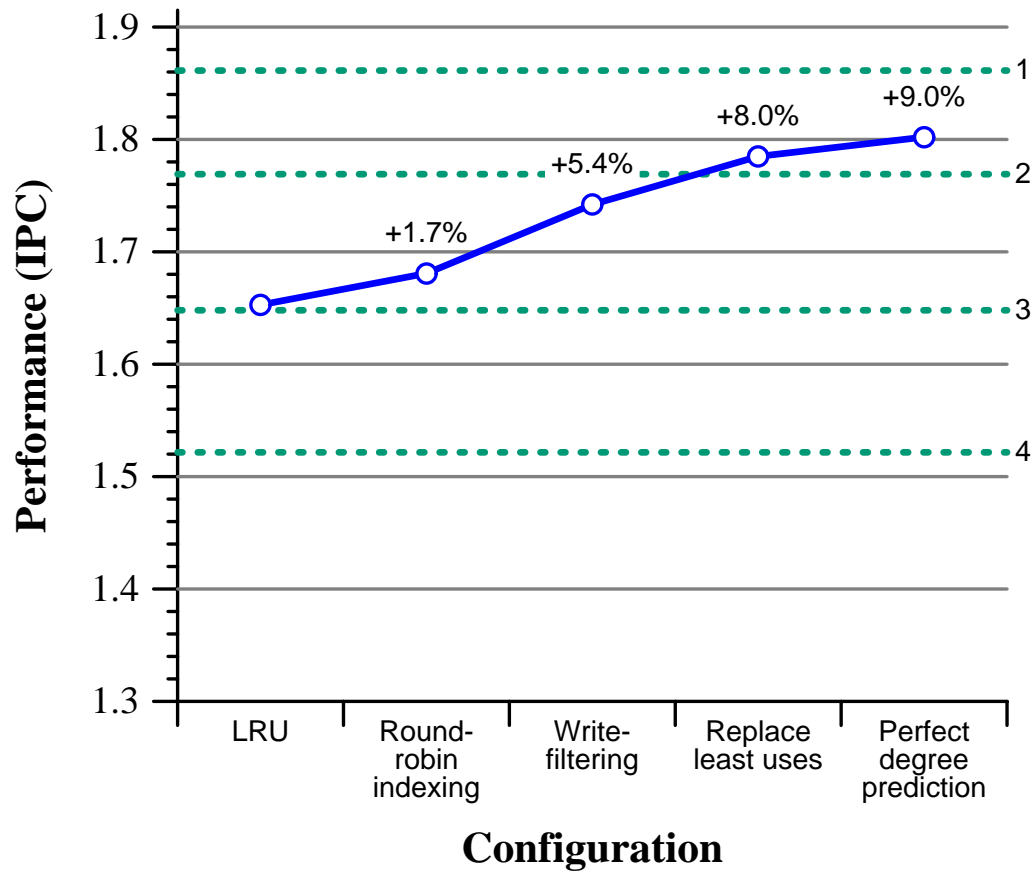
Backing file latency can be
lower than monolithic

- Few shared read ports
- 24-port \Rightarrow 8-port

Use-based register cache
tracks fully-bypassed
register file



Incremental Performance Breakdown



Outline

Motivation and Overview

Register Caching

Use-based Register Cache Management

Decoupled Indexing

Evaluation

Conclusion

- Future Work
- Questions

Future Work

Augmented heuristics to reduce misses from write-filtering?

- Account for mis-speculation
- Use additional information (static, operand type, etc.)

