RENO: a RENname-based instruction Optimizer

Vlad Petric, Tingting Sha, Amir Roth

University of Pennsylvania

Penn
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	Dynamic optimizations (RENO)
– file/function boundaries	+ no boundaries

Why Optimize in Hardware?

Static optimizations	Dynamic optimizations (RENO)
– file/function boundaries	+ no boundaries
– conservative memory info	+ can speculate

Why Optimize in Hardware?

Static optimizations	Dynamic optimizations (RENO)
– file/function boundaries	+ no boundaries
– conservative memory info	+ can speculate
– correct on all paths	+ correct on dynamic path

Why Optimize in Hardware?

Static optimizations	Dynamic optimizations (RENO)
– file/function boundaries	+ no boundaries
– conservative memory info	+ can speculate
– correct on all paths	+ correct on dynamic path
+ removes insns at fetch	– removes insns at rename

Why Optimize in Hardware?

Static optimizations	Dynamic optimizations (RENO)
– file/function boundaries	+ no boundaries
– conservative memory info	+ can speculate
– correct on all paths	+ correct on dynamic path
+ removes insns at fetch	– 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

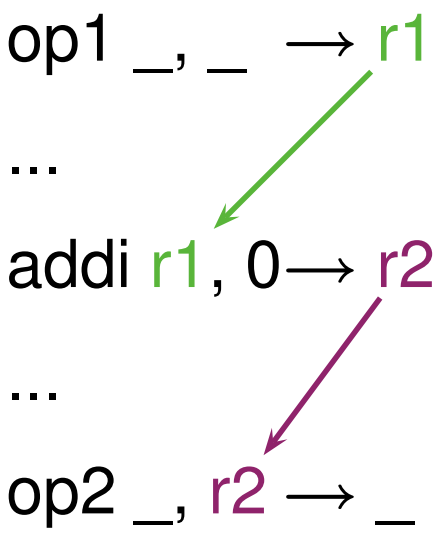
+ Amplifies execution core bandwidth

+ Map-table manipulations only

RENO optimization: way to detect sharing opportunities

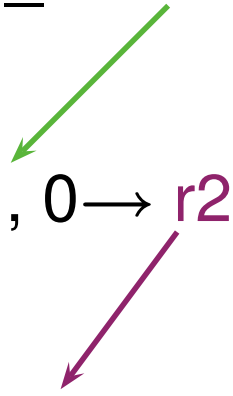
Conventional Processing

Static insn	MapTable action	Execution dataflow

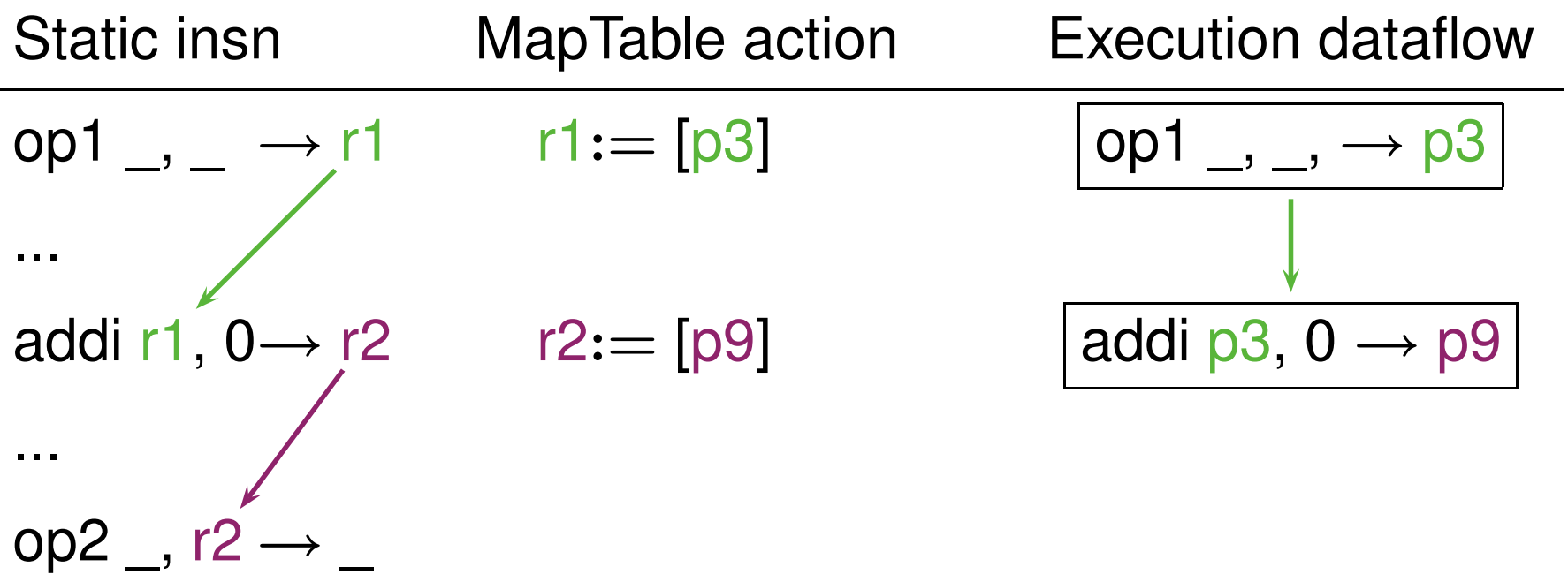


Conventional Processing

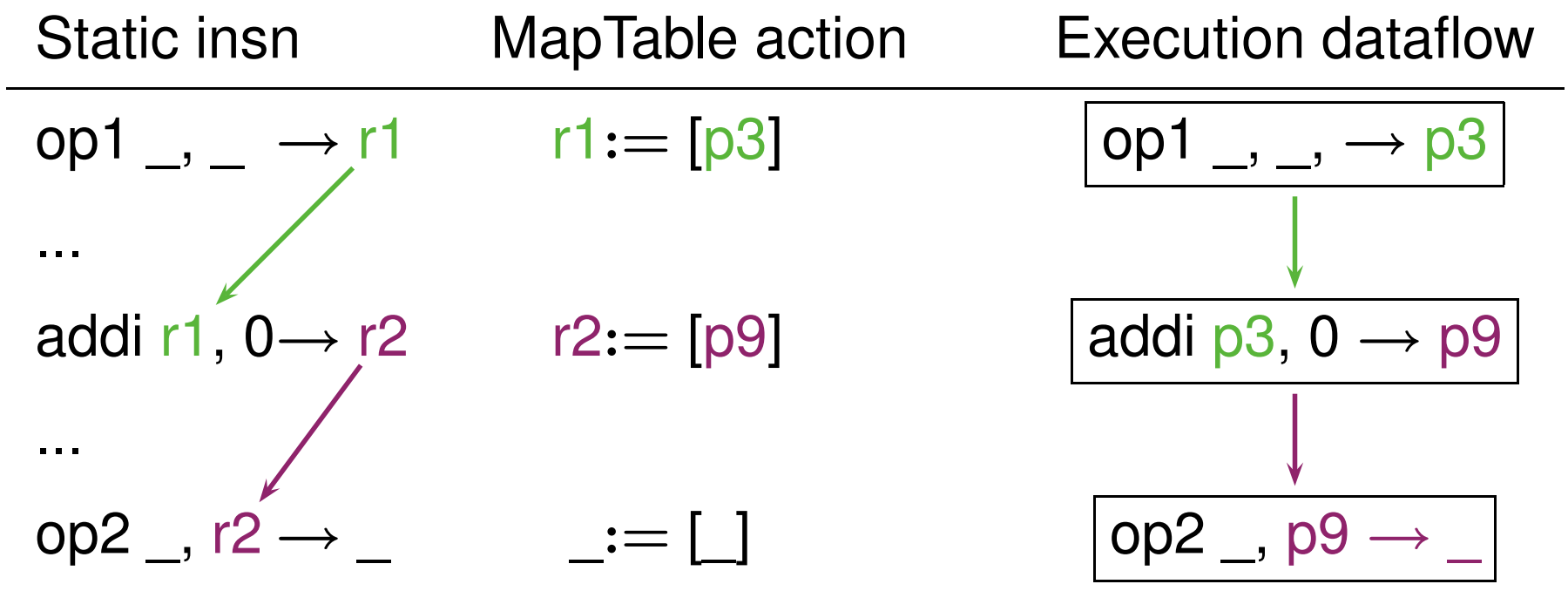
Static insn	MapTable action	Execution dataflow
op1 _, _ ⟶ r1	r1:= [p3]	op1 _, _, ⟶ p3
...		
addi r1, 0⟶ r2		
...		
op2 _, r2 ⟶ _		



