
RENO: a RENAME-based instruction Optimizer

Vlad Petric, Tingting Sha, Amir Roth

University of Pennsylvania

RENO...

Is a modified MIPS R10K renamer

Implements dynamic versions of static optimizations

Unifies these previously proposed optimizations

- Move Elimination [“Unified Renaming”, Jourdan+]
- Register Allocation [“Register Integration”, Petric+]
- Common Subexpression Elimination [same]

Adds Constant Folding

- Synergistic with other optimizations

Has a simple implementation

Why Optimize in Hardware?

Static optimizations

- file/function boundaries

Dynamic optimizations (RENO)

- + no boundaries

Why Optimize in Hardware?

Static optimizations

- file/function boundaries
- conservative memory info

Dynamic optimizations (RENO)

- + no boundaries
- + can speculate

Why Optimize in Hardware?

Static optimizations

- file/function boundaries
- conservative memory info
- correct on all paths

Dynamic optimizations (RENO)

- + no boundaries
- + can speculate
- + correct on dynamic path

Why Optimize in Hardware?

Static optimizations

- file/function boundaries
- conservative memory info
- correct on all paths
- + removes insns at fetch

Dynamic optimizations (RENO)

- + no boundaries
- + can speculate
- + correct on dynamic path
- removes insns at rename

Why Optimize in Hardware?

Static optimizations

- file/function boundaries
- conservative memory info
- correct on all paths
- + removes insns at fetch

Dynamic optimizations (RENO)

- + no boundaries
- + can speculate
- + correct on dynamic path
- removes insns at rename

RENO complements static compiler optimizations

- Removes 23% of instructions from –O4 optimized programs

Overview

RENO framework: physical register sharing

RENO optimizations

- Focus on constant folding

Hardware implementation

Performance evaluation

Physical Register Sharing

RENO examines dynamic instructions “one at a time”

Optimizes away some instructions

How? **physical register sharing**

- Correct value already exists in some physical register
 - 1 Instruction’s map-table output set to that register
 - 2 Instruction itself bypasses execution core

Performance and implementation benefits

- + Reduces dataflow graph latency
- + Amplifies execution core bandwidth
- + Map-table manipulations only

RENO optimization: way to detect sharing opportunities

Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1 ... addi r1, 0 → r2 ... op2 _, r2 → _		

```
graph TD; op1["op1 _, _ → r1"] --> addi["addi r1, 0 → r2"]; addi --> op2["op2 _, r2 → _"]
```

Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1 ... addi r1, 0 → r2 ... op2 _, r2 → _	r1 := [p3]	op1 _, _, → p3

```
graph TD; R1[ ] --> R1Action[r1 := [p3]]; R2[ ] --> R2Action[r2 := [p3]];
```

Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3]	op1 _, _, → p3
...		↓
addi r1, 0 → r2	r2 := [p9]	addi p3, 0 → p9
...		
op2 _, r2 → _		

Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3]	op1 _, _, → p3
...		↓
addi r1, 0 → r2	r2 := [p9]	addi p3, 0 → p9
...		↓
op2 _, r2 → _	_ := []	op2 _, p9 → _

RENO_{ME}: Move Elimination

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1 ... addi r1, 0 → r2 ... op2 _, r2 → _		

Diagram illustrating the execution dataflow between instructions:

- The first instruction (op1) has a result **r1** (green).
- The second instruction (**addi**) has a source **r1** (green) and a result **r2** (purple).
- The third instruction (op2) has a source **r2** (purple) and a result **_**.

Arrows show the flow of values from the result of one instruction to the source of the next.

Optimization detection: **addi _, 0 → _**

RENO_{ME}: Move Elimination

Static insn	MapTable action	Execution dataflow
$op1 _, _ \rightarrow r1$... $\text{addi } r1, 0 \rightarrow r2$... $op2 _, r2 \rightarrow _$	$r1 := [p3]$	$op1 _, _, \rightarrow p3$

Optimization detection: $\text{addi } _, 0 \rightarrow _$

RENO_{ME}: Move Elimination

Static insn	MapTable action	Execution dataflow
$op1 _, _ \rightarrow r1$... $\text{addi } r1, 0 \rightarrow r2$... $op2 _, r2 \rightarrow _$	$r1 := [p3]$ $r2 := [p3] \text{ (reuse)}$	$op1 _, _, \rightarrow p3$

Optimization detection: $\text{addi } _, 0 \rightarrow _$

RENO_{ME}: Move Elimination

Static insn	MapTable action	Execution dataflow
$op1 _, _ \rightarrow r1$... $\text{addi } r1, 0 \rightarrow r2$... $op2 _, r2 \rightarrow _$	$r1 := [p3]$ $r2 := [p3] \text{ (reuse)}$ $_ := []$	$op1 _, _, \rightarrow p3$ $op2 _, p3 \rightarrow _$

Optimization detection: $\text{addi } _, 0 \rightarrow _$

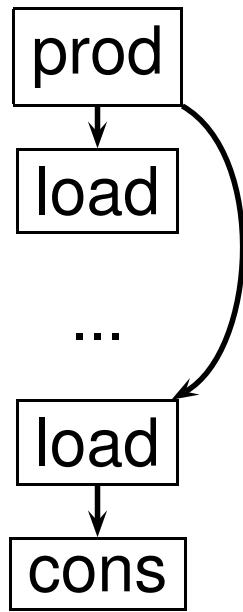
RENO_{ME}: Move Elimination

Static insn	MapTable action	Execution dataflow
$op1 _, _ \rightarrow r1$... $\text{addi } r1, 0 \rightarrow r2$... $op2 _, r2 \rightarrow _$	$r1 := [p3]$ $r2 := [p3] \text{ (reuse)}$ $_ := []$	$op1 _, _, \rightarrow p3$ $op2 _, p3 \rightarrow _$

Optimization detection: $\text{addi } _, 0 \rightarrow _$

- + Latency: move removed from dataflow graph
- + Bandwidth: move removed from execution core

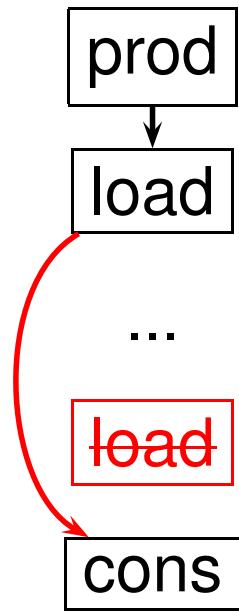
RENO_{CSE/RA}: Register Integration



CSE/RegAlloc using physical register sharing

- RegAlloc: short-circuit stack store-load pairs
 - aka Speculative memory bypassing
 - Optimization detection: memoization table (expensive)

RENO_{CSE/RA}: Register Integration



CSE/RegAlloc using physical register sharing

- RegAlloc: short-circuit stack store-load pairs
 - aka Speculative memory bypassing
 - Optimization detection: memoization table (expensive)

RENO_{CSE/RA}: Register Integration



CSE/RegAlloc using physical register sharing

- RegAlloc: short-circuit stack store-load pairs
 - aka Speculative memory bypassing
 - Optimization detection: memoization table (expensive)

RENO_{CSE/RA}: Register Integration



CSE/RegAlloc using physical register sharing

- RegAlloc: short-circuit stack store-load pairs
 - aka Speculative memory bypassing
 - Optimization detection: memoization table (expensive)

RENO_{CF}: Constant Folding

RENO_{ME}: register moves (**addi** _, 0 → _)

- Not common: compilation artifacts

RENO_{CF}: Constant Folding

RENO_{ME}: register moves (**addi** _, 0 → _)

- Not common: compilation artifacts

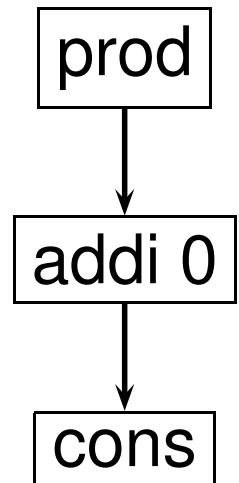
RENO_{CF}: register-immediate additions (**addi** _, 4 → _)

- + Surprisingly common: address calculation, etc.
 - 12% in SPECint
 - 16% in MediaBench
- + Synergy with CSE/RegAlloc
 - Main benefit of CSE/RegAlloc: eliminating loads
 - Address calculations are dataflow “glue” between loads
 - Eliminate without expensive table lookups

Physical Register Sharing++

RENO_{ME/CSE/RA}

- Instructions don't compute “new” results
- Physical register sharing sufficient



Physical Register Sharing++

RENO_{ME/CSE/RA}

- Instructions don't compute “new” results
- Physical register sharing sufficient

prod

addi 0

cons

Physical Register Sharing++

RENO_{ME/CSE/RA}

- Instructions don't compute “new” results
- Physical register sharing sufficient

prod

addi 0

cons

RENO_{CF}

- Instructions compute “new” results
- Physical register sharing insufficient

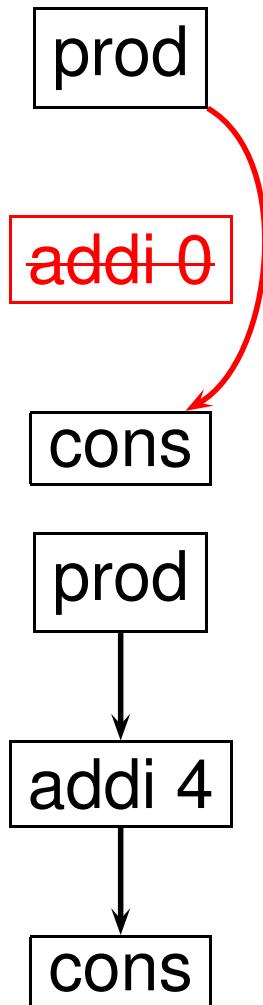
Physical Register Sharing++

RENO_{ME/CSE/RA}

- Instructions don't compute “new” results
- Physical register sharing sufficient

