

# Optimizing Inter-Instruction Communication through Degree of Use Prediction

**J. Adam Butts**

**butts@cs.wisc.edu**

***University of Wisconsin–Madison***

***Thesis Proposal***

***September 24, 2002***

# Overview

---

**Value communication is a key component of execution**

**Value communication structures are **overly general****

- Simple communication patterns dominate, **but...**
- Must support **all** possible communication patterns for **all** values

**A new model for value communication**

- Choose suitable communication method on a per-value basis
- Based on **degree of use prediction**

**Three components to new model**

- **Determining the nature of the communication of each value**
- **Developing new, speculative methods for efficient communication**
- **Methods for mis-speculation recovery**

# Overview

---

## Communication optimizations

- Useless instruction elimination
- Register file management
- Dynamic operation chaining

## **Reduce utilization** of current communication structures

- Enable greater ILP at fixed cycle time
- Enable higher frequency with smaller structures

# Outline

---

## Overview

## Motivation

- Inter-instruction communication
- Degree of use
- Value communication in “real” programs

## Degree of Use Prediction

## Useless Instruction Elimination

## Register File Optimizations

## Dynamic Operation Chaining

## Summary

# Inter-Instruction Communication

---

## Register communication model

- Majority of value communication occurs through registers
- Instructions connected indirectly by register name
- Efficient representation, **but...**
- Implicit assumption of multiple consumers

## Significant effort is spent supporting this model

- Large, multi-ported register files
- Complicated bypass networks
- Broadcast tag match for instruction wakeup

**How to describe the actual needs of a value?**

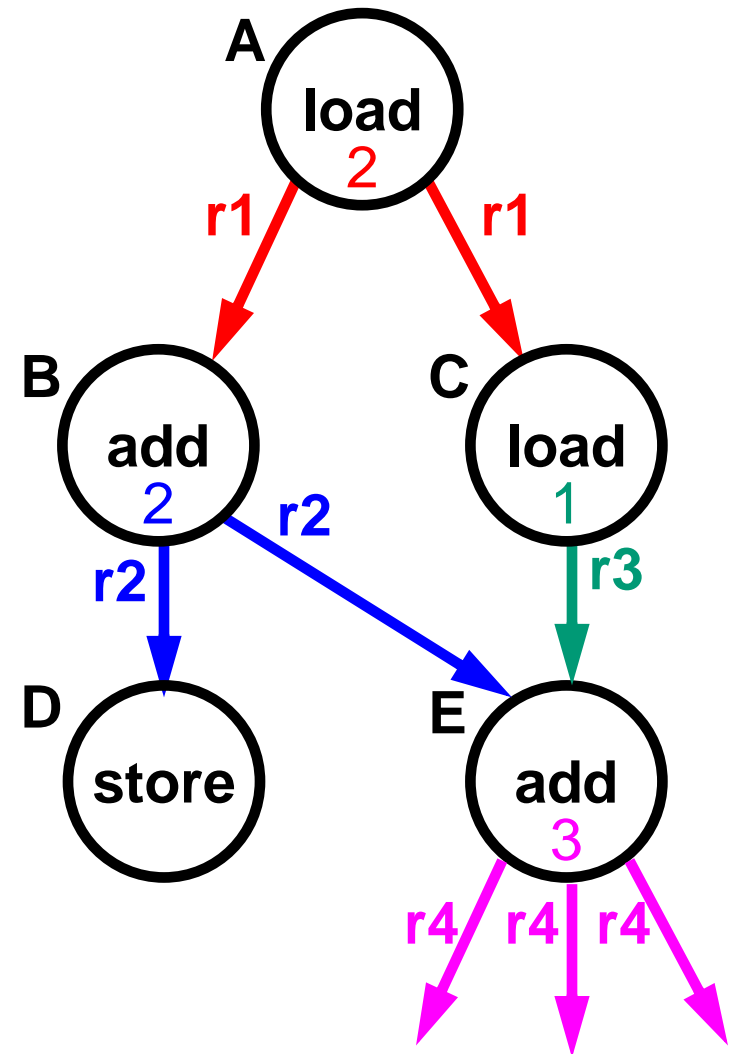
# Degree of Use

**Definition:** number of times a given dynamic value is used

- A direct indicator of the communication requirements of a value
- Function of both number of static consumers and dynamic control flow

**Focus on register values**

- **All** communicating instructions use at least one register
- Values not tracked through memory
  - Loads produce a new value
  - Stores produce no value
- Memory degree of use possible



# Characterizing Degree of Use

## Degree of use statistics

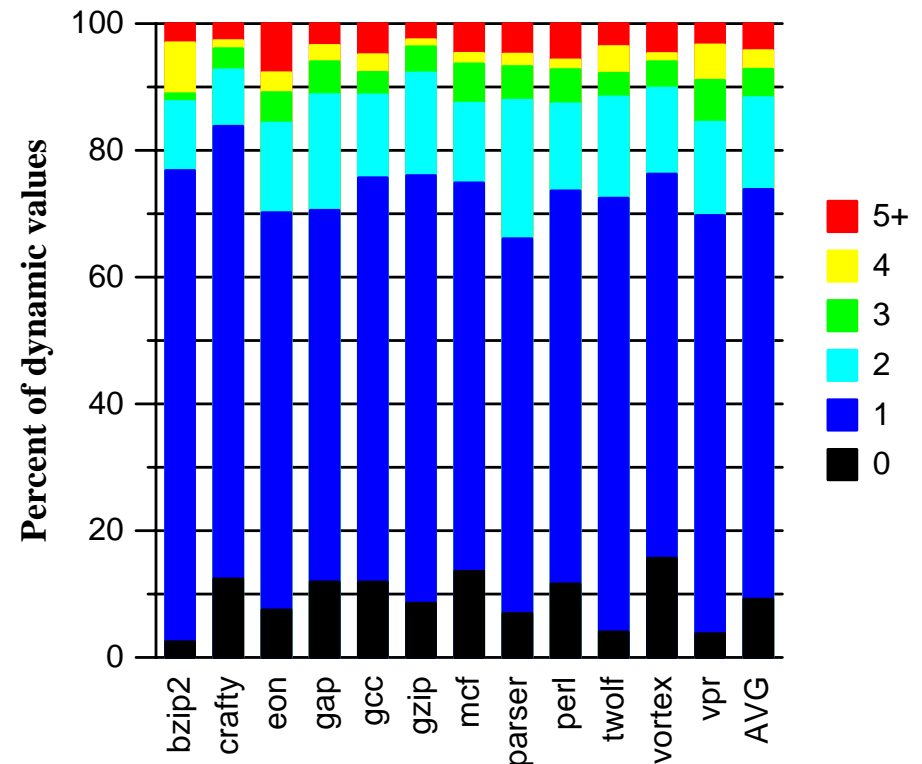
- Average: **1.66**
- Mode (most frequent): **1**
- Maximum: ~**330 M** (bzip2)

## FP benchmarks

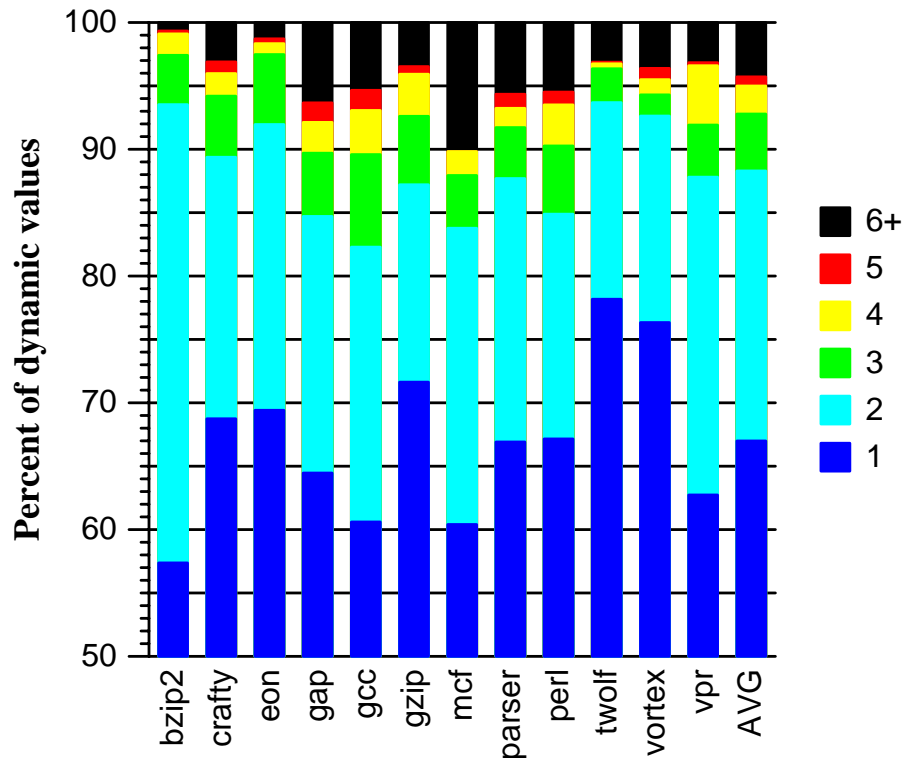
- Higher average: **1.83**
- Fewer 0, more 1, 2

## Independent of compiler

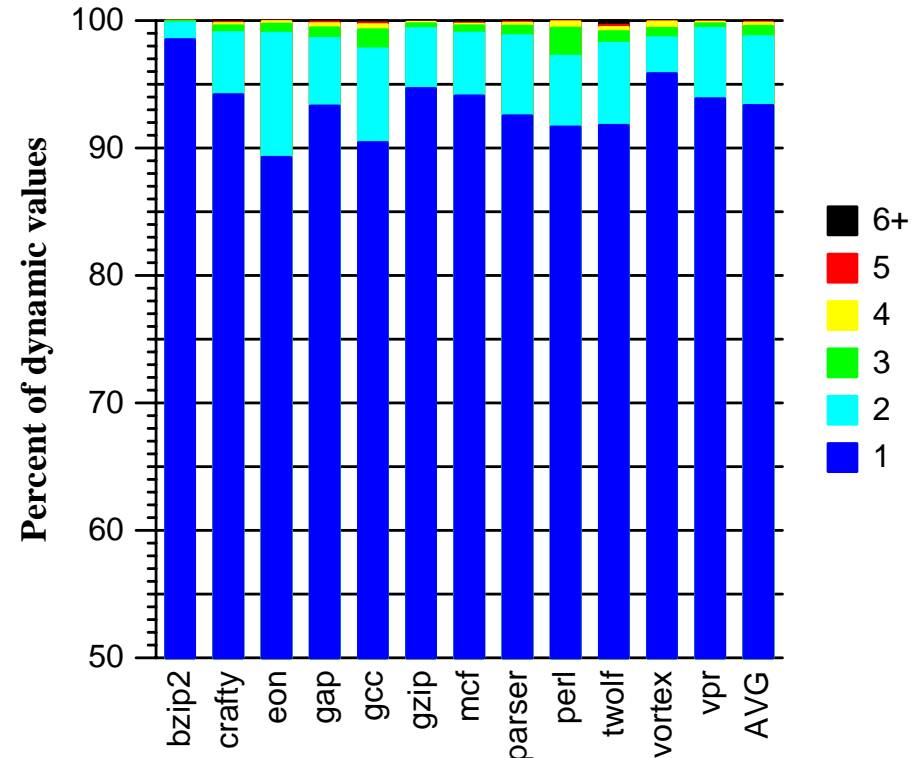
- gcc results similar
- Franklin and Sohi observed similar results on MIPS ISA and SPEC92 benchmarks