Conventional Processing

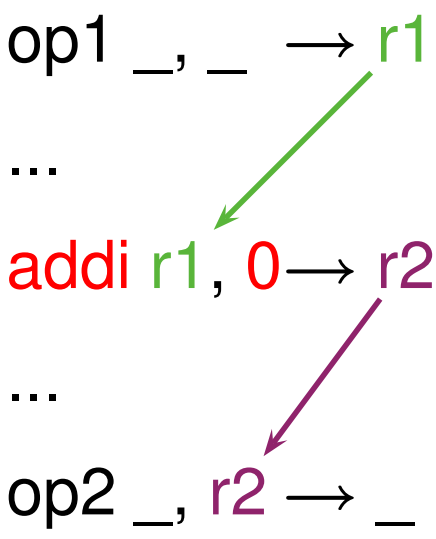


Conventional Processing



RENO$_{ME}$: Move Elimination

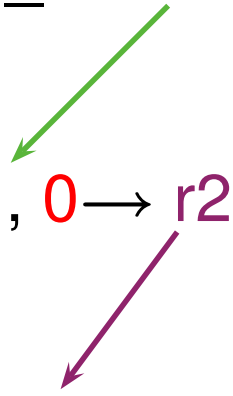
Static insn	MapTable action	Execution dataflow