RENO_{CF}

- Instructions compute “new” results
- Physical register sharing insufficient



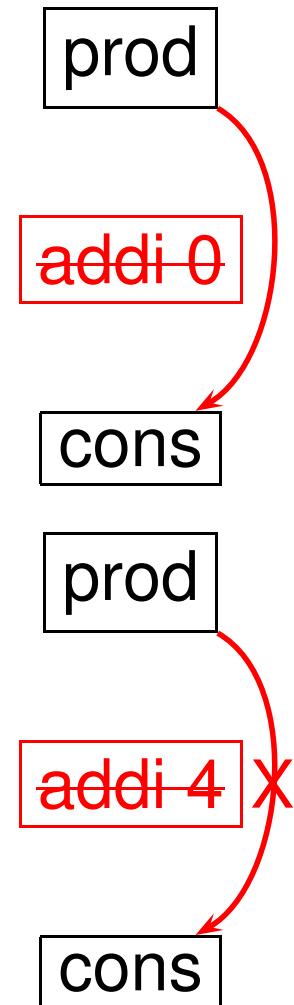
Physical Register Sharing++

RENO_{ME/CSE/RA}

- Instructions don't compute “new” results
- Physical register sharing sufficient

RENO_{CF}

- Instructions compute “new” results
- Physical register sharing insufficient



Physical Register Sharing++

RENO_{ME/CSE/RA}

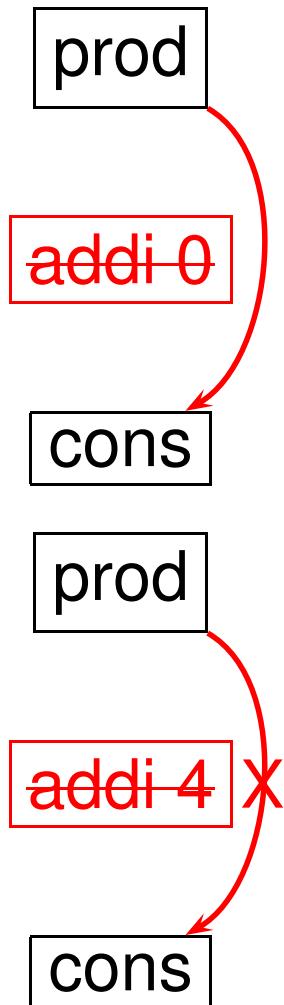
- Instructions don't compute "new" results
- Physical register sharing sufficient

RENO_{CF}

- Instructions compute "new" results
- Physical register sharing insufficient

Solution: extended renaming semantics

- Extended map table: LREG := [PREG, DISP]
- +DISP implicitly fused to any consumer



Physical Register Sharing++

RENO_{ME/CSE/RA}

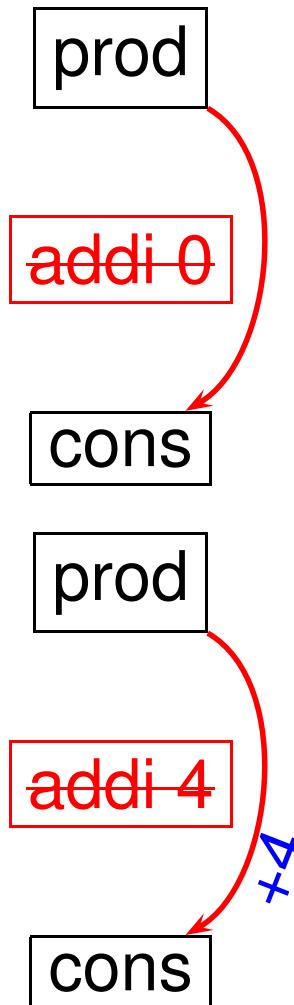
- Instructions don't compute "new" results
- Physical register sharing sufficient

RENO_{CF}

- Instructions compute "new" results
- Physical register sharing insufficient

Solution: extended renaming semantics

- Extended map table: LREG := [PREG, DISP]
- +DISP implicitly fused to any consumer



Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1		
...		
addi r1, 4 → r2		
...		
addi r2, 16 → r3		
...		
op2 _, r3 → _		

```
graph TD; op1["op1 _, _ → r1"] --> addi1["addi r1, 4 → r2"]; addi1 --> addi2["addi r2, 16 → r3"]; addi2 --> op2["op2 _, r3 → _"];
```

Conventional Processing

Static insn	MapTable action	Execution dataflow
$op1 _, _ \rightarrow r1$	$r1 := [p3]$	$op1 _, _ \rightarrow p3$
...		
$addi r1, 4 \rightarrow r2$		
...		
$addi r2, 16 \rightarrow r3$		
...		
$op2 _, r3 \rightarrow _$		

```
graph TD; A[op1 _, _ → r1] --> B[addi r1, 4 → r2]; B --> C[addi r2, 16 → r3]; C --> D[op2 _, r3 → _]
```

Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3]	op1 _, _ → p3
...		
addi r1, 4 → r2	r2 := [p9]	addi p3, 4 → p9
...		
addi r2, 16 → r3		
...		
op2 _, r3 → _		

The diagram illustrates the conventional processing flow across three columns: Static insn, MapTable action, and Execution dataflow. The Static insn column lists instructions like op1 and addi. The MapTable action column shows the generation of memory addresses [p3] and [p9]. The Execution dataflow column shows the final executed instructions with their results boxed. Colored arrows (green, purple, blue) connect the corresponding rows between the three columns, indicating the flow of data from static instructions to map table actions and finally to execution dataflow.

Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3]	op1 _, _ → p3
...		
addi r1, 4 → r2	r2 := [p9]	addi p3, 4 → p9
...		
addi r2, 16 → r3	r3 := [p6]	addi p9, 16 → p6
...		
op2 _, r3 → _		

Conventional Processing

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3]	op1 _, _ → p3
...		
addi r1, 4 → r2	r2 := [p9]	addi p3, 4 → p9
...		
addi r2, 16 → r3	r3 := [p6]	addi p9, 16 → p6
...		
op2 _, r3 → _	_ := []	op2 _, p6 → _

RENO_{CF}: Constant Folding

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1		
...		
addi r1, 4 → r2		
...		
addi r2, 16 → r3		
...		
op2 _, r3 → _		

The diagram illustrates the flow of constant folding through three stages: Static insn, MapTable action, and Execution dataflow. The first four rows represent the initial state where constants are being loaded into registers (r1, r2, r3). The fifth row represents the final state where the constant values are being used in an operation (op2). Four colored arrows (green, red, purple, blue) point downwards from the first four rows to the fifth row, indicating the progression of the folding process.

RENO_{CF}: Constant Folding

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3, 0]	op1 _, _ → p3
...		
addi r1, 4 → r2		
...		
addi r2, 16 → r3		
...		
op2 _, r3 → _		

The diagram illustrates the process of constant folding. It shows four rows of data corresponding to the columns: Static insn, MapTable action, and Execution dataflow. The first row contains the instruction 'op1 _, _ → r1' with its corresponding MapTable action 'r1 := [p3, 0]' and execution dataflow 'op1 _, _ → p3'. A green arrow points from 'r1' in the first row to 'r1' in the second row. The second row contains the instruction 'addi r1, 4 → r2' with its corresponding MapTable action and execution dataflow. A purple arrow points from 'r2' in the second row to 'r2' in the third row. The third row contains the instruction 'addi r2, 16 → r3' with its corresponding MapTable action and execution dataflow. A blue arrow points from 'r3' in the third row to 'r3' in the fourth row. The fourth row contains the instruction 'op2 _, r3 → _'.