# Characterizing Degree of Use



**67%** of values from instructions generating **one degree of use**



**93%** of values have same degree of use as the **last** value from the same instruction

Instruction identity is **significant** factor in determining degree of use



# Outline

---

Overview

Motivation

## **Degree of Use Prediction**

- Predictor organization
- Forward control flow signatures
- Evaluation

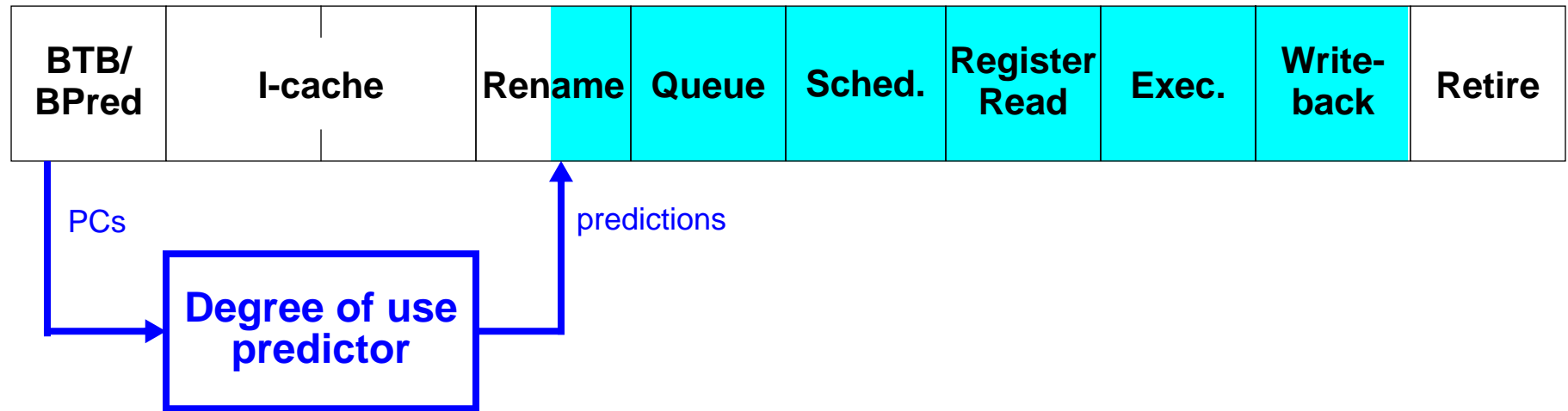
**Useless Instruction Elimination**

**Register File Optimizations**

**Dynamic Operation Chaining**

**Summary**

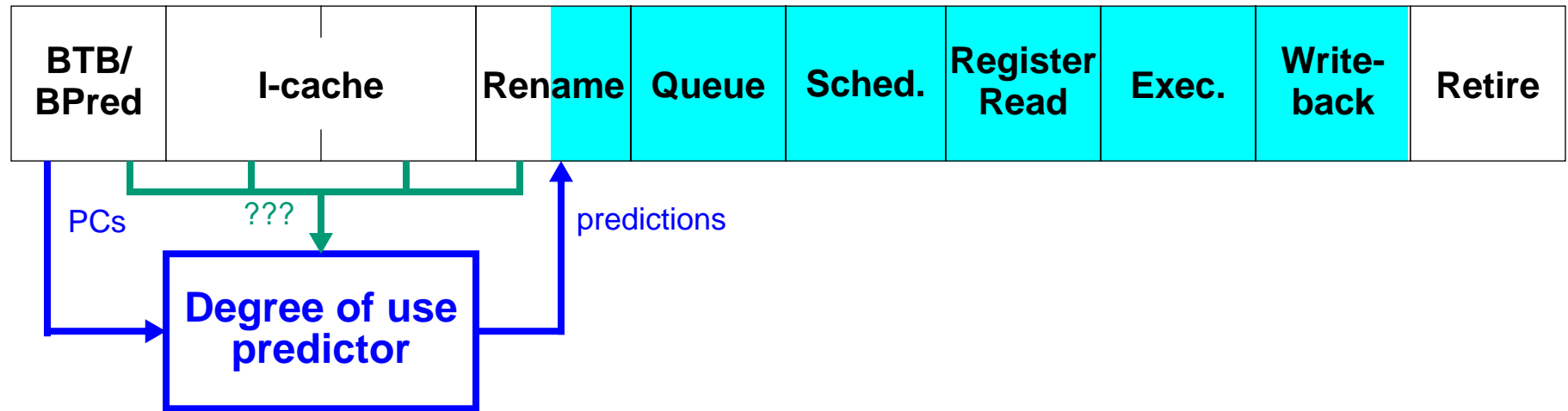
# Degree of Use Prediction



**Degree of use predictor provides timely, per-value knowledge**

- Indexed with instruction PC

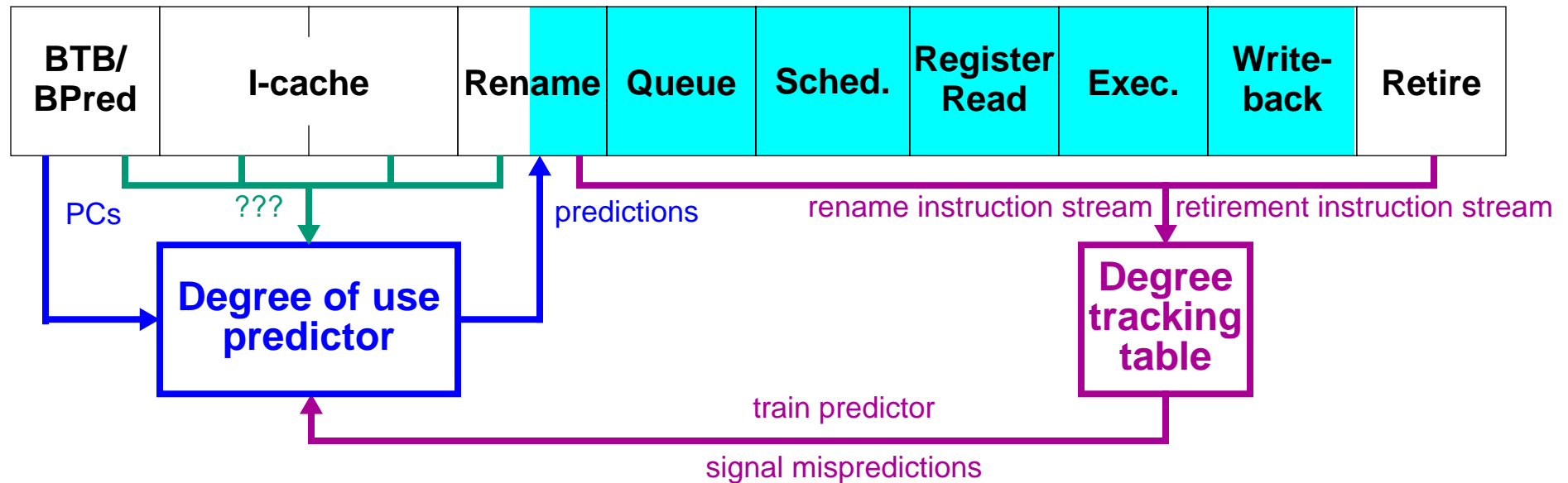
# Degree of Use Prediction



**Degree of use predictor provides timely, per-value knowledge**

- Indexed with instruction PC
- Use other pipeline information

# Degree of Use Prediction



## Degree of use predictor provides timely, per-value knowledge

- Indexed with instruction PC
- Use other pipeline information

## Observe instruction stream for training/misprediction detection

- Rename instruction stream - faster resolution, false paths
- Retire instruction stream - slower resolution, always correct path

# Forward Control Flow Signatures

How to differentiate multiple possible degrees of use?

**Future** control flow uniquely determines degree of use

- All uses occur after value is generated
- Observed uses depend solely on which path is taken

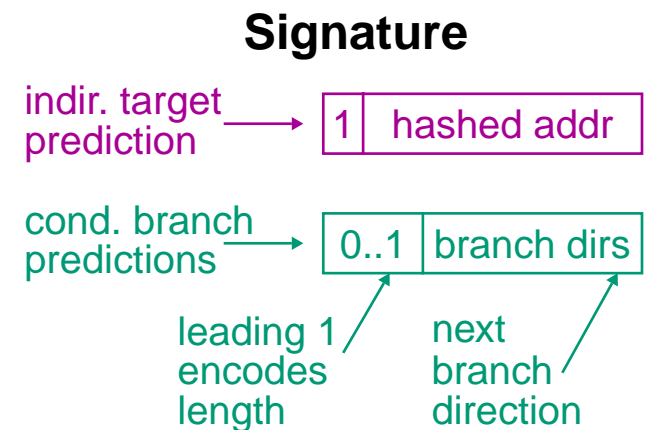
**Predicted future control flow is available**

- Degree predictor resides in middle of pipeline
- Future control predictions are available in earlier pipeline stages

**Not quite perfect**

- Predictions may not be accurate
- Pipeline depth limits number of predictions available

**Signature encodes future control flow from conditional or indirect branches**



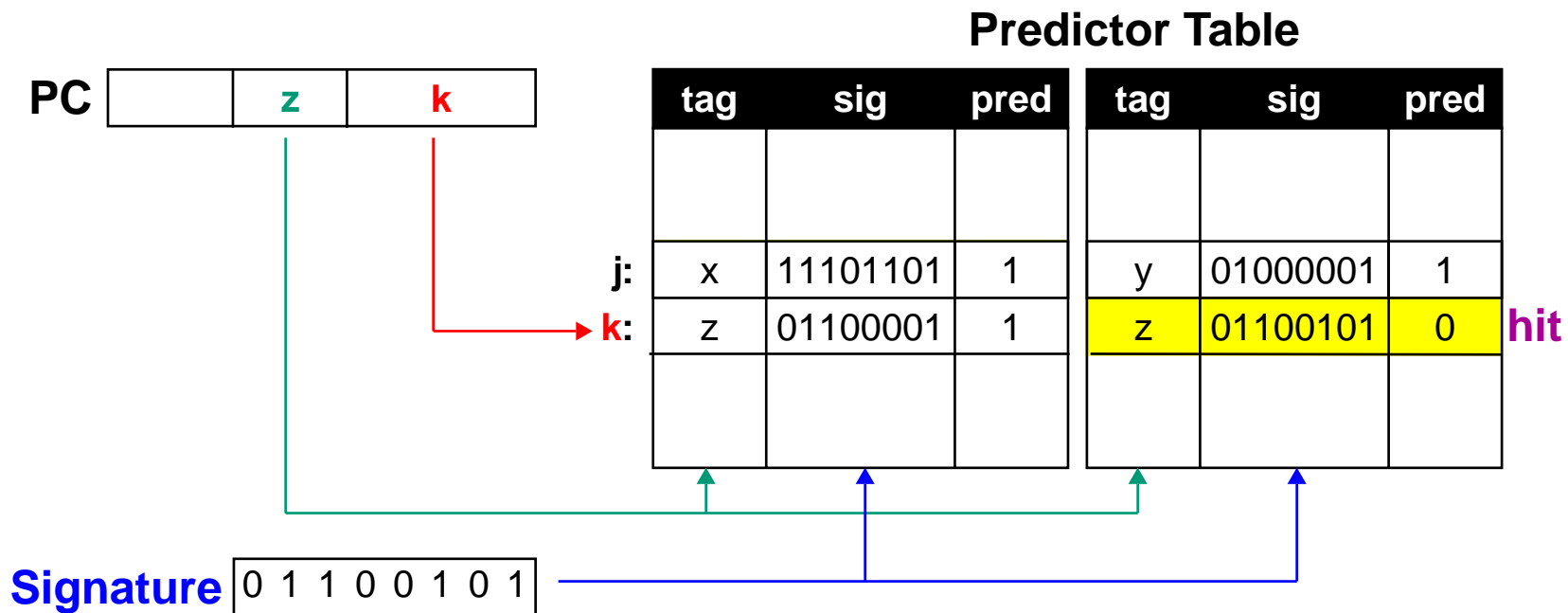
# Predictor Microarchitecture

## Associate predictions with instruction identity

- Index with **low-order PC bits**, tag entries with **higher-order bits**

## Support **multiple predictions** per static instruction

- Use a set-associative predictor organization
- Use **control flow signature** as part of tag