Optimization detection: addi _, 0 → _

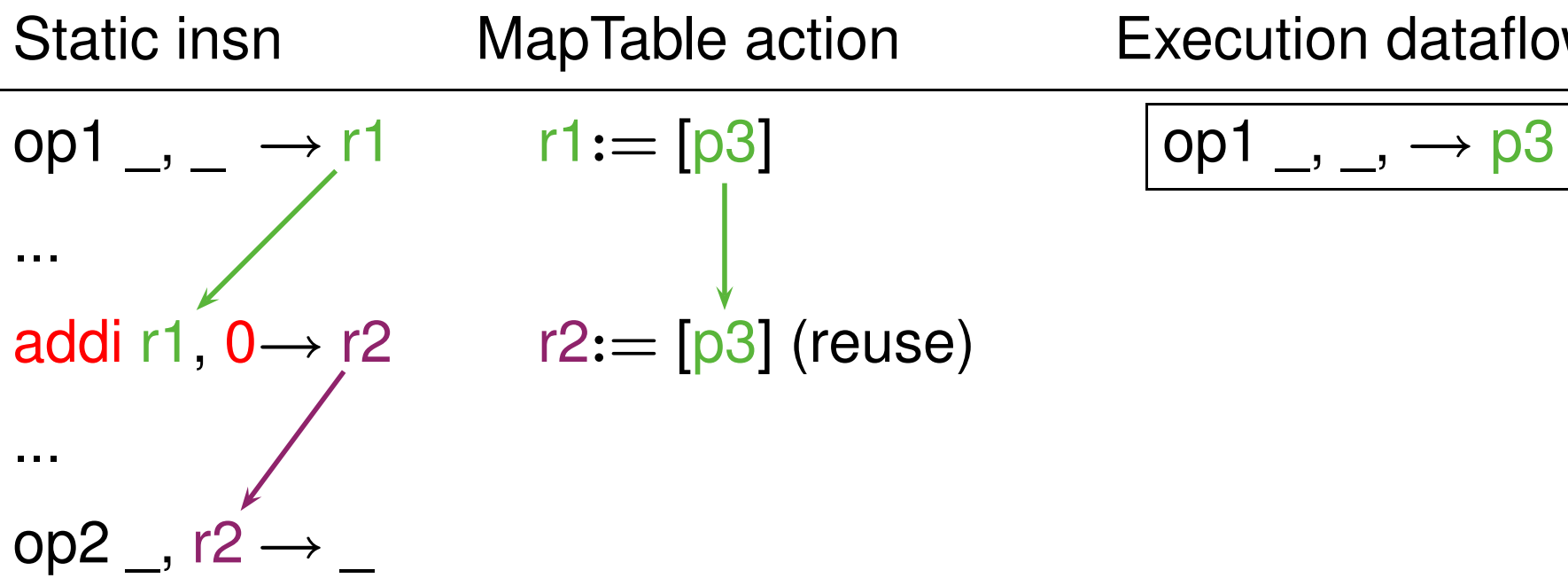
RENO$_{ME}$: Move Elimination

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



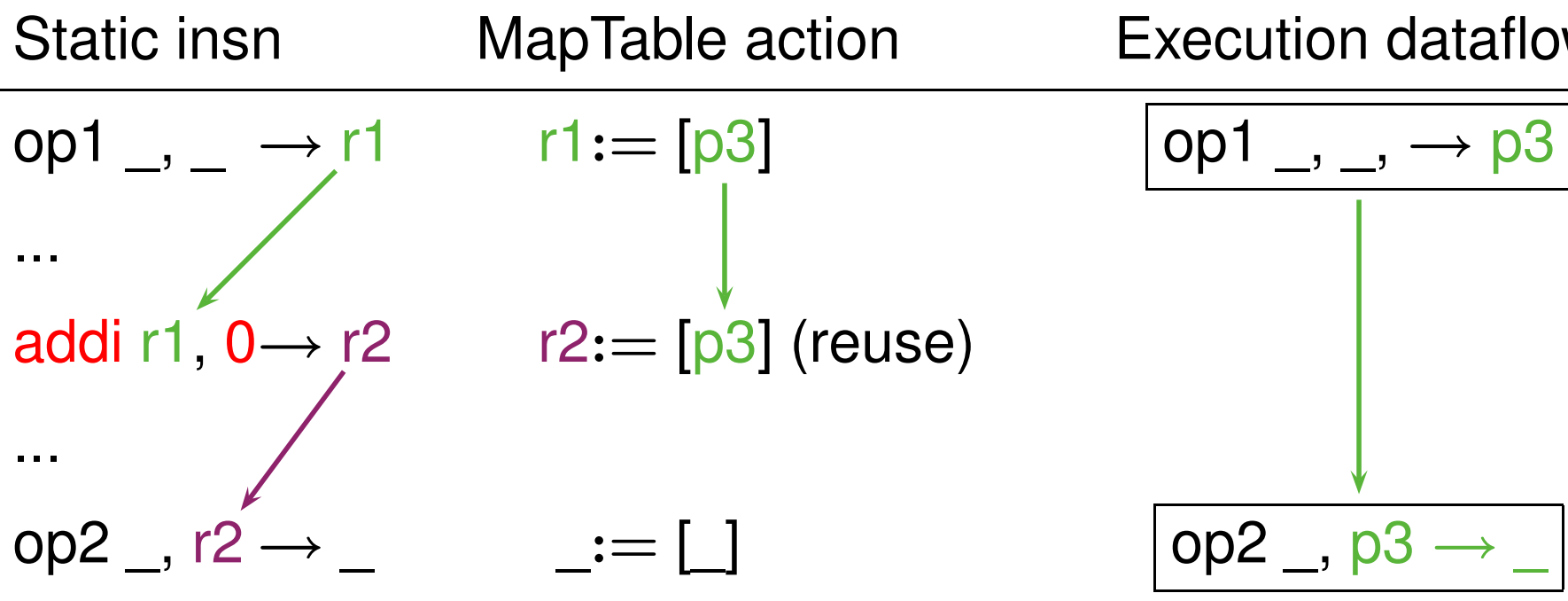
Optimization detection: addi _, 0 ⟶ _

RENO$_{ME}$: Move Elimination



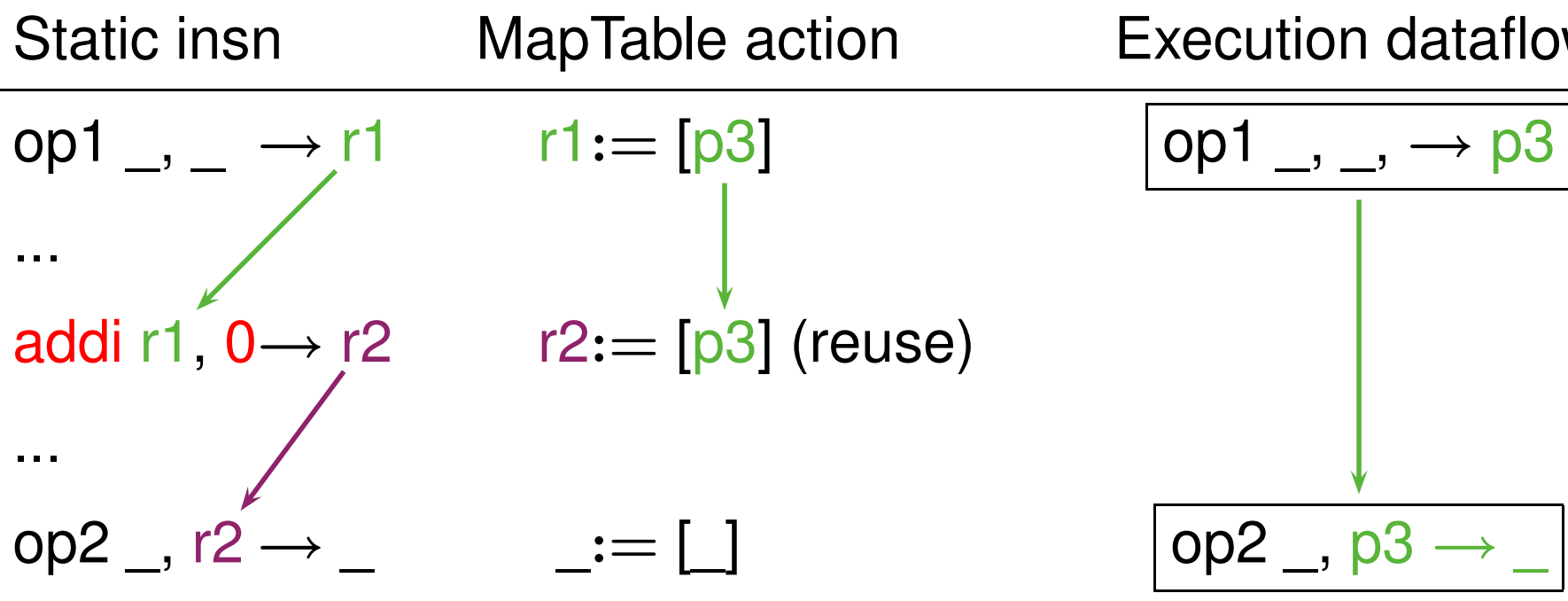
Optimization detection: addi _, 0 → _

RENO$_{ME}$: Move Elimination



Optimization detection: addi _, 0 → _