Deterministic scheduling latency

- Degree of use prediction + use counting
- False positive problem

Additional indexing schemes to reduce conflict misses

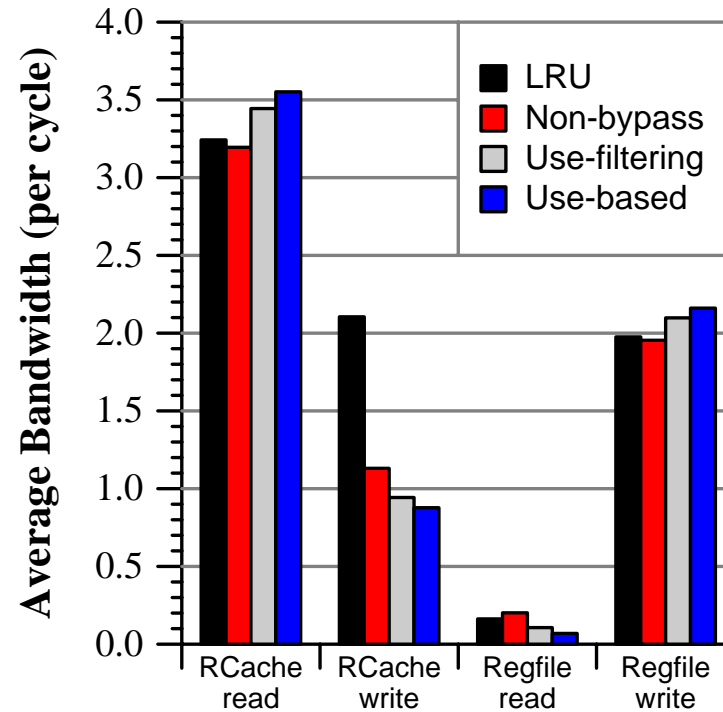
- Apply degree of use information
- Synchronization of front-end and register cache

Combine with previous work to reduce cache write ports

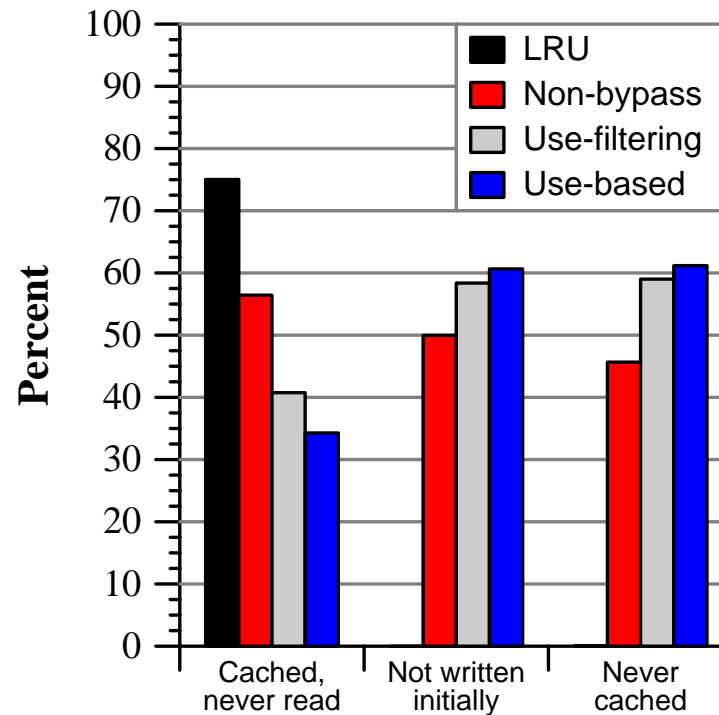
- Cache write bandwidth **<1 value per cycle**
- Requires arbitration, queueing, extra bypassing