# Predictor Performance

## High accuracy

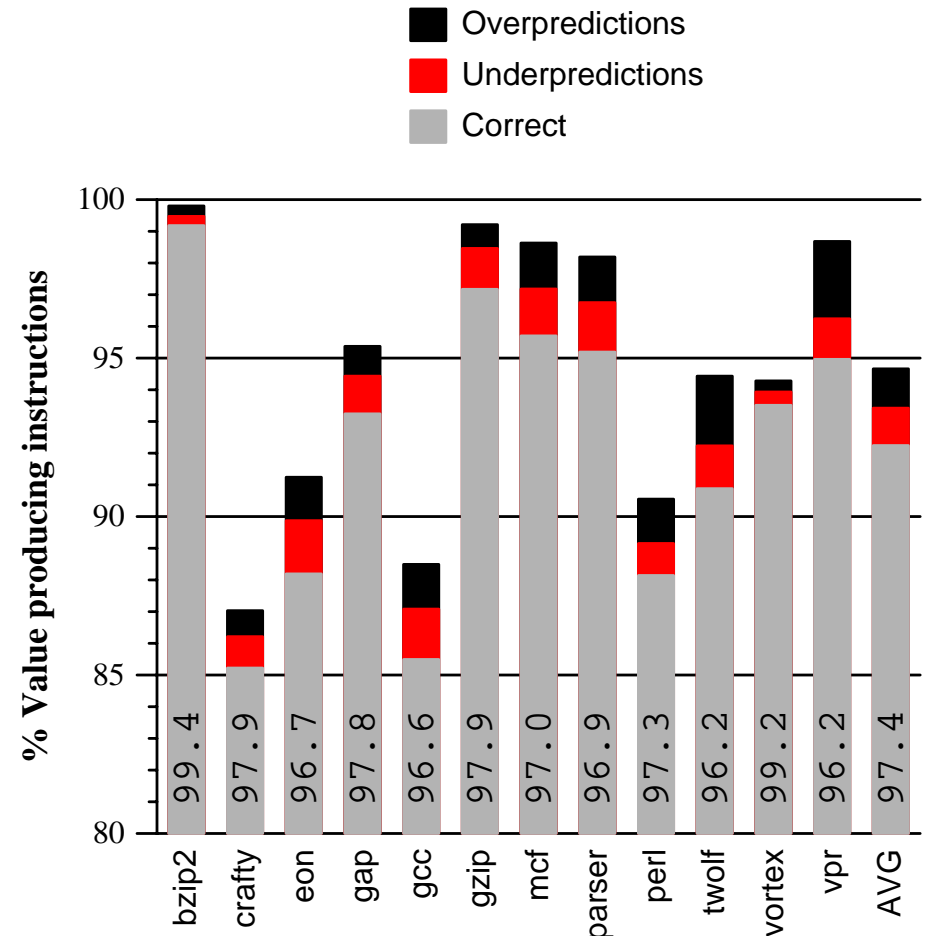
- **< 3%** misprediction rate
- Worst case < 4%
- Includes overpredictions and **underpredictions**

## High coverage

- **92%** of values receive a **correct** prediction
- Coverage depends on code working set
- Strongly dependent on predictor size, organization

## Low overhead

- 8.5 KB, relaxed timing



# Predictor Performance by Degree

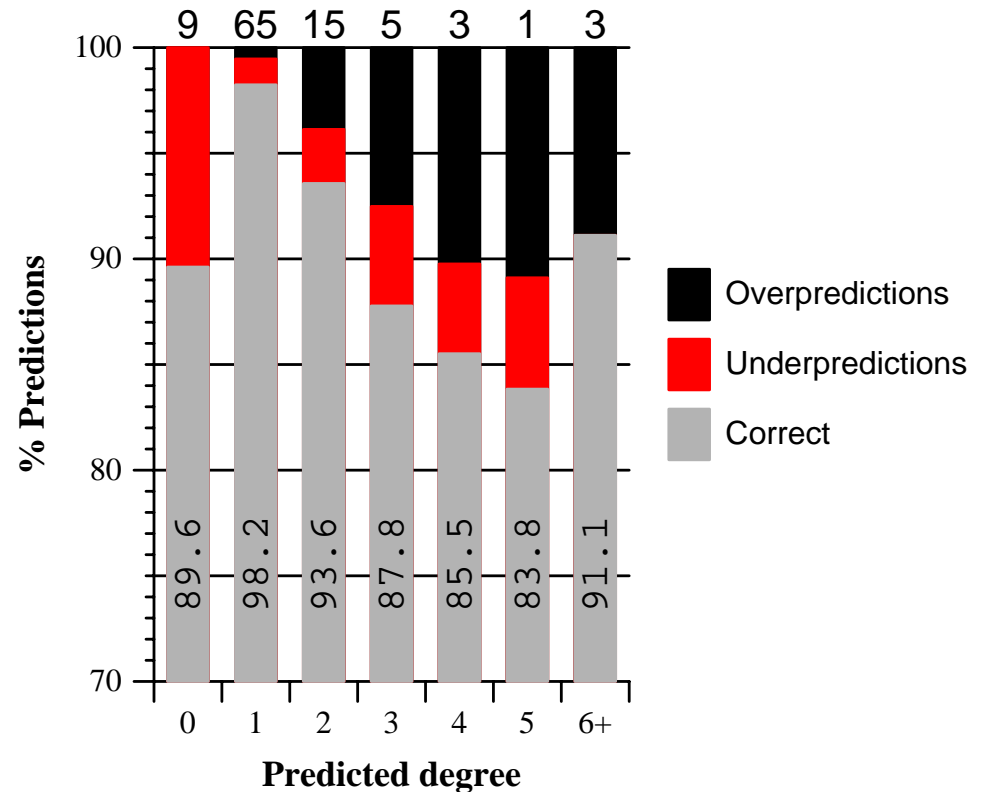
Prediction frequency reflects  
degree of use distribution

Accuracy depends on degree

- Degree of use 1 predictions
  - Most accurate
  - Most mispredictions also
- Accuracy diminishes with increasing predicted degree

Under- vs. overpredictions

- Predicted degree <2:  
probably underprediction
- >2: probably overprediction





# Outline

---

Overview

Motivation

Degree of Use Prediction

## **Useless Instruction Elimination**

- Sources of useless instructions
- Mechanism of elimination
- Evaluation

**Register File Optimizations**

**Dynamic Operation Chaining**


**Summary**

# Useless Instructions

**Useless instructions** generate values with degree of use zero

Search( ):

```
...
addl    a2, 0x1, a2
cmovge  t3, t3, t5
sra     t5, 0x2, t5
xor     t5, 0x1, t5
bne     t5, X
ldbu    t7, 0(t6)
cmovlt  s0, a3, a0
stl     a2, -19686(gp)
bis     t7, 0x8, t7
stb     t7, 0(t6)
X: bis   zero, s2, a2
...
```



sv\_upgrade( ):

```
...
bis     zero, 0x1, v0
...
ret     zero, (ra), 1
```

sv\_grow( ):

```
...
bsr     ra, sv_upgrade
ldq     v0, 0(s0)
...
```

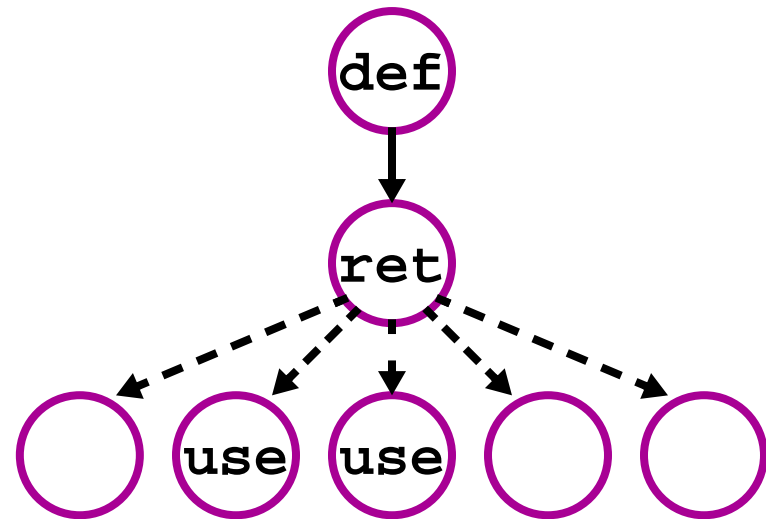
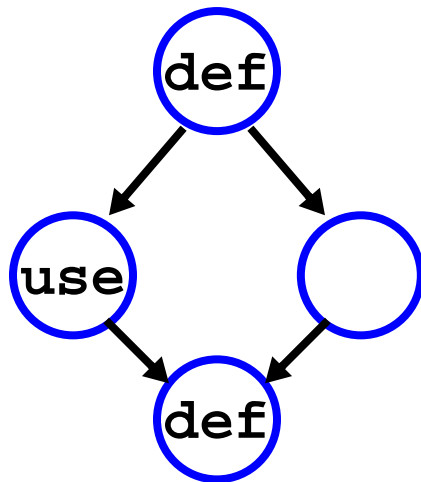
av\_fake( ):

```
...
bsr     ra, sv_upgrade
addl    s0, 0x1, v0
...
```

# Useless Instruction Sources and Costs

## Sources of useless instructions

- Partially dead instructions created by **compiler optimizations**
- Dead instructions requiring **interprocedural analysis** to detect



## Costs of useless instructions

- Physical register handling
- Instruction window slots
- ALU occupancy
- Cache bandwidth
- Register file bandwidth

# Eliminating Useless Instructions

PC<sub>i</sub>: ldq r5, 8(sp)  
PC<sub>j</sub>: addl r1, r5, r6  
PC<sub>k</sub>: stq r2, 0(r8)  
...  
PC<sub>x</sub>: bis r0, r3, r6

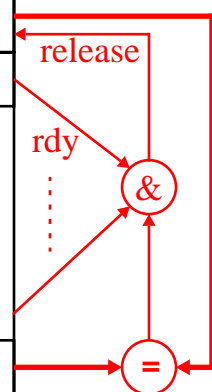
Predicted Useless Table

	V	instruction	ROB
4:	1	addl r1, r5, r6	7

scheduler

Reorder Buffer

	instruction	PUT
6:	ldq r5, 8(sp)	—
7:	addl r1, r5, r6	4
8:	stq r2, 0(r8)	—
21:	bis r0,r3,r6	4