RENO$_{ME}$: Move Elimination

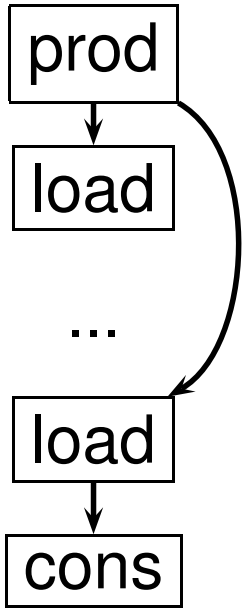


Optimization detection: addi _, 0 → _

\+ Latency: move removed from dataflow graph

\+ Bandwidth: move removed from execution core

RENO$_{CSE/RA}$: Register Integration

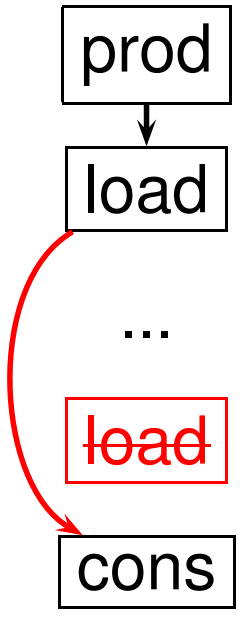


CSE/RegAlloc using physical register sharing

- 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 $\rightarrow$ _)