RENO_{CF}: Constant Folding

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3, 0]	op1 _, _ → p3
...		
addi r1, 4 → r2	r2 := [p3, 4]	
...		
addi r2, 16 → r3		
...		
op2 _, r3 → _		

```
graph TD; r1[r1 := [p3, 0]] --> r2[r2 := [p3, 4]]; r2 --> r3[r3]; r3 --> result["op1 _, _ → p3"];
```

RENO_{CF}: Constant Folding

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3, 0]	op1 _, _ → p3
...		
addi r1, 4 → r2	r2 := [p3, 4]	
...		
addi r2, 16 → r3	r3 := [p3, 20]	
...		
op2 _, r3 → _		

```
graph TD; r1[0] --> addi1[4]; addi1 --> addi2[16]; addi2 --> op2[r3];
```

Renamer accumulates displacements

RENO_{CF}: Constant Folding

Static insn	MapTable action	Execution dataflow
op1 _, _ → r1	r1 := [p3, 0]	op1 _, _ → p3
...		
addi r1, 4 → r2	r2 := [p3, 4]	
...		
addi r2, 16 → r3	r3 := [p3, 20]	
...		
op2 _, r3 → _	_ := []	op2 _, (p3+20) → _

Renamer accumulates displacements

Consumer performs fused addi-op2 operation

RENO Implementation

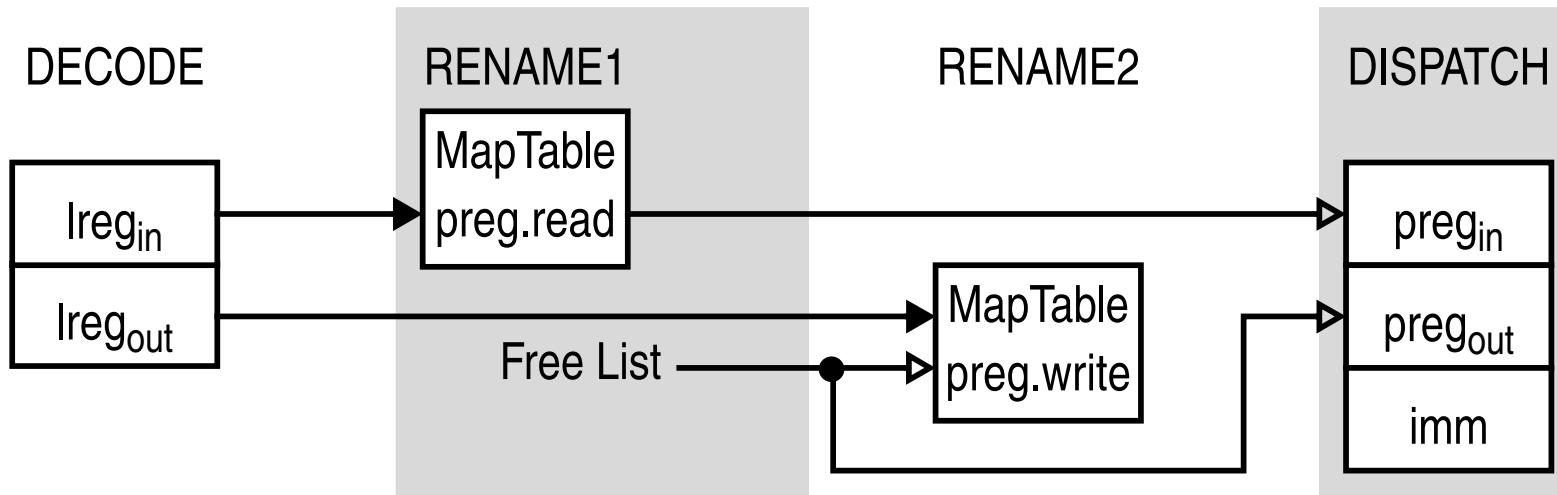
Rename

- RENO_{ME/CSE/RA}: physical register sharing
- RENO_{CF}: physical register sharing++