Add **predicted useless table** (PUT), pointers in ROB

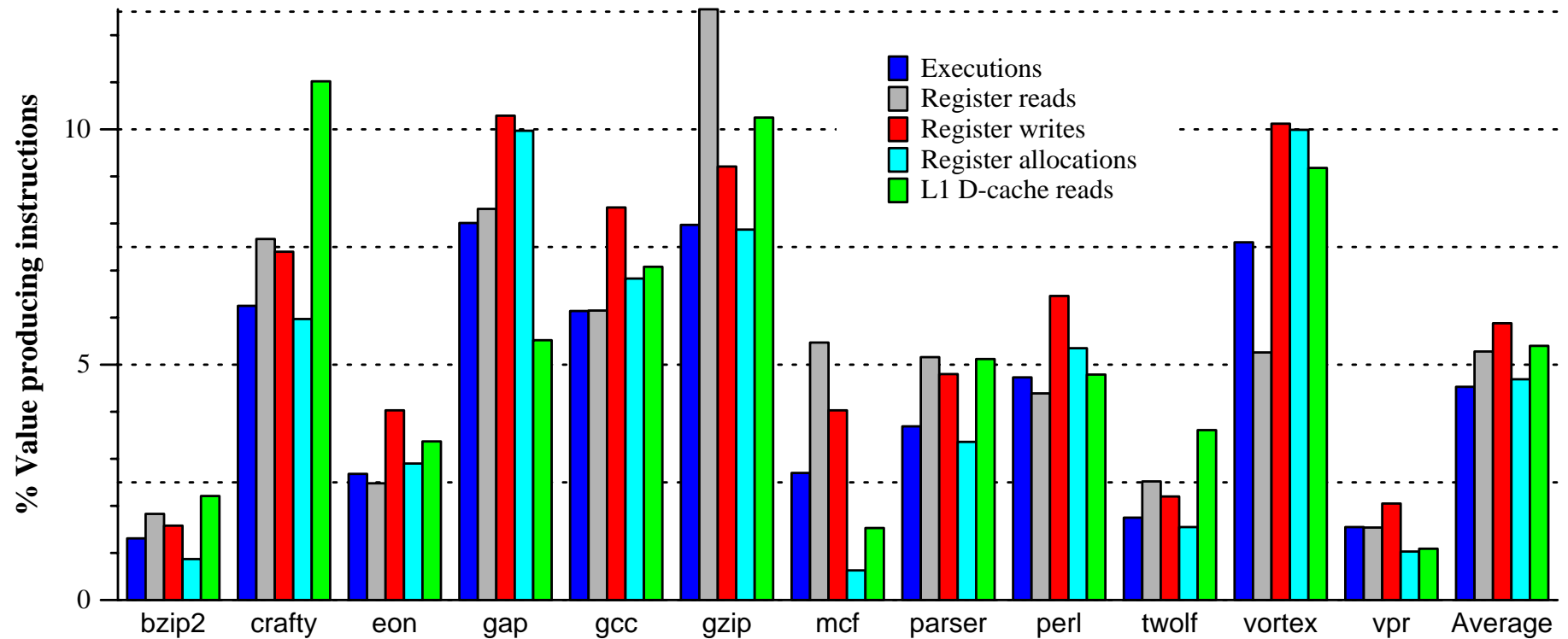
- Tracks predictions, stores partially-renamed instructions for recovery

Modify **retirement logic**

- Retire useless instruction only when **verifying instruction** and all intervening instructions are ready to retire

Allow **scheduling from PUT** for recovery

# Results: Resource Utilization



**Global ~5% reduction in utilization of many critical resources**

- Several benchmarks see **>10%** reductions

**Relative reductions depend on instruction mix**

- ALU operations vs. loads
- 0-, 1-, and 2-input instructions

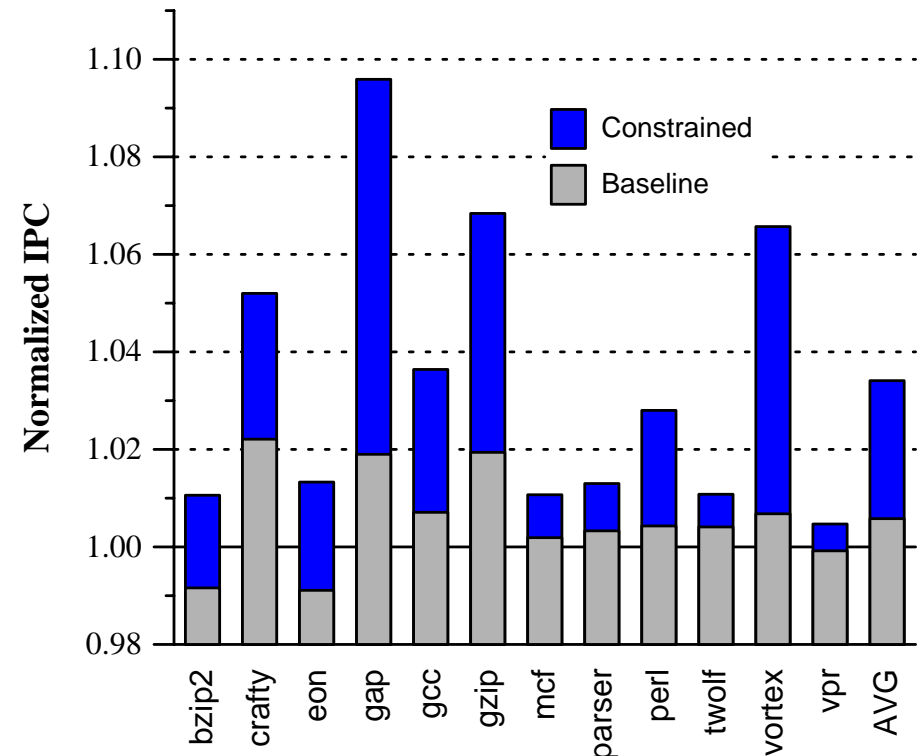
# Results: Performance

## Performance impact depends on resource contention

- 0.6% speedup on resource-rich **baseline** model
- 3.4% speedup on resource-**constrained** model
- SMT processors?

## 70% of potential speedup

- **NOT** mis-speculations
- Instructions not eliminated
  - Unidentified (predictor)
  - Unverifiable (ROB size)
- Retirement holdup



# Outline

---

Overview

Motivation

Degree of Use Prediction

Useless Instruction Elimination

## Register File Optimizations

- Two-level register file management
- Minimizing register lifetime
- Value recovery

Dynamic Operation Chaining

Summary

# Avoiding Register File Writes

---

Use alternate communication mechanism—the **bypass network**

- Already exists
- Designed for communication with a few consumers
- *Avoid communicating through the register file what has already been communicated through the bypass network*
- Degree of use indicates what communication is required

Bypass network does **not** retain values

- If register file is not written, need a means to **recover values**

**Two applications** to avoid writing a fast, full-sized register file

- **Bypass counting** to manage a two-level register file
- **Late allocation and early release** of physical registers



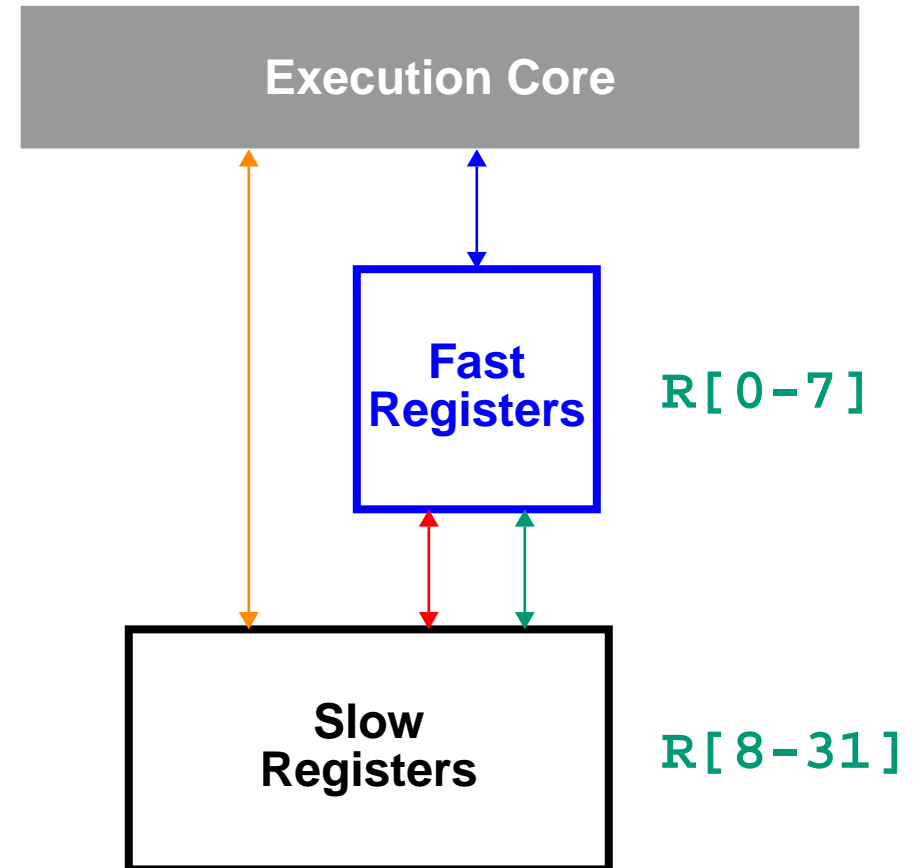
# Background: Two-Level Register Files

## 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



**Fast registers: speculative register file**

**Slow registers: value recovery**

# Application 1: Bypass Counting

---

Based on *non-bypass* scheme proposed by Cruz et al.

- All values are written into lower-level file
- Also write value to register cache if result was **not bypassed**
- **Values with multiple consumers may be bypassed to some**
  - Def-first use distance largely independent of degree of use
  - Subsequent consumers experience higher latency

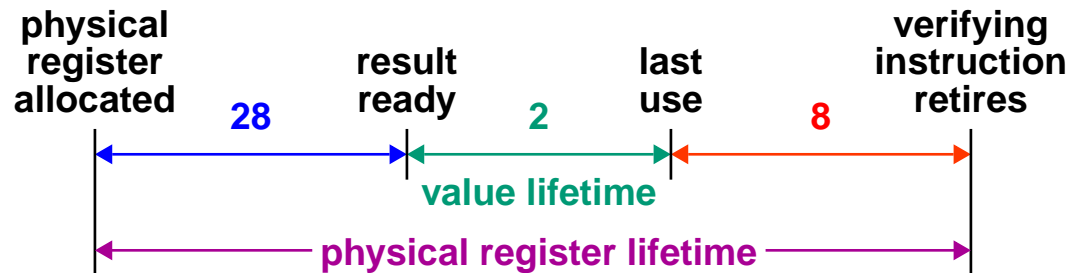
## Bypass counting

- Leverage degree of use knowledge
- Write to cache if **number of bypasses < predicted degree of use**

# Background: Register vs. Value Lifetimes

**Physical register lifetimes** significantly exceed **value lifetimes**

- Leads to poor utilization of physical registers
- More physical registers required  $\Rightarrow$  big and slow register file



**Empty time:** for dependency tracking

- Physical register allocated at decode time (dependency tracking)
- Value written after execution executes

**Dead time:** for recovery purposes

- Value dead after last use
- Register freed when verifying instruction retires

# Application 2: Late Alloc. & Early Release

---

*Virtual-physical registers* (González et al.) **eliminate empty time**

- Separates **dependence tracking** from **value storage**
- **Virtual-physical register** assigned at decode time
- **Real physical register** assigned when result is ready
- Maintain a mapping between the two

**Combine virtual-physical registers with bypass counting**

- Dependency-tracking is preserved
- Allocate physical register after execute **EXCEPT**
- Do not allocate register if bypass count  $\geq$  **predicted degree of use**