– Not common: compilation artifacts

RENO$_{CF}$: Constant Folding

RENO$_{ME}$: register moves (addi _, 0 $\longrightarrow$ _)

– Not common: compilation artifacts

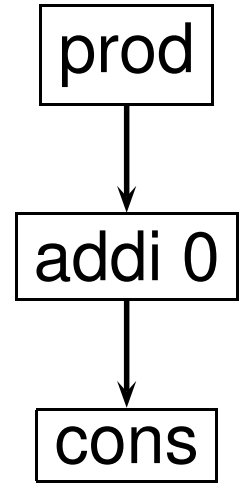
RENO$_{CF}$: register-immediate additions (addi _, 4 $\longrightarrow$ _)

+ 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$_{ME/CSE/RA}$

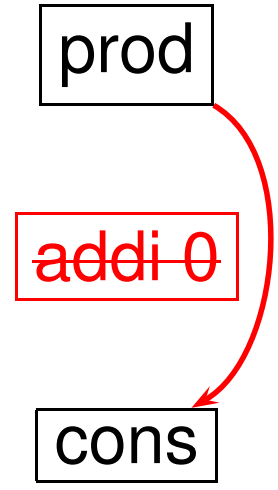
- Instructions don't compute "new" results
- Physical register sharing sufficient



Physical Register Sharing++

RENO$_{ME/CSE/RA}$

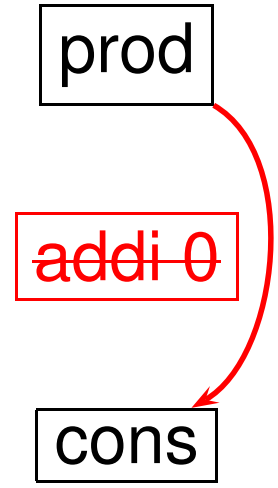
- Instructions don't compute "new" results
- Physical register sharing sufficient



Physical Register Sharing++

RENO$_{ME/CSE/RA}$

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



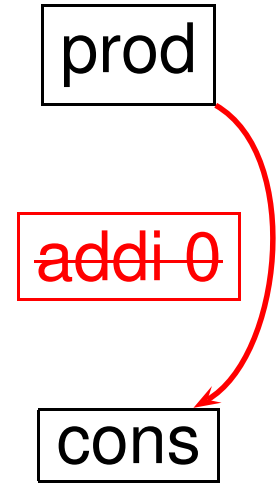
RENO$_{CF}$

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

Physical Register Sharing++

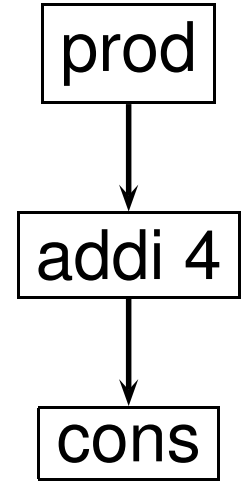
RENO$_{ME/CSE/RA}$

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



RENO$_{CF}$

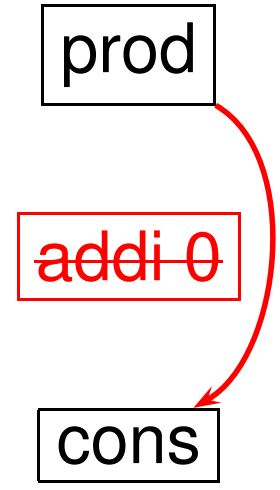
- Instructions compute "new" results
- Physical register sharing insufficient