Execute

- RENO_{ME/CSE/RA}: nothing
- RENO_{CF}: fused “addi-X” functional units

Conventional Renaming

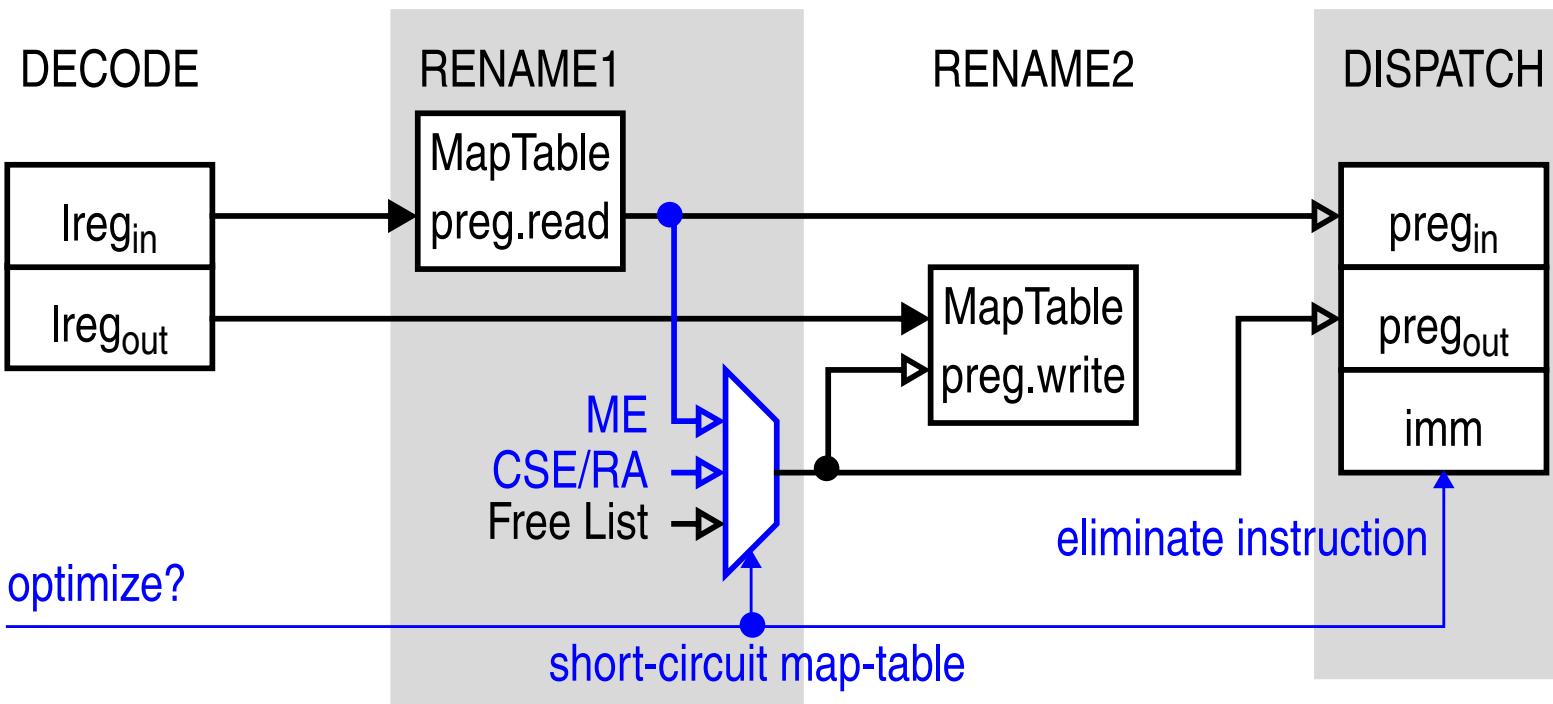


Example: scalar, one register input per instruction

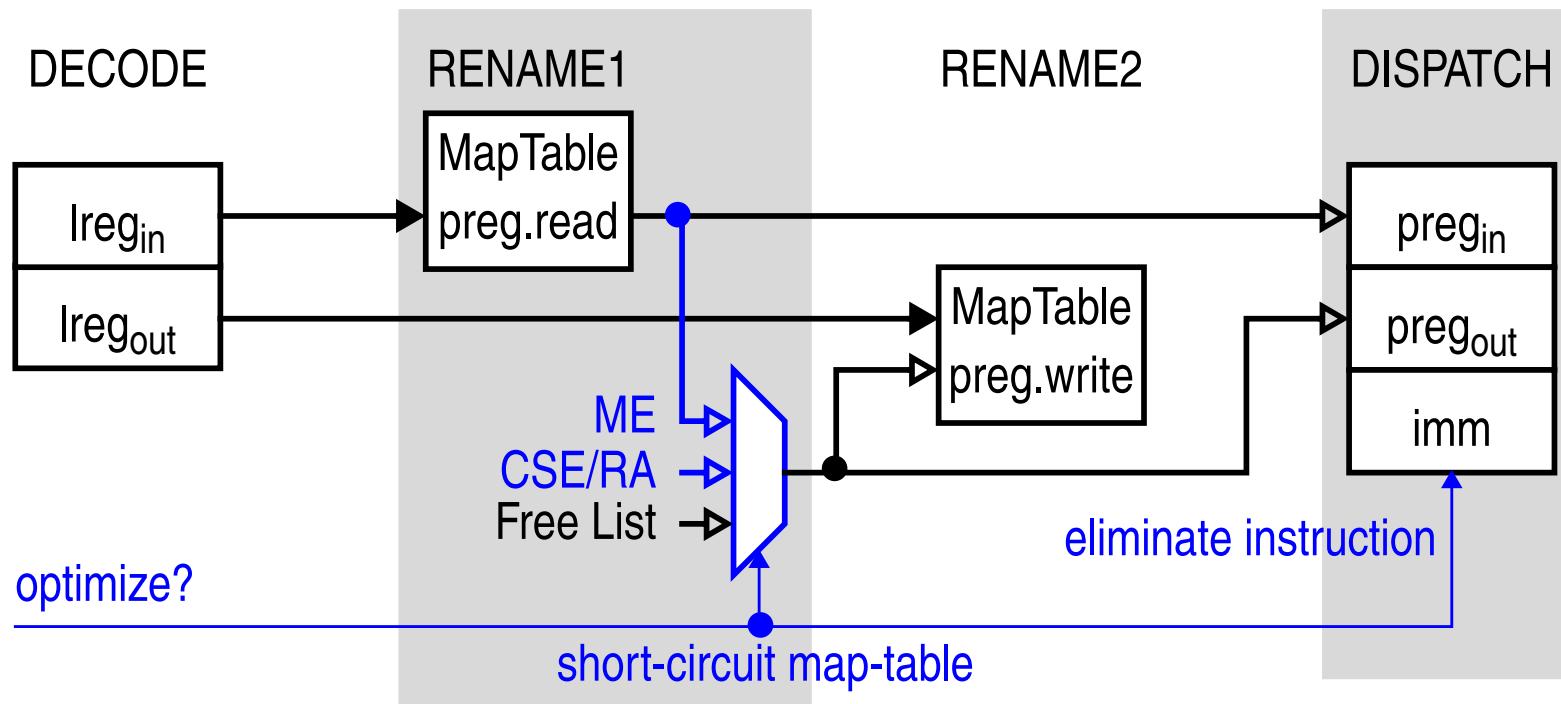
- See paper for superscalar



RENO_{ME/CSE/RA} Renaming



RENO *ME/CSE/RA* Renaming

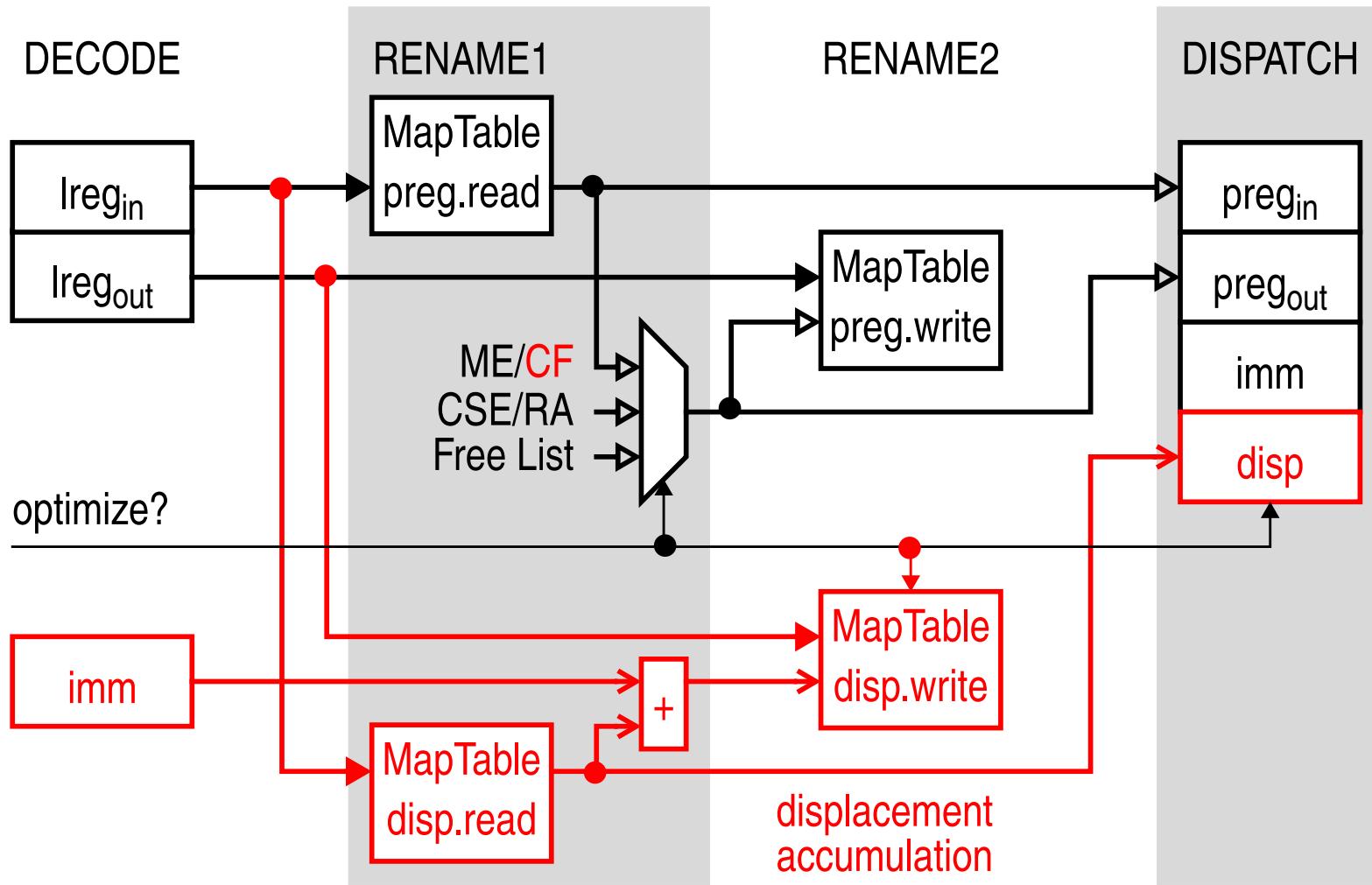


Superscalar? see paper

- $O(N^2)$: same as conventional renaming
- Don't optimize dependent insns in same cycle
- We *believe* additional renaming stages are unnecessary