**Can also attack **dead time****

- Free physical register when total reads  $\geq$  **predicted degree of use**

# Value Recovery

---

**Speculative nature of scheme requires **value recovery** due to...**

- A degree of use underprediction
- A control mis-speculation requiring re-execution

## **Three recovery strategies**

- **Store values somewhere else**
  - All values are stored
  - Recovery structure can be optimized (distributed, write-only storage)
  - Two-level register file
- **Recreate values through re-execution**
  - Must ensure availability of inputs and instructions
  - Useless instruction elimination
- **Checkpoint recovery**
  - Periodic snapshot of state + limited re-execution on recovery

# Outline

---

Overview

Motivation

Degree of Use Prediction

Useless Instruction Elimination

Register File Optimizations

**Dynamic Operation Chaining**

- Instruction chains
- Back end chain execution model
- Dependence chain optimizations

**Summary**

# Avoiding Value Movement

---

## Scheduling window too general

- Producer does not know **location** of consumer(s)
- Expensive tag broadcast, yet most instructions wake one consumer

## Bring consumer instruction to the value

- Direct producer of **single-use value** to **special execution unit**
- Direct consumer to same location
- Avoids value communication and scheduling

## Potential benefits

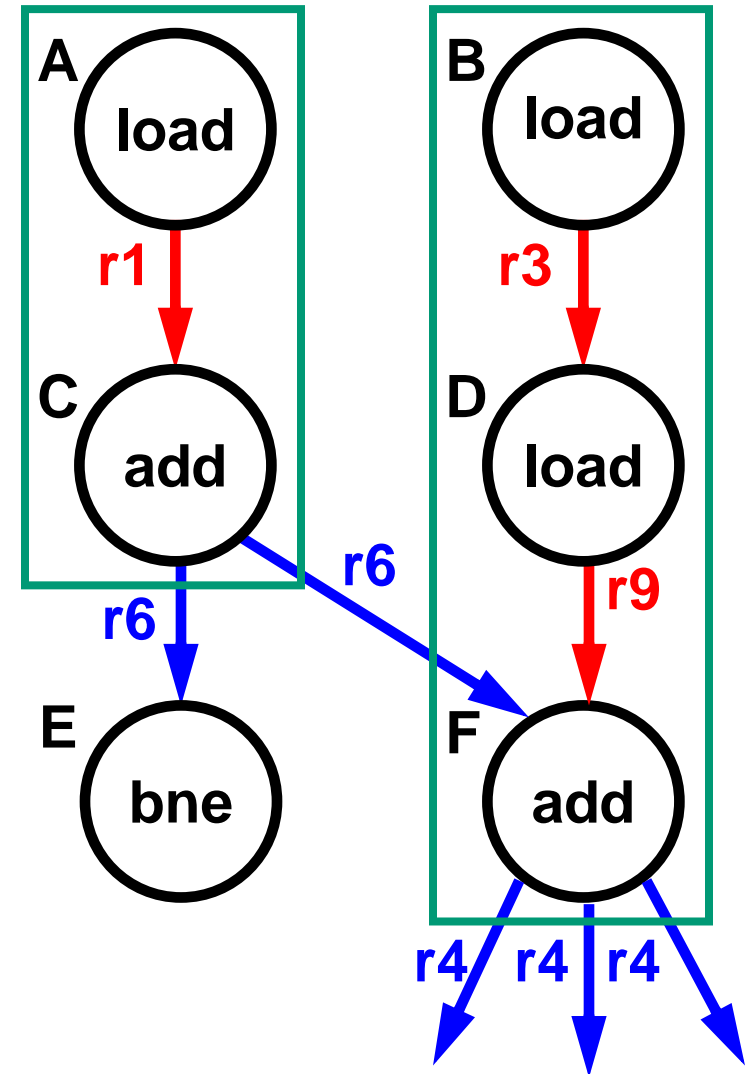
- Consumers of single-use values **do not occupy window**
  - No scheduling → no tag broadcast
  - Smaller window **or** more lookahead
- **More execution bandwidth** with same communication structures
  - Fewer read ports, write ports, bypass connections for special ALUs

# Instruction Chains

## Definitions

- **Instruction chain**: sequence of data-dependent dynamic instructions connected by degree of use 1 values
- **Chain terminating instruction**: Instruction with input from chain, but:
  - output predicted degree of use > 1
  - output only into another chain
  - no output

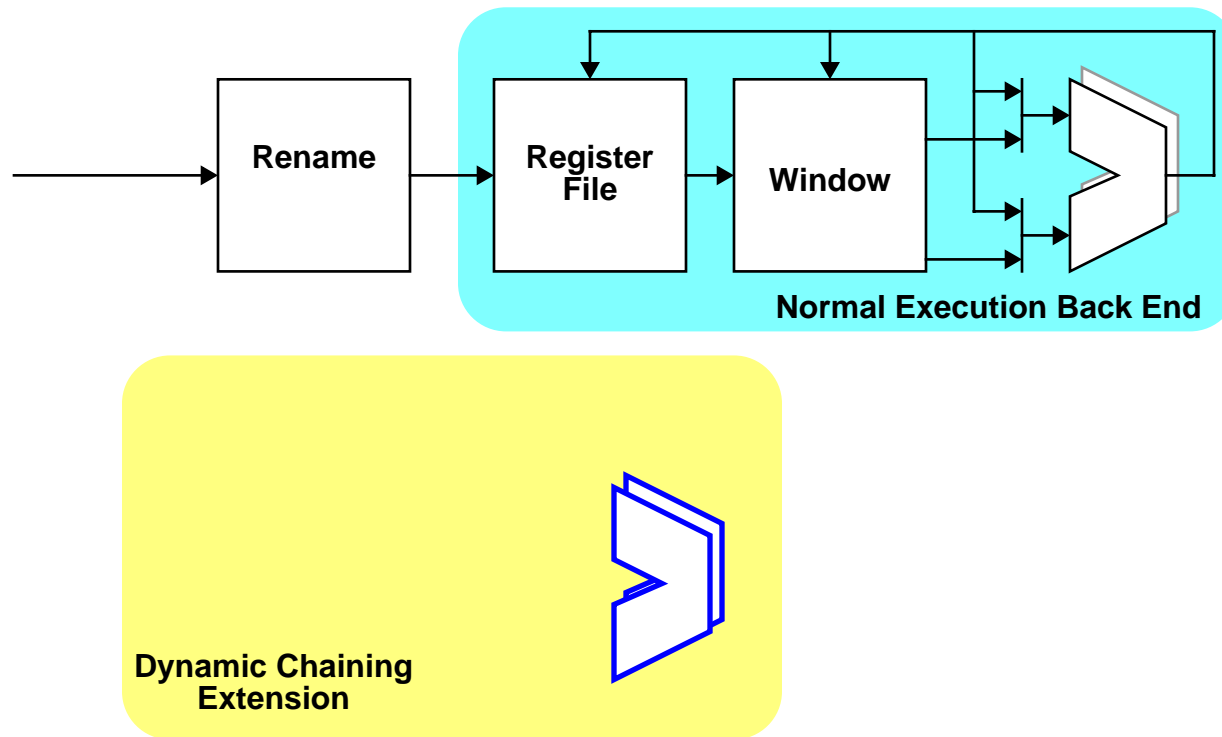
Identify chains dynamically with a degree of use predictor and execute them efficiently