Questions

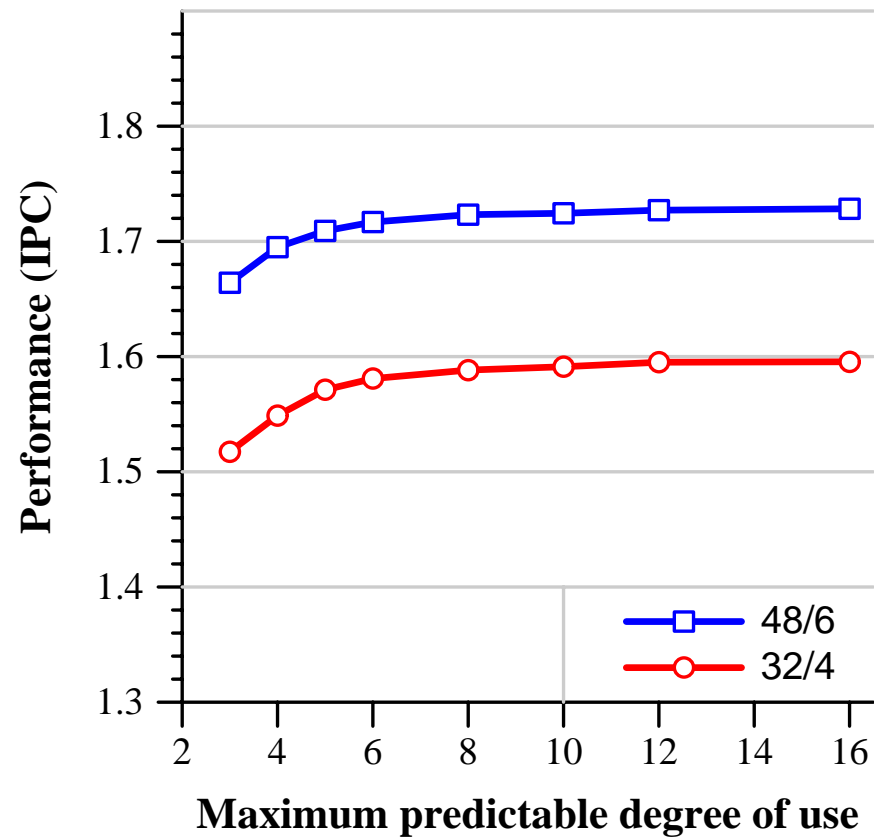
Access Bandwidth



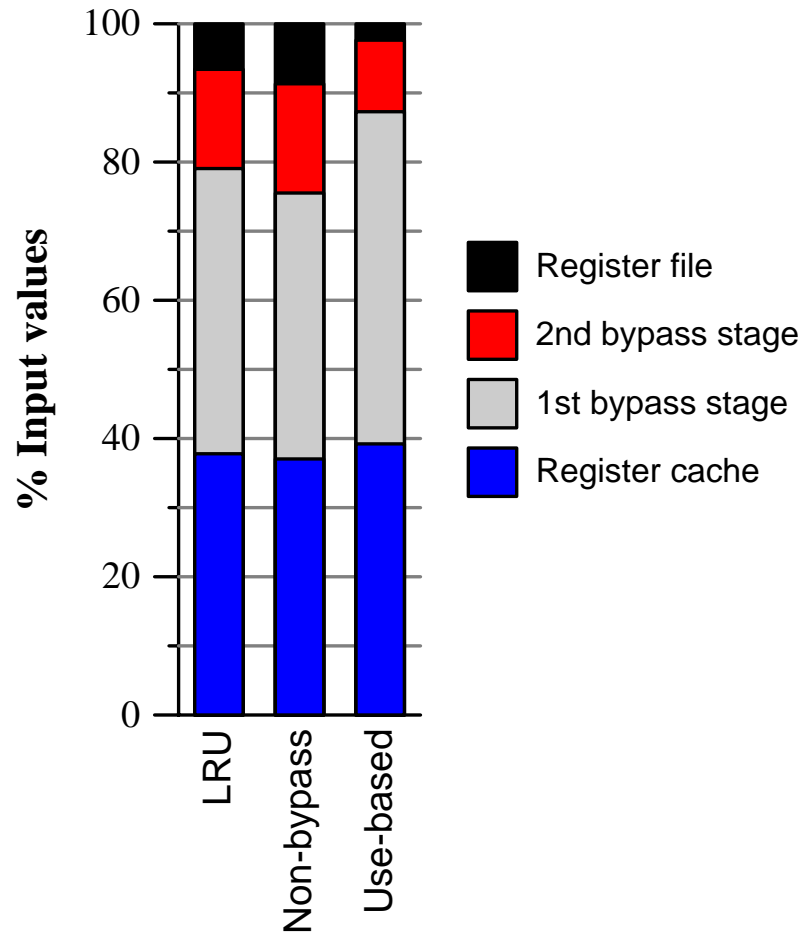
Write-filtering Effects



Maximum Degree of Use



Sources of Values for Execution



Comparison with Two-Level RegFile