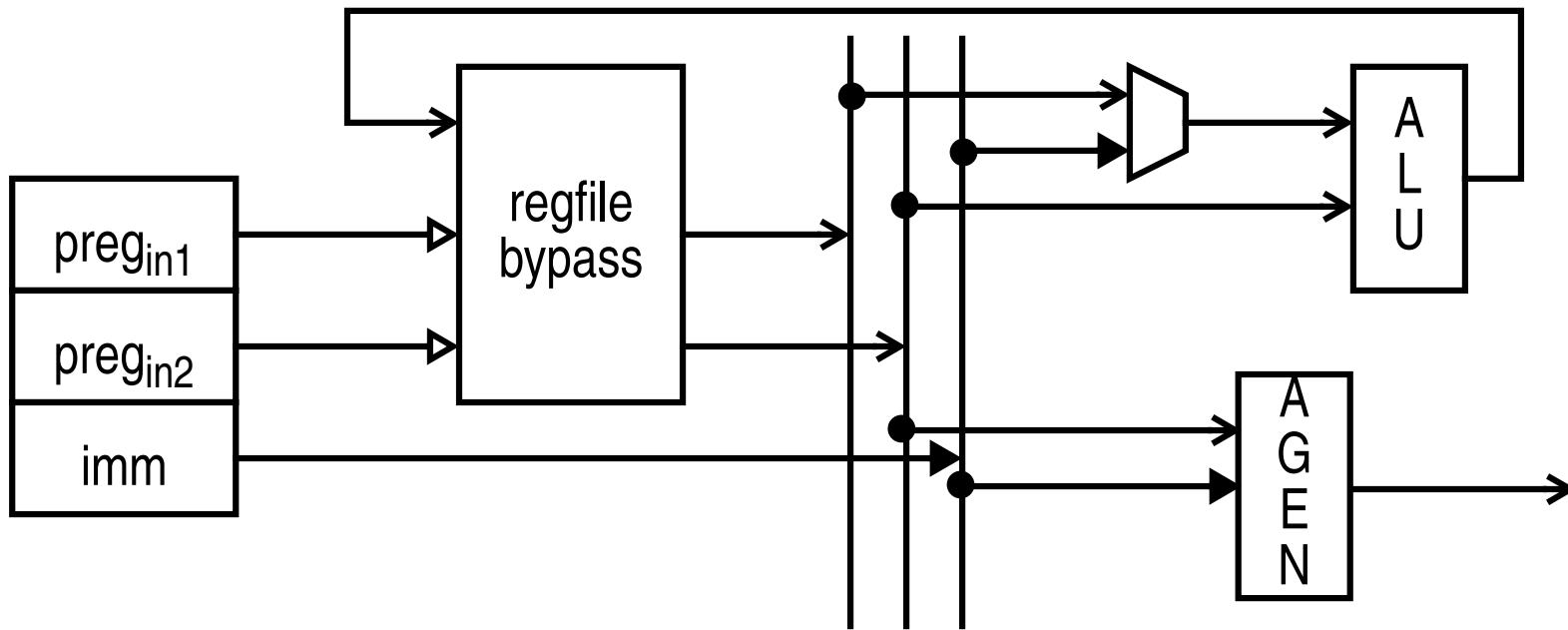
RENO_{CF} Renaming



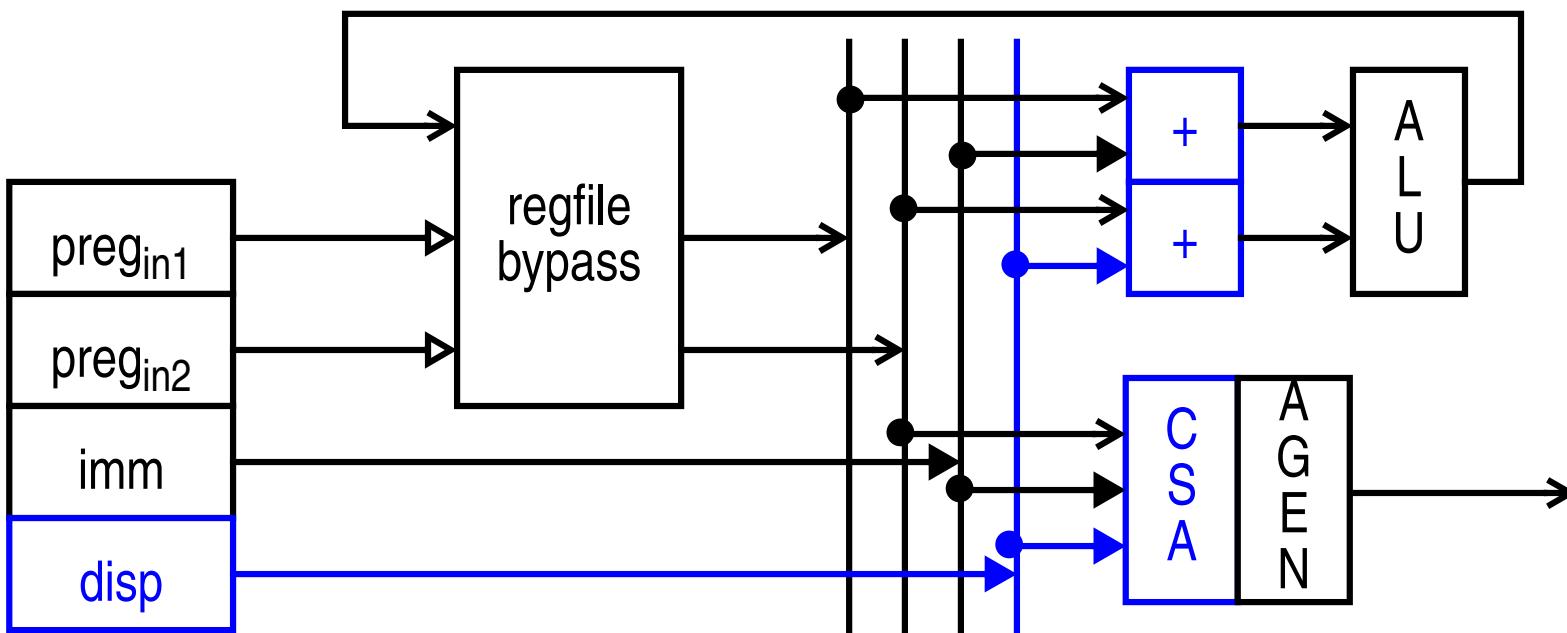
RENO_{ME} + parallel displacement accumulation circuit



Conventional Execution



RENO_{CF} Execution

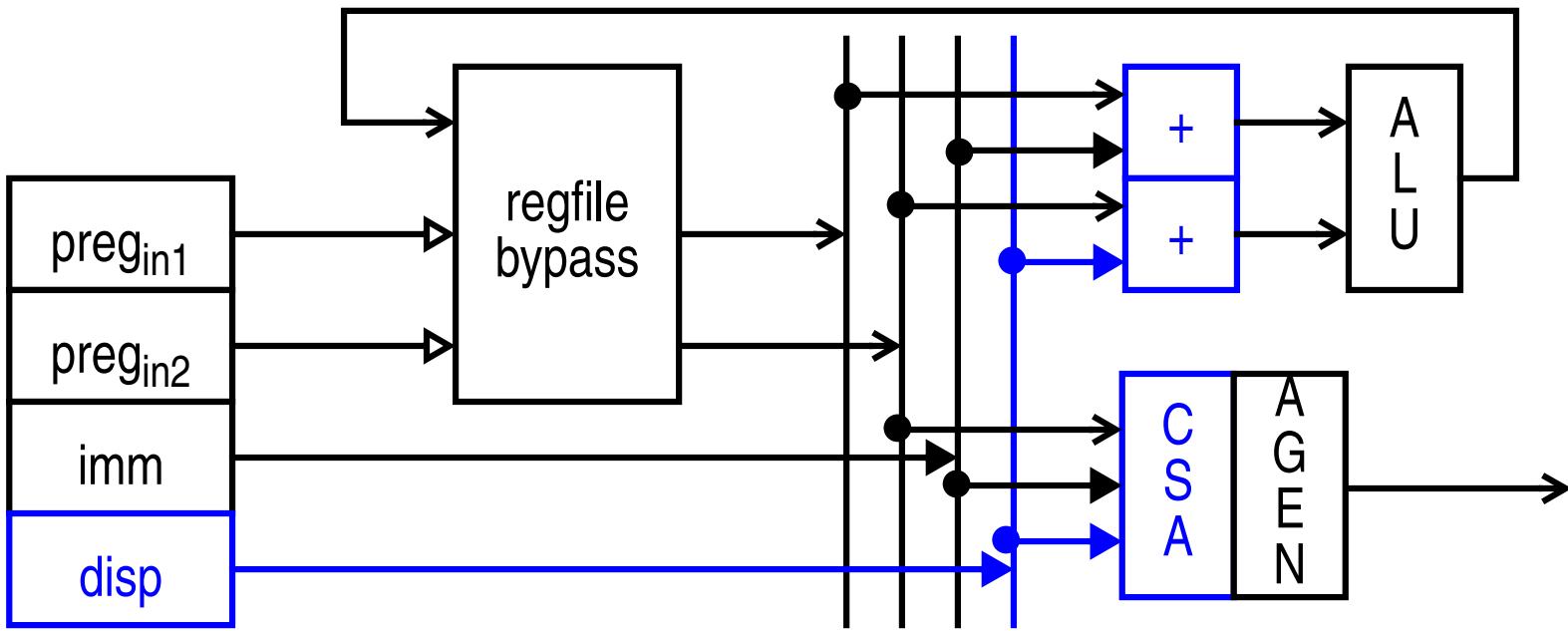


All operations take an additional (immediate) input

- 1-cycle penalty for general “addi-X” fusion
- + 0-cycle penalty “addi-addi” fusion (carry-save adder)
 - Most common and performance critical case (agen)



RENO_{CF} Execution



All operations take an additional (immediate) input

- 1-cycle penalty for general “addi-X” fusion
- + 0-cycle penalty “addi-addi” fusion (carry-save adder)
 - Most common and performance critical case (agen)

Regfile/bypass not complicated



Experimental Evaluation

What we care about

- Elimination rates (i.e., “coverage”) and performance
- RENO_{CF} synergy with RENO_{CSE/RA}
- See paper for sensitivity, bandwidth amplification, etc.