# Efficient Chain Execution

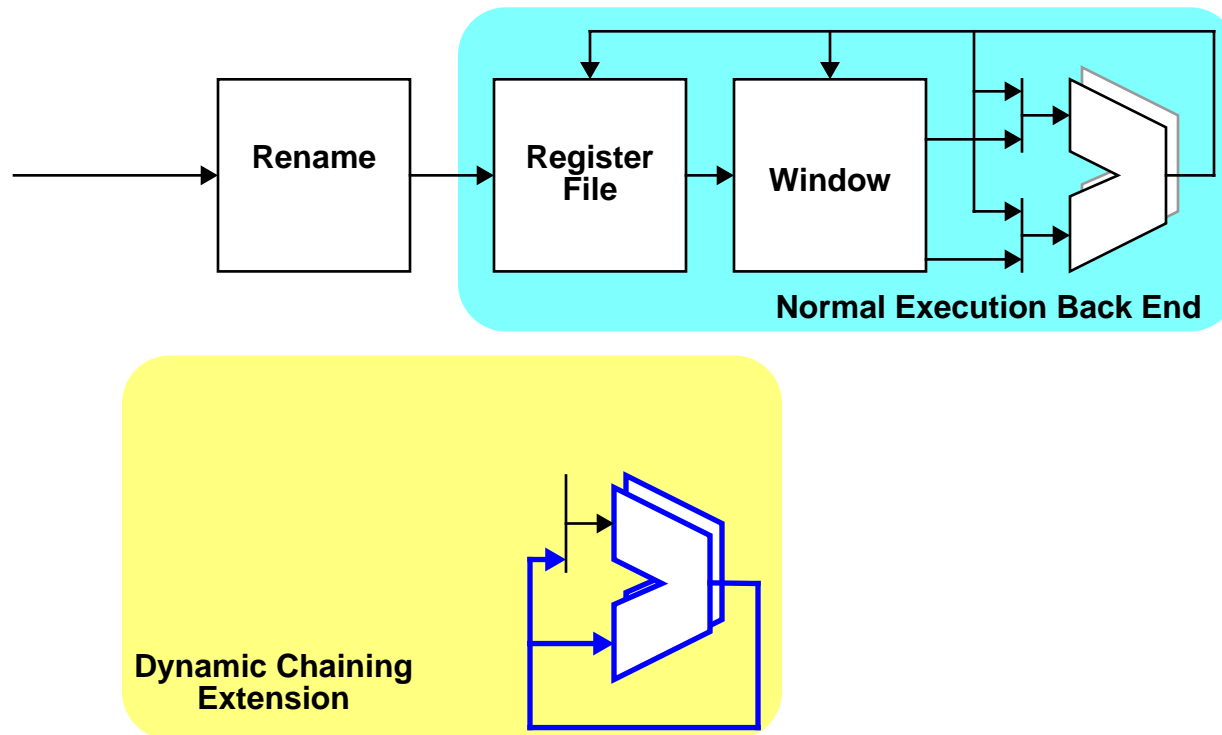
Execute chains on **chaining ALUs**



# Efficient Chain Execution

## Execute chains on **chaining ALUs**

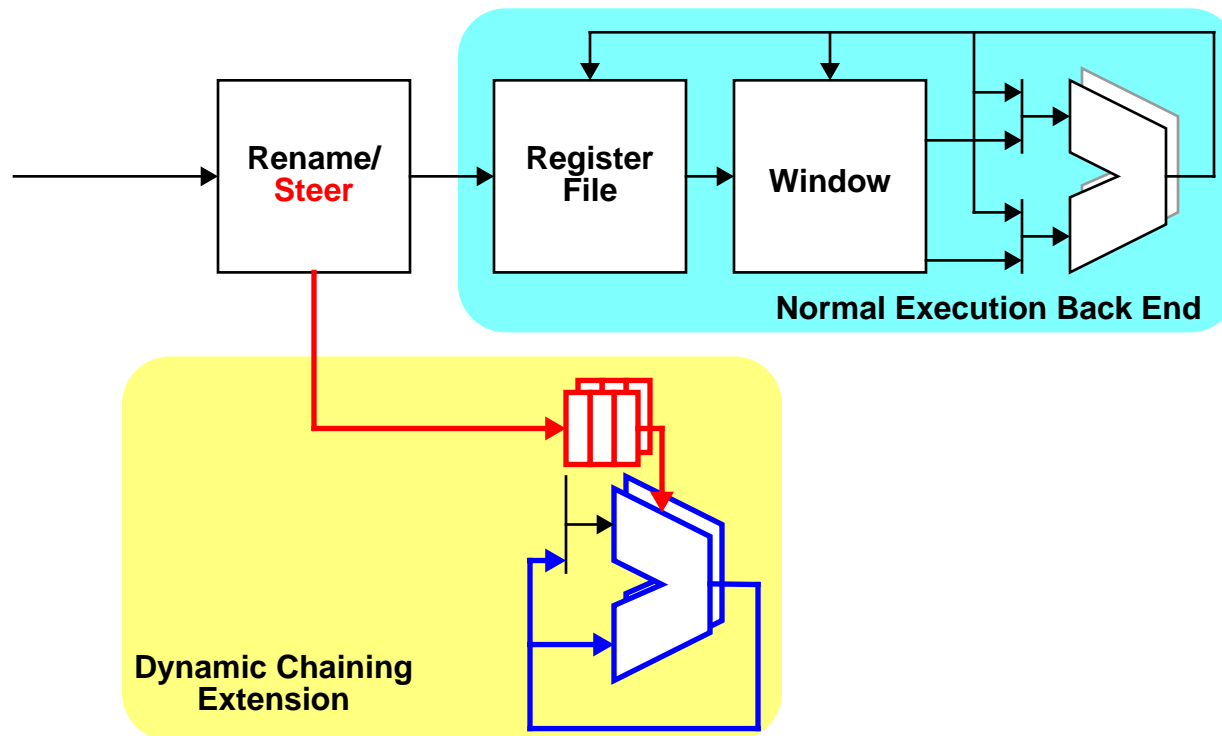
- Result **bypasses only** to same ALU for next instruction in chain



# Efficient Chain Execution

## Execute chains on **chaining ALUs**

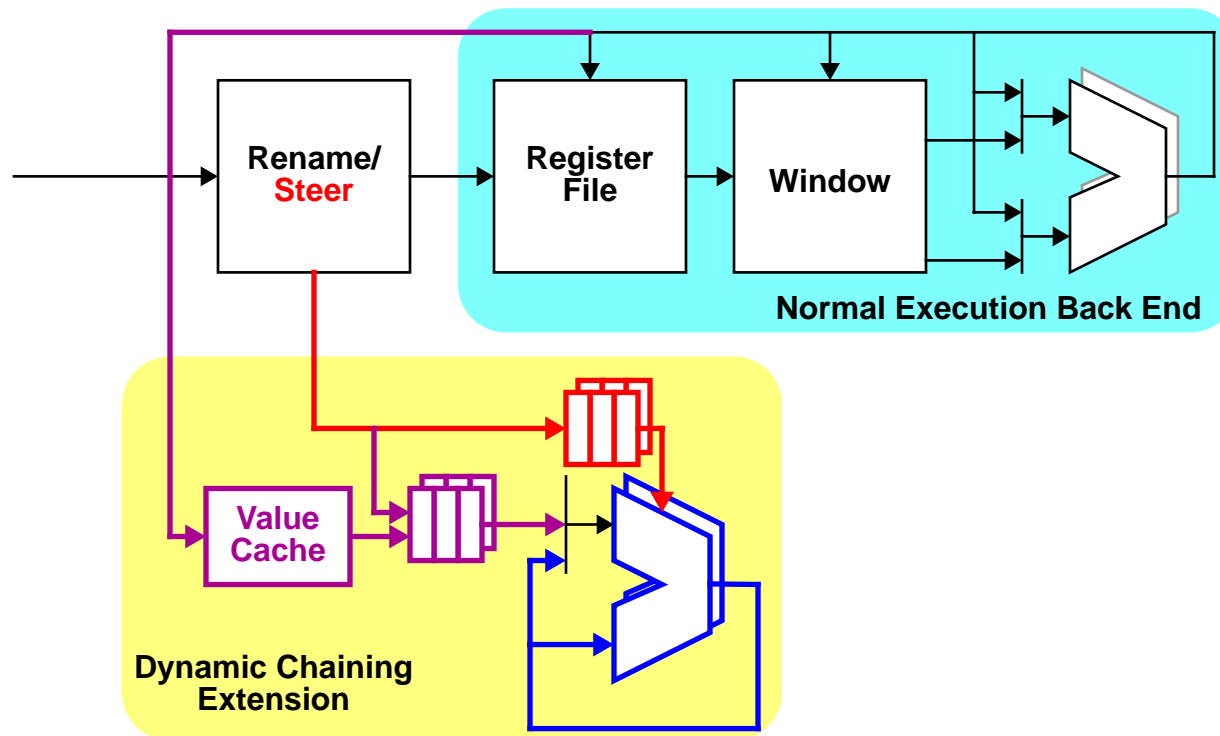
- Result **bypasses only** to same ALU for next instruction in chain
- **Steer** instructions into queues based on input operands



# Efficient Chain Execution

## Execute chains on **chaining ALUs**

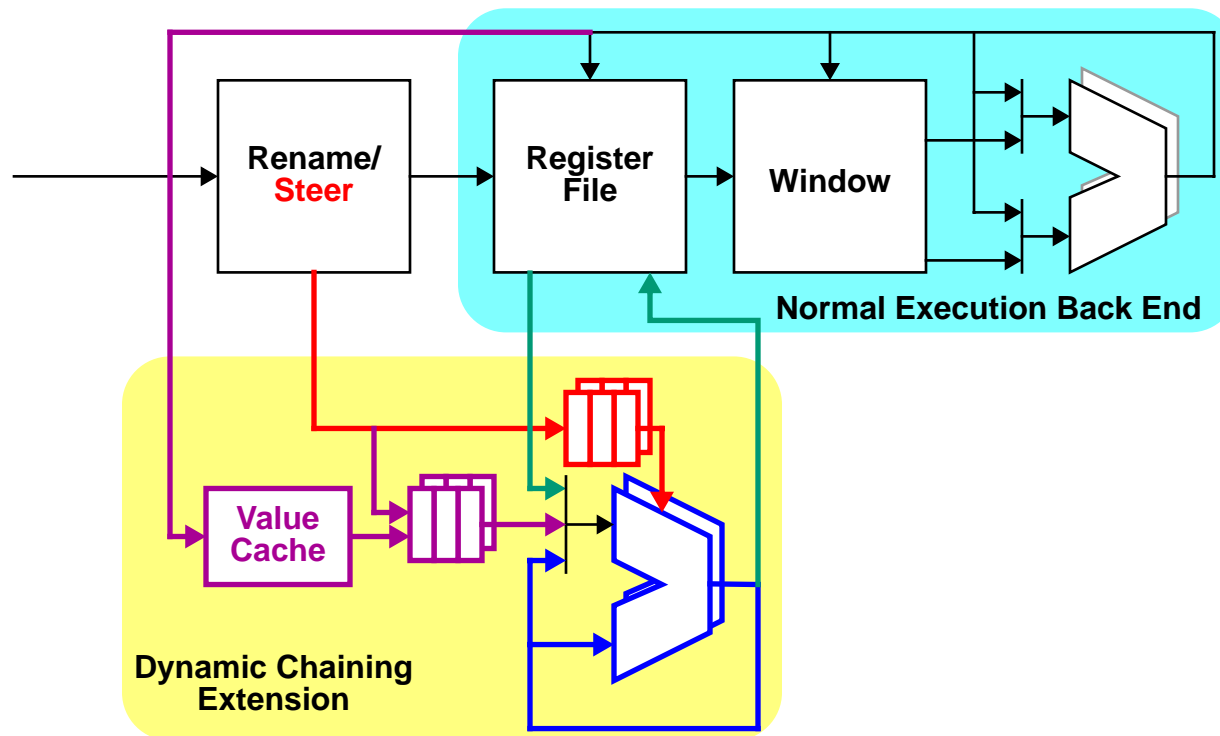
- Result **bypasses only** to same ALU for next instruction in chain
- **Steer instructions into queues based on input operands**
- Value queues, high-use value cache reduce read bandwidth



# Efficient Chain Execution

## Execute chains on **chaining ALUs**

- Result **bypasses only** to same ALU for next instruction in chain
- **Steer instructions into queues based on input operands**
- Value queues, high-use value cache reduce read bandwidth
- Chaining ALUs share register file read, write ports



# Dependence Chain Optimizations

---

## Execute pairs of **dependent operations** in the same cycle

- Eligible operations depend on latency, cycle time constraints
  - Dependent logical operations
  - 3-input additions
- Facilitated by **proximity in chaining queues**
  - Modify chaining queue to detect this situation
  - Issue dependent operations simultaneously
- Requires **modified ALU**
  - Necessary modifications have been described previously
  - May want to limit number of register inputs

## **Eliminate entire chain** when terminated by useless instruction

- Additional benefit from eliminating **transitively useless instructions**

# Outline

---

Overview

Motivation

Degree of Use Prediction

Useless Instruction Elimination

Register File Optimizations

Dynamic Operation Chaining

## Summary

- Related Work
- Status and Schedule
- Contributions

# Related Work

---

## Degree of use

- Register traffic analysis - Franklin and Sohi
- Analytical-statistical model - Eeckhout and Bosschere

## Useless instruction elimination

- Partial dead-code elimination - Knoop et al.
- Exploiting dead value information - Martin, Roth, and Fischer
- Ineffectual instruction sequences - Rotenberg

## Two-level register files

- Hierarchical registers for scientific computing - Swensen and Patt
- Software managed hierarchical RF for VLIW - Zalamea et al.
- Caching processor general registers - Yung and Wilhelm, Cruz et al.
- Value aging buffer - Hu and Martonosi
- Copy to backup when no live uses - Balasubramonian et al.



# Related Work

---

## Register lifetime optimizations

- Virtual-physical registers - González et al.
- Exploiting short-lived variables - Lozano and Gao
- A scalable RF architecture... - Wallace and Bagherzadeh

## Dependence-based clustering

- Complexity-effective SS processors - Palacharla, Jouppi, and Smith
- Instruction-level distributed processing - Kim and Smith

## Collapsing dependent instructions

- Interlock collapsing ALUs - Malik, Eickemeyer, and Vassiliadis
- Perf. potential of collapsing - Sazeides, Vassiliadis, and Smith
- High perf. 3-1 interlock collapsing ALUs - Philips and Vassiliadis
- Instruction pre-processing in trace procs - Jacobson and Smith

# Status and Schedule

---

## Complete (modulo small tasks)

- Motivation, characterization of degree of use (MICRO-02)
- Degree of use prediction (MICRO-02)
- Useless instruction elimination (MICRO-02)

## TODO

- Dynamic operation chaining (**Fall 2002**)
- Register file optimizations (**Spring 2003**)
- Write dissertation (**Summer-Fall 2003**)

## Planned graduation in **fall of 2003**

# Summary of Contributions

---

## Alternative model for inter-instruction communication

- Per-value determination of communication patterns
- Value communication optimizations
- Recovery for speculative optimizations

## Degree of use prediction

- Provides intuitive, direct knowledge of communication requirements
- Predictable with high coverage, accuracy

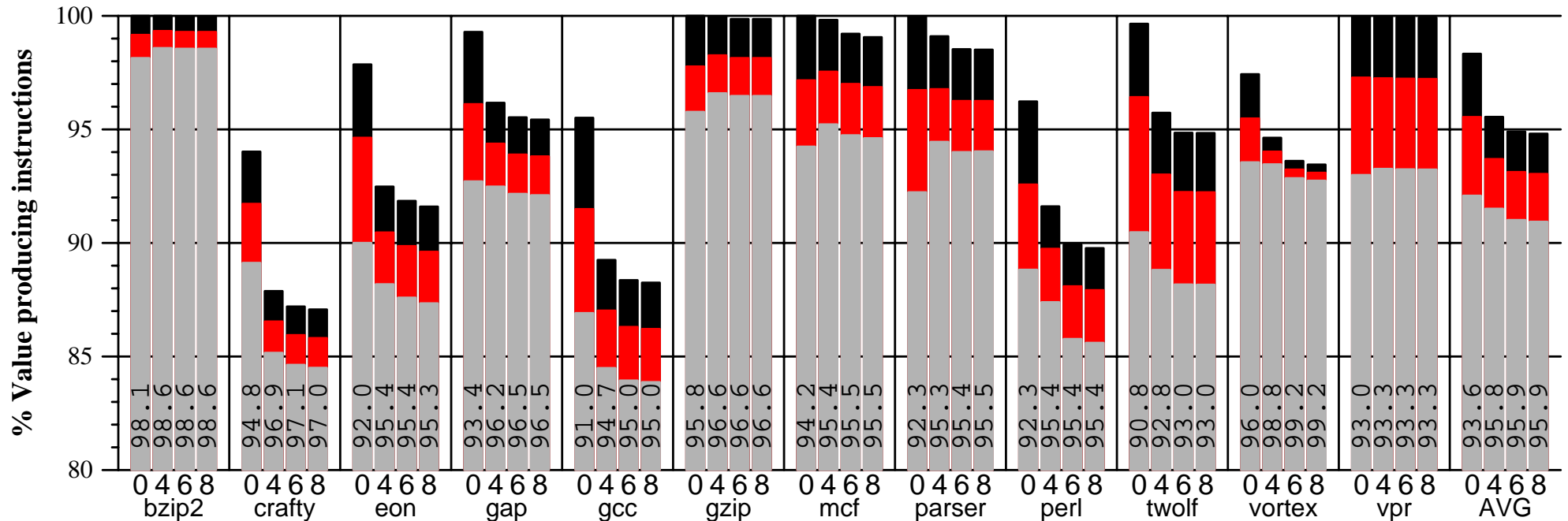
## Value communication optimizations

- Useless instruction elimination
- Register file optimizations
- Dynamic operation chaining

# Backup slides

---

# Predictor Parameter Sensitivity

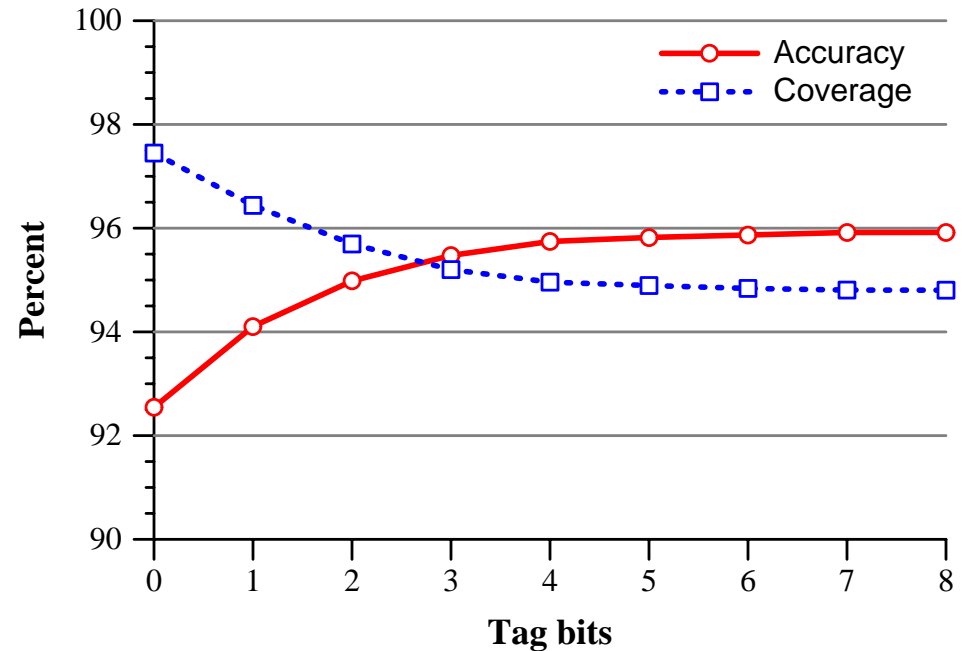
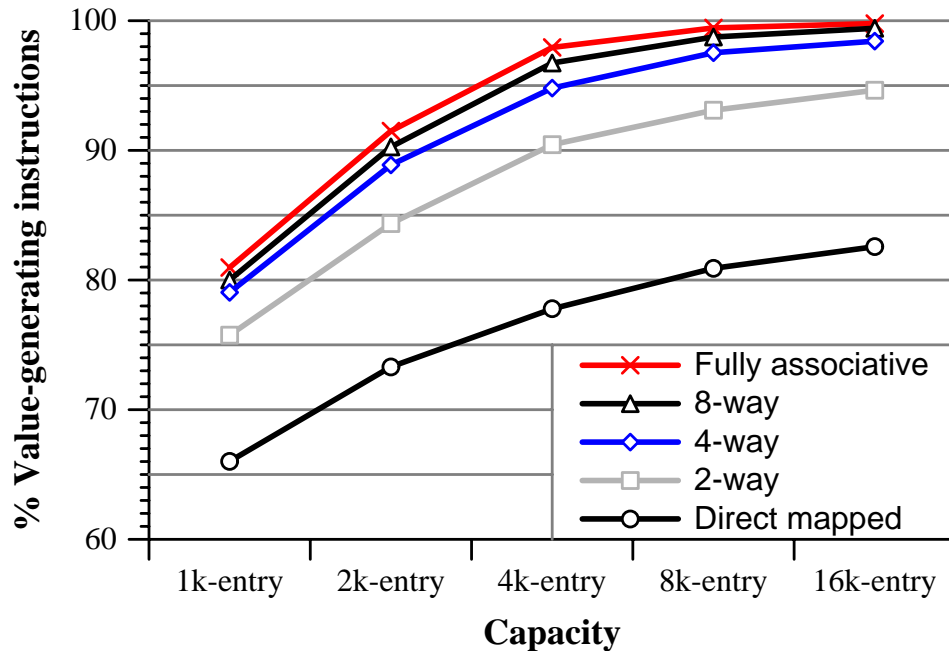


**Use of control flow signature increases accuracy**

**Little benefit beyond six signature bits**

- 98% of instructions have fewer than four branch directions available
- Most of the benefit achieved with 4 signature bits

# Predictor Parameter Sensitivity



## Coverage is a strong function of capacity and organization

- Behavior similar to other types of caches

## Increasing tag bits reduces destructive aliasing

- 6 bits OK for these programs
- Signature helps

# Degree of Use Predictor Details

---

## Predictor table

- 1K-sets x 4-way set associative, random replacement
- 6-bit signature + 6-bit tag + 3-bit degree + 2-bit confidence per entry
- Total storage requirement: **8.5 KB**

## Relaxed timing constraints

- Predictor table access overlaps I-cache access

# Observing an Instruction Stream

## Degree tracking table

- Operate on rename or retirement stream
- Saturating counter per architectural register
- Increment on use of associated register
- Clear when corresponding register is written

Degree Tracking Table

	PC	sig	uses	pred
r5:	j	y	1	2
r6:	k	x	0	1

PC<sub>j</sub>: ldq r5, 8(sp)

PC<sub>k</sub>: addl r1, r5, r6

## Predictor training

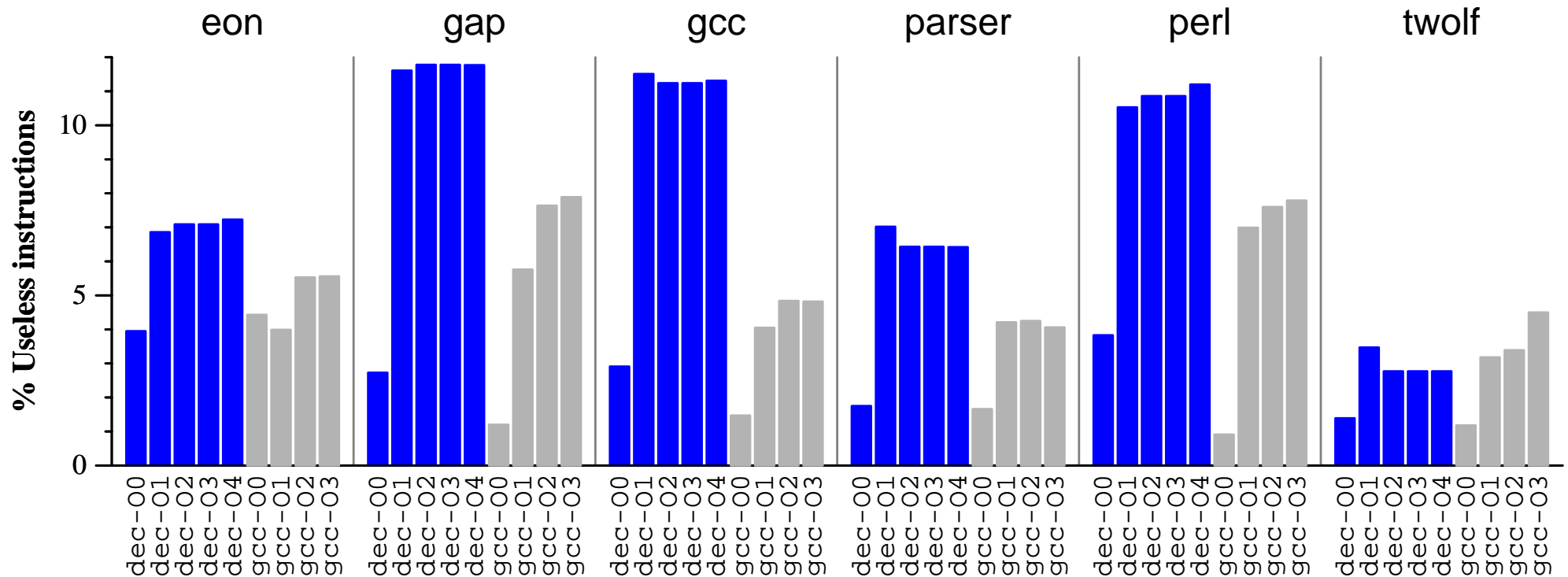
- Add **PC** of writer, **signature** when writer was renamed
- Send PC, signature, uses to predictor when entry cleared

## Detection of mispredictions

- Add **predicted degree of use** field for comparison
- **Underprediction** when uses exceed predicted degree of use
- **Overprediction** when overwrite occurs before predicted uses met



# Compiler Effect on Useless Instructions



## Useless instructions increase with compiler optimization level

- Compiler independent
- Both fraction and absolute number increase
- Due primarily to hoisting caused by instruction scheduling

# Eliminating Useless Instructions

Degree Tracking Table


	PC	uses	pred	PUT
r5:	i	0	2+	—
r6:	—	—	—	—

Predicted Useless Table

	V	instruction	ROB
4:	0	—	—

Reorder Buffer

	instruction	PUT
6:	ldq r5, 8(sp)	—
7:		
8:		
21:		


 PC<sub>i</sub>: ldq r5, 8(sp)  
 PC<sub>j</sub>: addl r1, r5, r6  
 PC<sub>k</sub>: stq r2, 0(r8)  
 ...  
 PC<sub>x</sub>: bis r0, r3, r6

Add **PUT** (predicted useless table) to track prediction status

- Entries contain valid bit, decoded instruction, and ROB entry pointer

**Augment** DTT, ROB with pointers into PUT

# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	0	2+	—
r6:	—	—	—	—

PC<sub>i</sub>: ldq r5, 8(sp)

PC<sub>j</sub>: addl r1, r5, r6

PC<sub>k</sub>: stq r2, 0(r8)

...