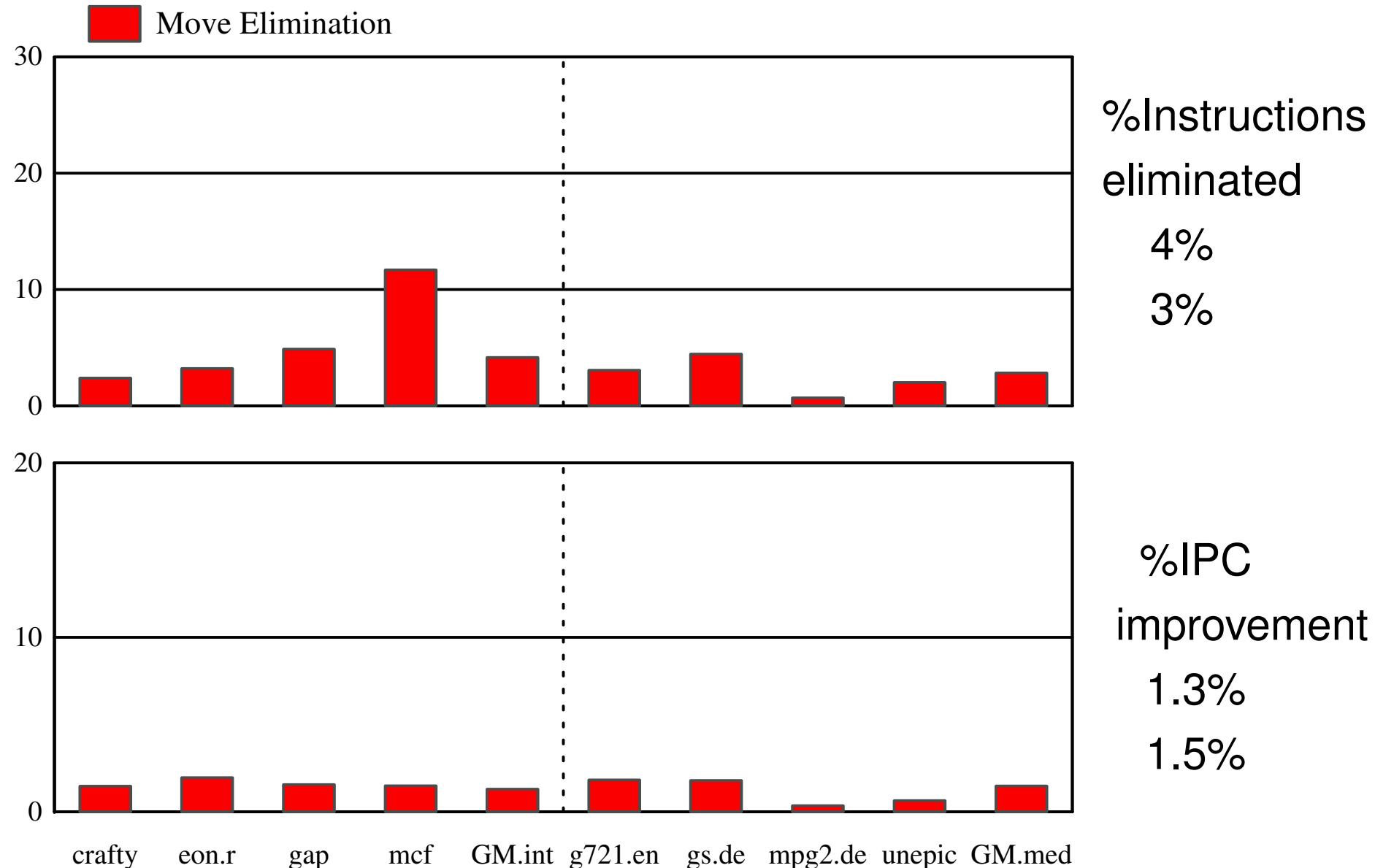
Benchmarks: SPECint2000, MediaBench

- DEC OSF **-O4**

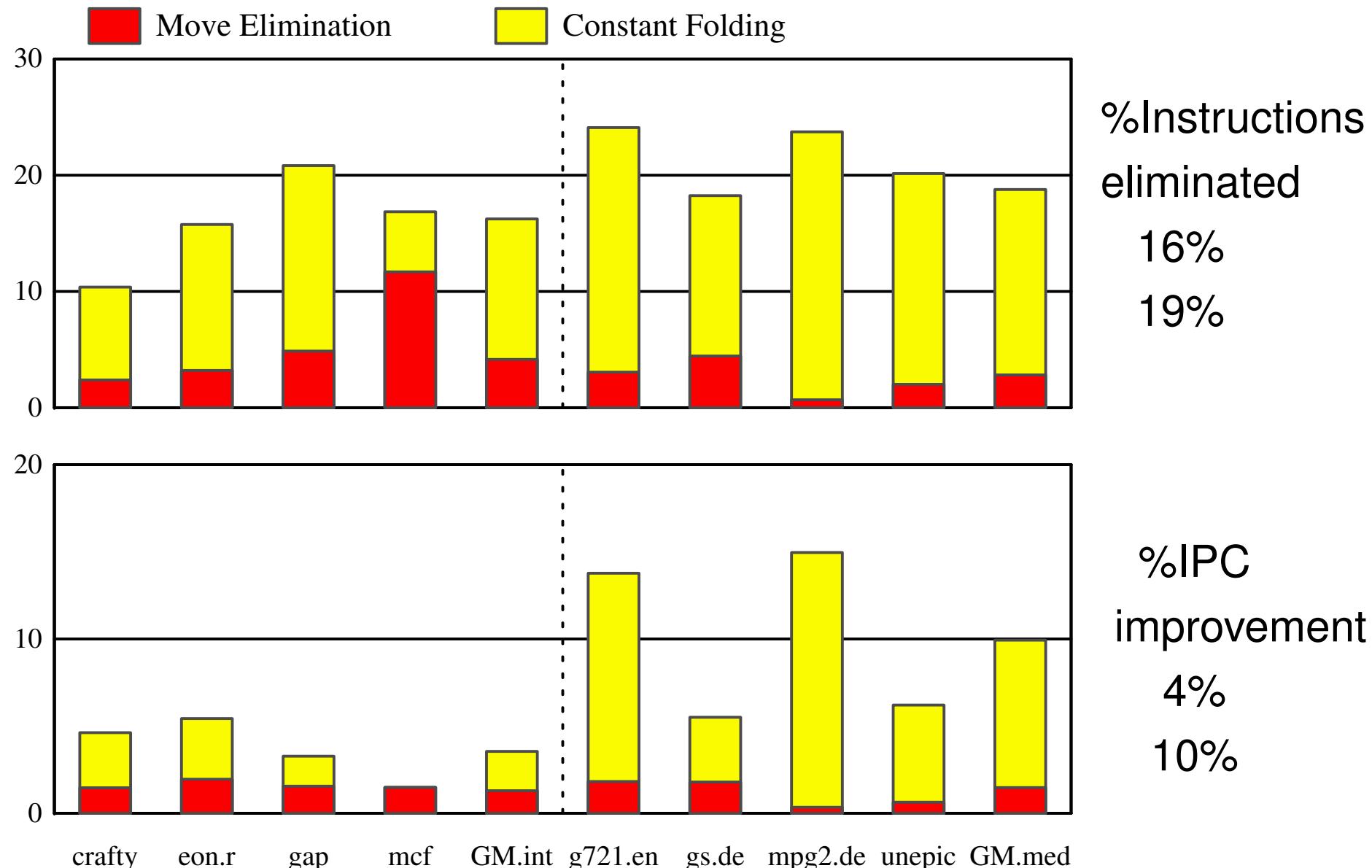
Simulator: Alpha SimpleScalar++

- 4-wide, out-of-order, 128-entry ROB, 50-entry issue queue
- 32KB D\$, 16KB I\$, 512KB L2
- RENO_{CF}: 16-bit displacements
- RENO_{CSE/RA}: 256-entry reuse table (**loads only**)

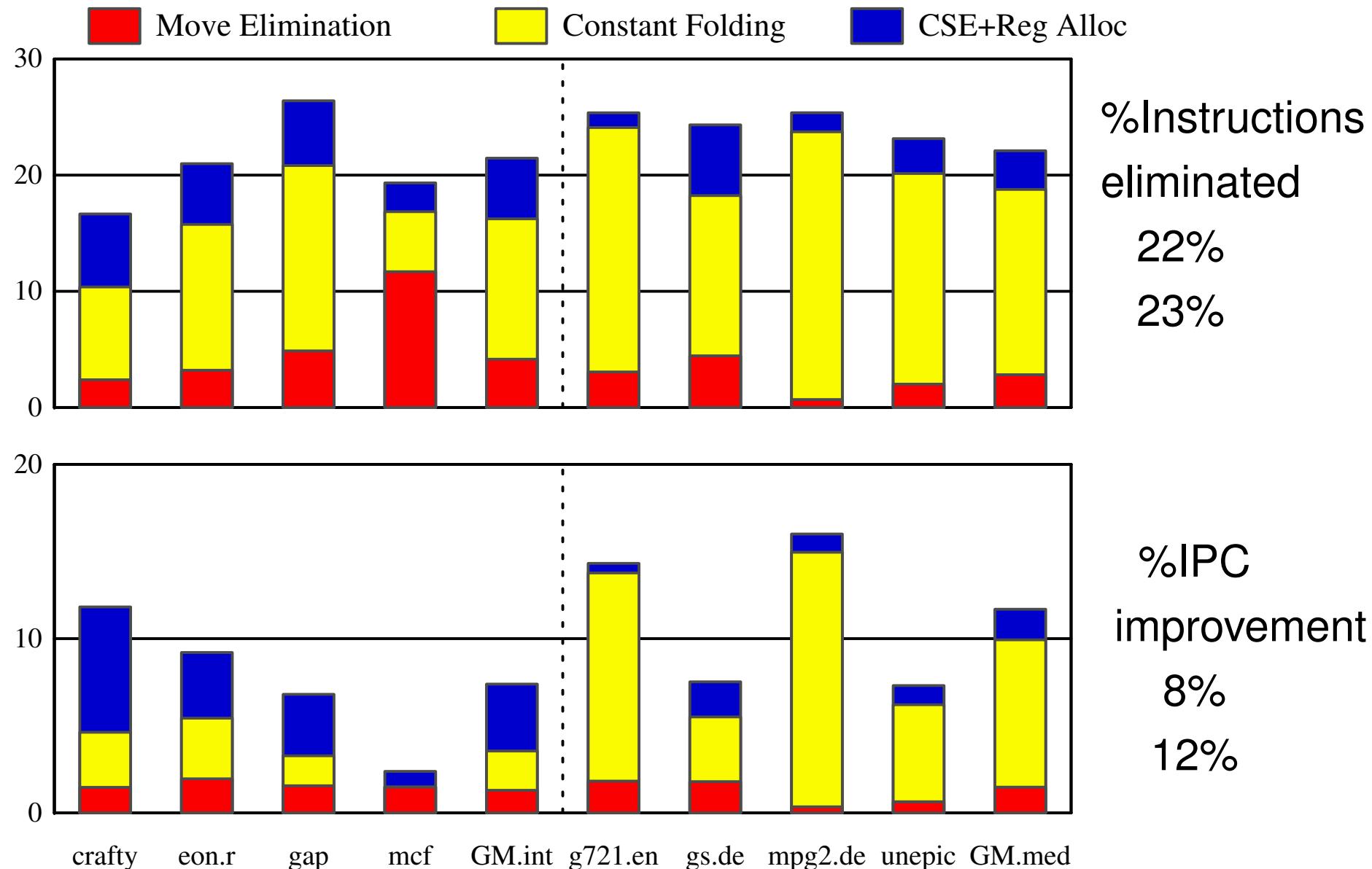
Coverage and Performance



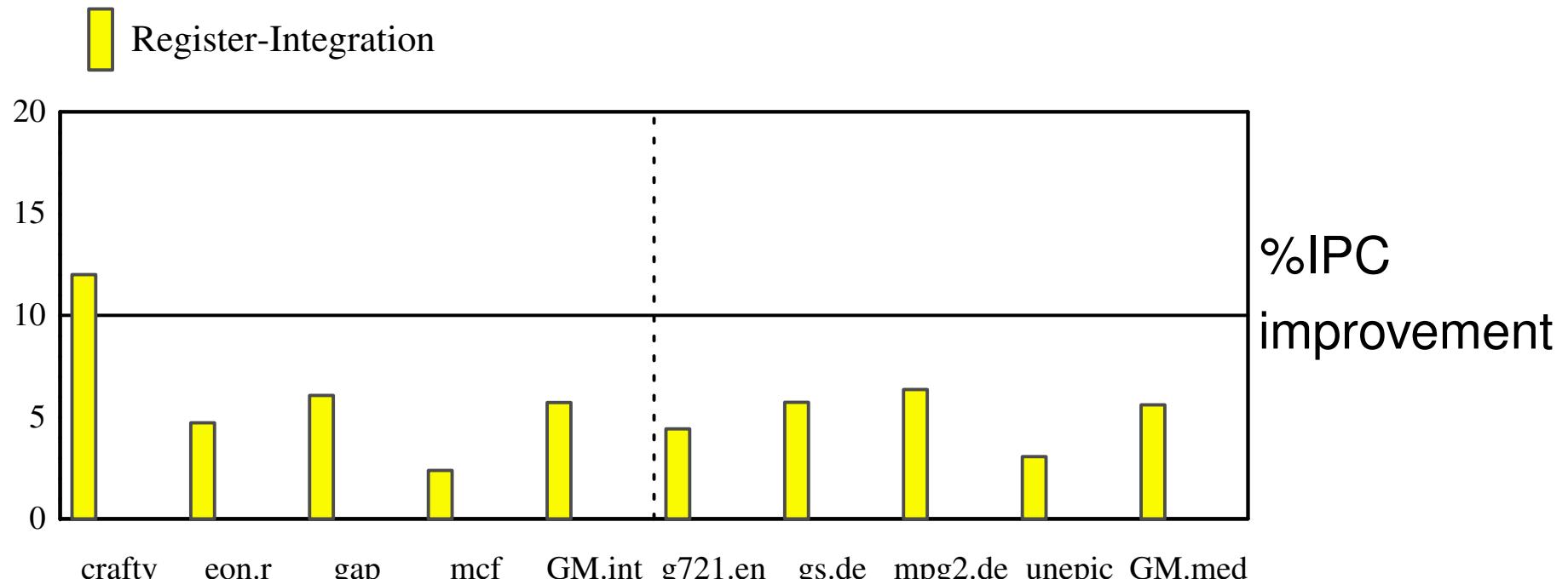
Coverage and Performance



Coverage and Performance



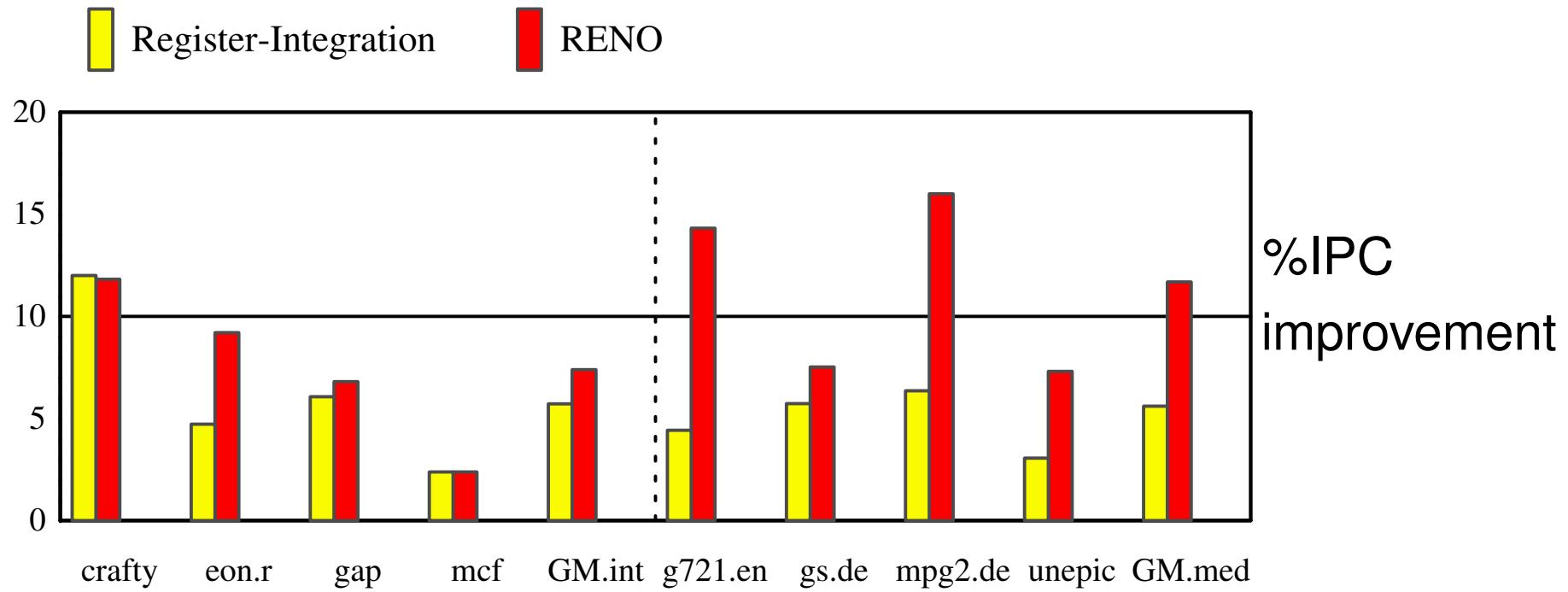
RENO_{CSE/RA} / RENO_{CF} Synergy



Register Integration: CSE for all, no folding

- Operations must be redundant
- Large tables, many lookups

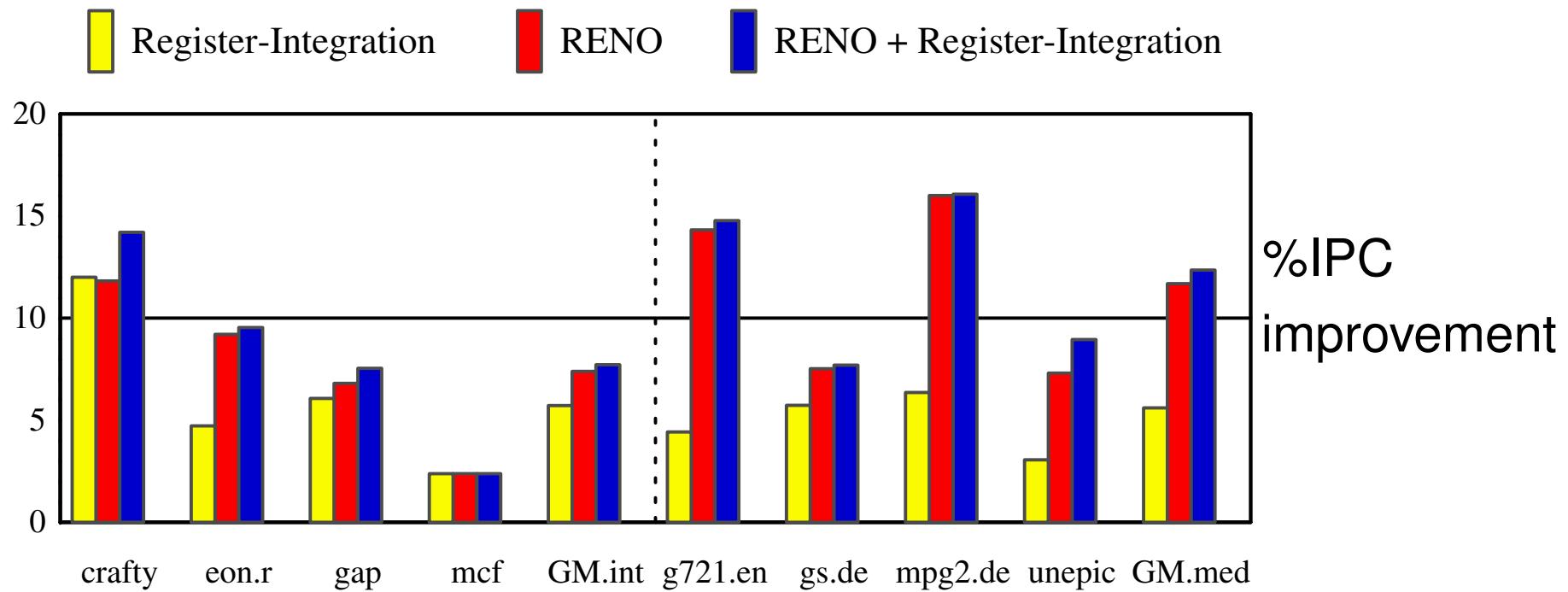
RENO_{CSE/RA} / RENO_{CF} Synergy



RENO: CSE for loads + folding

- + Better performance: addi's need not be redundant
- + Smaller tables, fewer lookups