PC<sub>x</sub>: bis r0, r3, r6

Predicted Useless Table

	V	instruction	ROB
4:	1	addl r1, r5, r6	7 ←

Reorder Buffer

	instruction	PUT
6:	ldq r5, 8(sp)	—
7:		
8:		
21:		

When a degree of use zero prediction is made:

- A free PUT entry is allocated for the instruction

# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	1	2+	—
r6:	j	0	0	4

PC<sub>i</sub>: ldq r5, 8(sp)  
PC<sub>j</sub>: addl r1, r5, r6  
PC<sub>k</sub>: stq r2, 0(r8)  
...  
PC<sub>x</sub>: bis r0, r3, r6

Predicted Useless Table

V	instruction	ROB
4:	1 addl r1, r5, r6	7

Reorder Buffer

	instruction	PUT
6:	ldq r5, 8(sp)	—
7:		
8:		
21:		

## When a degree of use zero prediction is made:

- A free PUT entry is allocated for the instruction
- The DTT is updated with a pointer to the PUT entry

# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	1	2+	—
r6:	j	0	0	4

PC<sub>i</sub>: ldq r5, 8(sp)

PC<sub>j</sub>: addl r1, r5, r6

PC<sub>k</sub>: stq r2, 0(r8)

...

PC<sub>x</sub>: bis r0, r3, r6

Predicted Useless Table

V	instruction	ROB
4:	1 addl r1, r5, r6	7

Reorder Buffer

	instruction	PUT
6:	ldq r5, 8(sp)	—
7:	—	4
8:		
21:		

## When a degree of use zero prediction is made:

- A free PUT entry is allocated for the instruction
- The DTT is updated with a pointer to the PUT entry
- A dummy entry is placed in the ROB with a pointer to the PUT entry

# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	1	2+	—
r6:	j	0	0	4

PC<sub>i</sub>: ldq r5, 8(sp)

PC<sub>j</sub>: addl r1, r5, r6

PC<sub>k</sub>: stq r2, 0(r8)

...

→ PC<sub>x</sub>: bis r0, r3, r6

Predicted Useless Table

	V	instruction	ROB
4:	1	addl r1, r5, r6	7

Reorder Buffer

	instruction	PUT
6:		
7:	—	4
8:	stq r2, 0(r8)	—
21:		

Intervening instructions are handled normally except:

- A **use of a predicted useless register** causes the corresponding PUT entry to be placed into the instruction window for scheduling
- **Context switches** cause entire PUT to be flushed into the window

# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	1	2+	—
r6:	j	0	0	4

PC<sub>i</sub>: ldq r5, 8(sp)  
PC<sub>j</sub>: addl r1, r5, r6  
PC<sub>k</sub>: stq r2, 0(r8)  
...  
PC<sub>x</sub>: bis r0, r3, r6

Predicted Useless Table

	V	instruction	ROB
4:	1	addl r1, r5, r6	7

Reorder Buffer

	instruction	PUT
6:		
7:	—	4
8:	stq r2, 0(r8)	—
21:	bis r0,r3,r6	4

When an overwrite of the degree zero register is observed:

- The PUT pointer is copied into the ROB entry of the overwriting insn

# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	1	2+	—
r6:	x	0	1	—

PC<sub>i</sub>: ldq r5, 8(sp)  
PC<sub>j</sub>: addl r1, r5, r6  
PC<sub>k</sub>: stq r2, 0(r8)  
...  
PC<sub>x</sub>: bis r0, r3, r6

Predicted Useless Table

	V	instruction	ROB
4:	1	addl r1, r5, r6	7

Reorder Buffer

	instruction	PUT
6:		
7:	—	4
8:	stq r2, 0(r8)	—
21:	bis r0,r3,r6	4

## When an overwrite of the degree zero register is observed:

- The PUT pointer is copied into the ROB entry of the overwriting insn
- The DTT is updated normally



# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	1	2+	—
r6:	x	0	1	—

$PC_i$ : ldq r5, 8(sp)  
 $PC_j$ : addl r1, r5, r6  
 $PC_k$ : stq r2, 0(r8)  
 ...  
 $PC_x$ : bis r0, r3, r6



Predicted Useless Table

	V	instruction	ROB
4:	1	addl r1, r5, r6	7

Reorder Buffer

	instruction	PUT
6:		
7:	—	4
8:	stq r2, 0(r8)	—
21:	bis r0,r3,r6	4

## Retiring a predicted useless instruction:

- The **overwriting instruction** must be ready to retire
- All **intervening instructions** must be ready to retire

# Eliminating Useless Instructions

Degree Tracking Table

	PC	uses	pred	PUT
r5:	i	1	2+	—
r6:	x	0	1	—

PC<sub>i</sub>: ldq r5, 8(sp)

PC<sub>j</sub>: addl r1, r5, r6

PC<sub>k</sub>: stq r2, 0(r8)

...

PC<sub>x</sub>: bis r0, r3, r6



Predicted Useless Table

	V	instruction	ROB
4:	0	—	—

Reorder Buffer

	instruction	PUT
6:		
7:		
8:	stq r2, 0(r8)	—
21:	bis r0,r3,r6	4

## Retiring a predicted useless instruction:

- The overwriting instruction must be ready to retire
- All intervening instructions must be ready to retire
- The ROB and PUT entries are reclaimed

# Handling Loads under UIE

---

## Mispredicted useless loads may be delayed

- Handle as any OoO processor

## Loads may have side effects

- Memory-mapped I/O, page faults, illegal addresses
- Must execute these loads
- Solutions
  - ISA change to mark such loads
  - Probe TLB to verify cacheable page

# UIE Processor Configurations

---

## Both

- Big YAGS branch predictor, RAS, cascaded indirect predictor
- 64 KB 2-way set associative L1 caches, unified 2MB 4-way L2
- 4-wide fetch, issue, retire

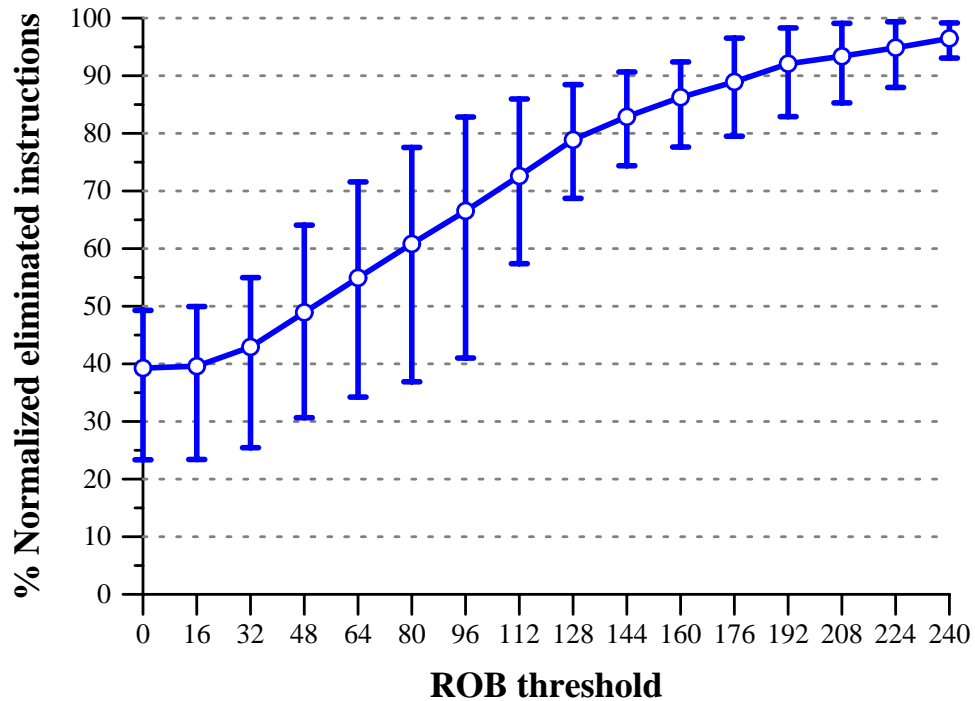
## Baseline (1/2 IBM Power 4)

- 1 simple integer, 1 complex integer, 1 branch
- 1 load, 1 store
- 1 simple FP, 1 complex FP

## Constrained (Transmeta Crusoe)

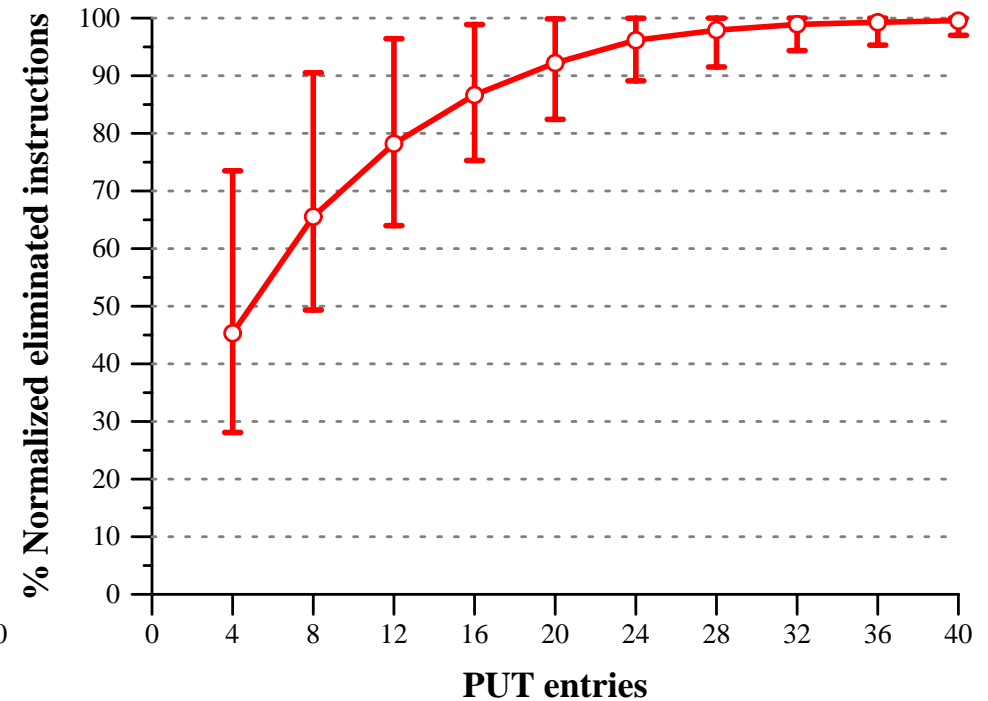
- 1 complex integer, 1 branch
- 1 load/store
- 1 complex FP

# UIE Parameter Sensitivity



## Higher ROB threshold

- More instructions eliminated
- Performance peaks earlier due to diminishing returns plus larger retirement holdup



## Larger PUT size

- More instructions eliminated
- More hardware overhead
- Scaling required with number of instructions in flight

# Dynamic Chaining vs. ILDP

---

**No changes required to ISA**

**No compiler support required**

- Chains are identified dynamically by degree of use prediction

**Leverages strengths of standard back end**

- Values that need wide distribution
- Deep lookahead in instruction stream for ILP