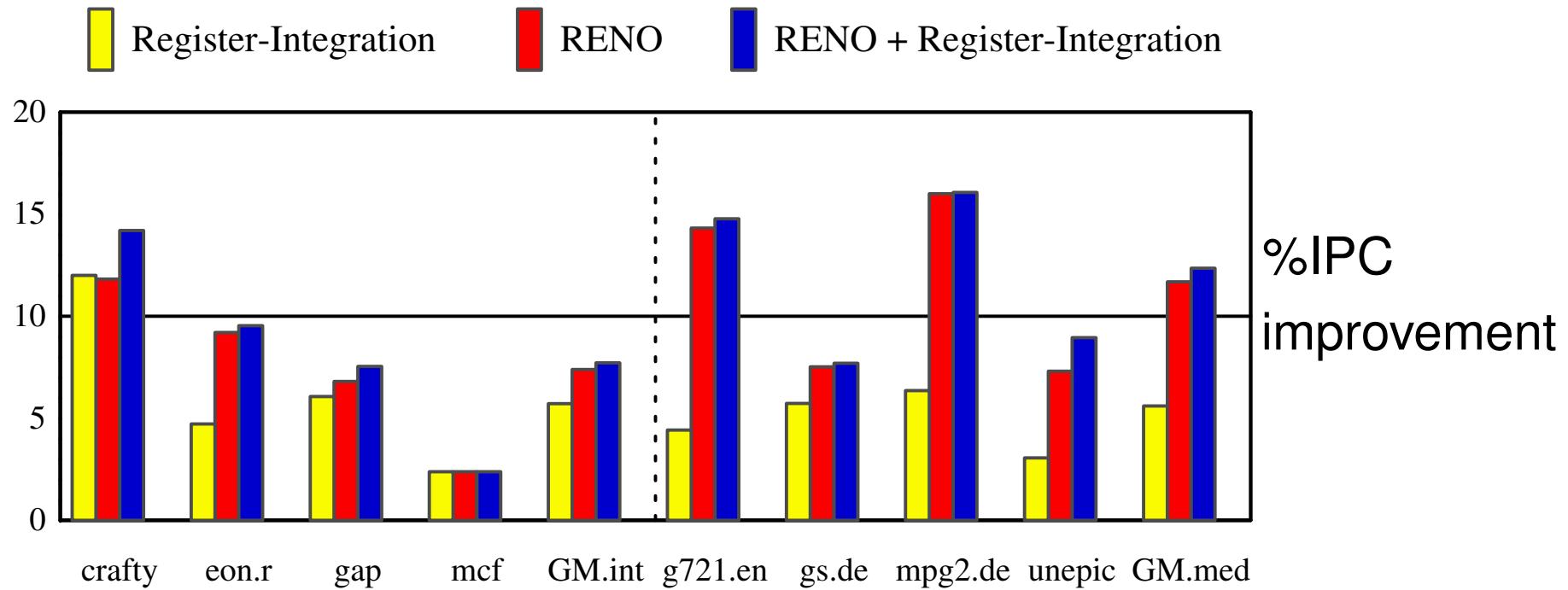
RENO_{CSE/RA} / RENO_{CF} Synergy



RENO+Register-Integration: CSE for all + folding

- Marginally better performance
 - Not worth larger tables, more accesses

RENO_{CSE/RA} / RENO_{CF} Synergy



RENO sweetspot

folding much more important than non-load CSE

RENO Summary

Concept

- Dynamic analogs of compiler optimizations

Implementation

- Unifies several previously proposed techniques
- Adds constant folding
- + Simple design

Effectiveness

- + Eliminates 23% of instructions from optimized programs

RENO and CO [Fahs+]

Similar ideas, different approaches

- Implementation complexity / optimization coverage tradeoff

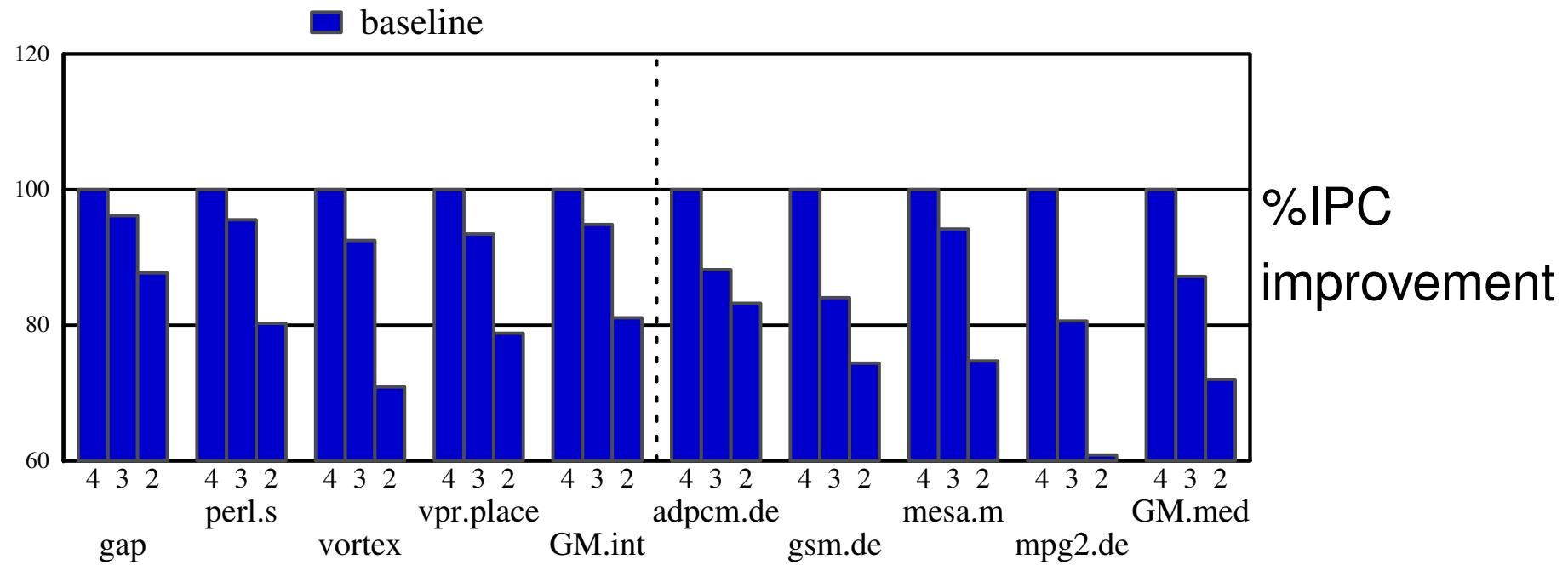
RENO: purely register-name based

- Modified R10K renamer (adds immediates only)
- 3-input functional units
- + Simpler implementation

CO (our understanding): register-name+**value** based

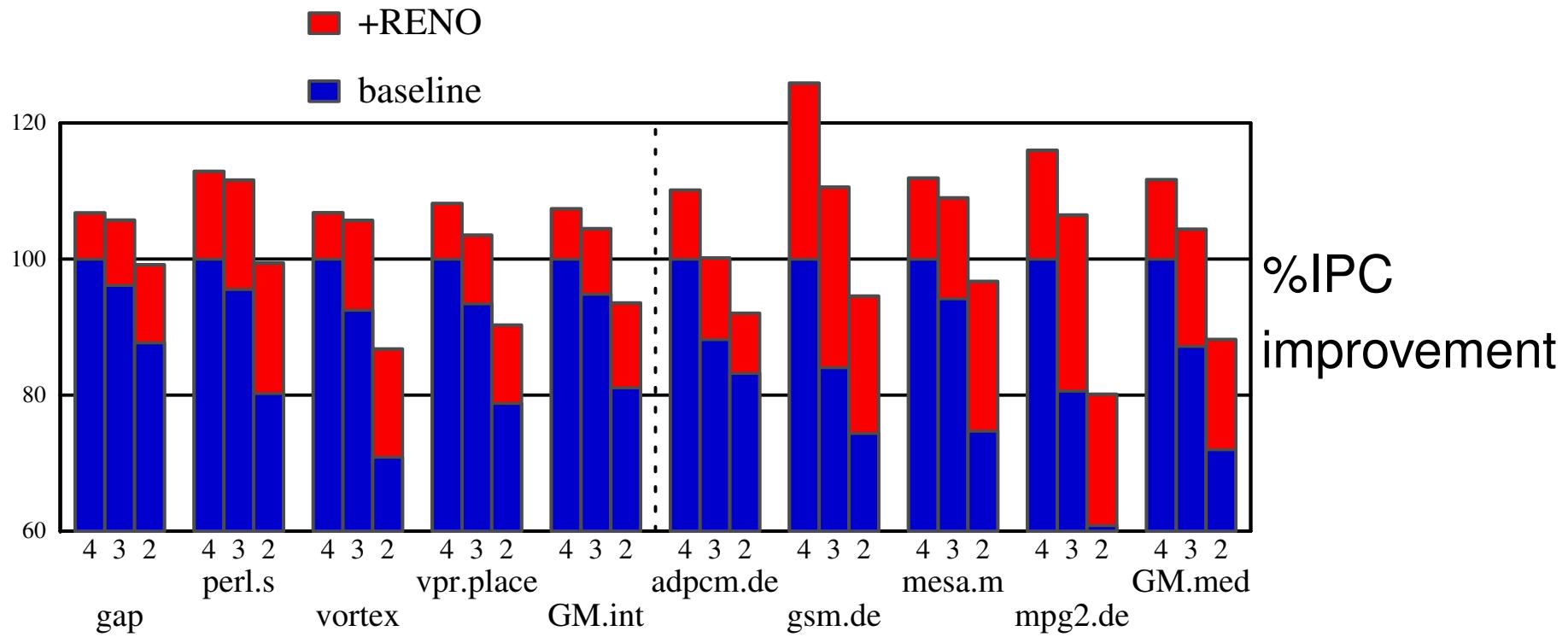
- Short in-order execution pipeline inside renaming
- Feedback paths from main core register file
- + Broader optimization scope

Core Bandwidth Reduction



Scaled down execution core: 3-wide and 2-wide

Core Bandwidth Reduction



Scaled down execution core: 3-wide and 2-wide

3-wide: small speedup (4%) vs. 4-wide

2-wide: performance within 6% for SPEC and 12% for MediaBench vs 4-wide