Physical Register Sharing++

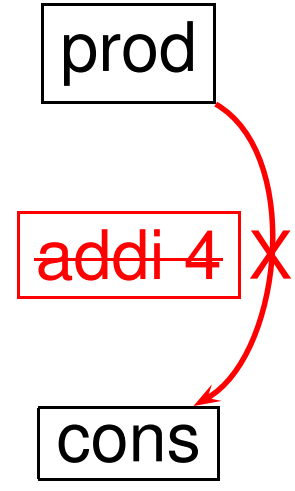
RENO$_{ME/CSE/RA}$

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



RENO$_{CF}$

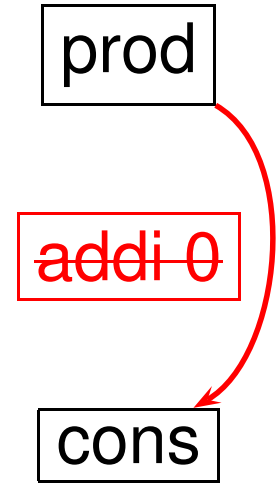
- Instructions compute "new" results
- Physical register sharing insufficient



Physical Register Sharing++

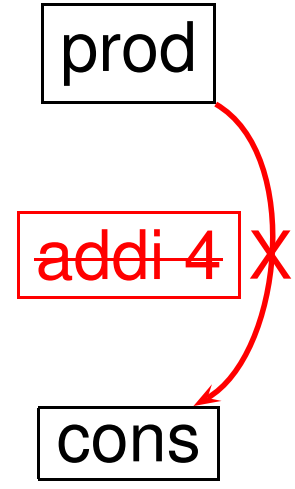
RENO$_{ME/CSE/RA}$

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



RENO$_{CF}$

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



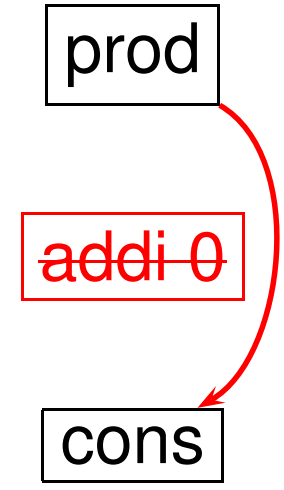
Solution: extended renaming semantics

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

Physical Register Sharing++

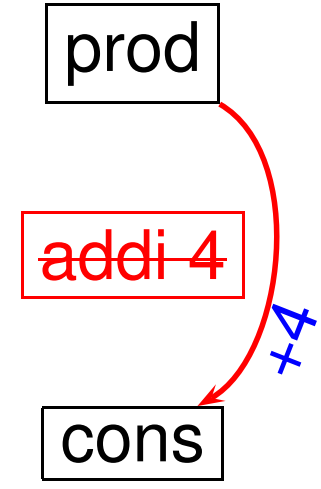
RENO$_{ME/CSE/RA}$

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



RENO$_{CF}$

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

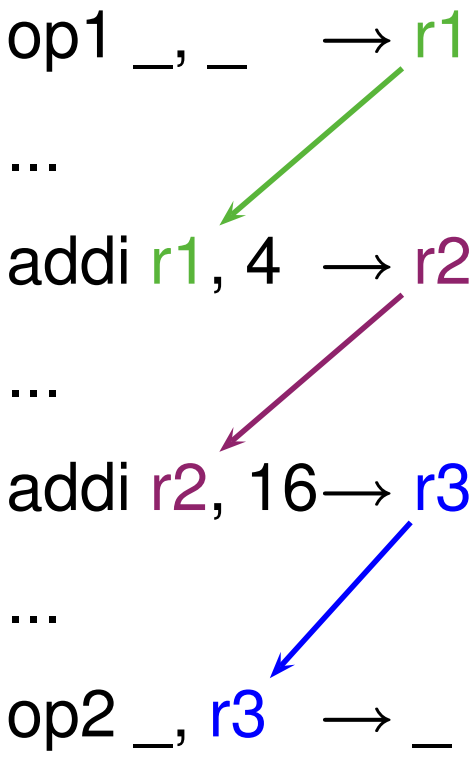


Solution: extended renaming semantics

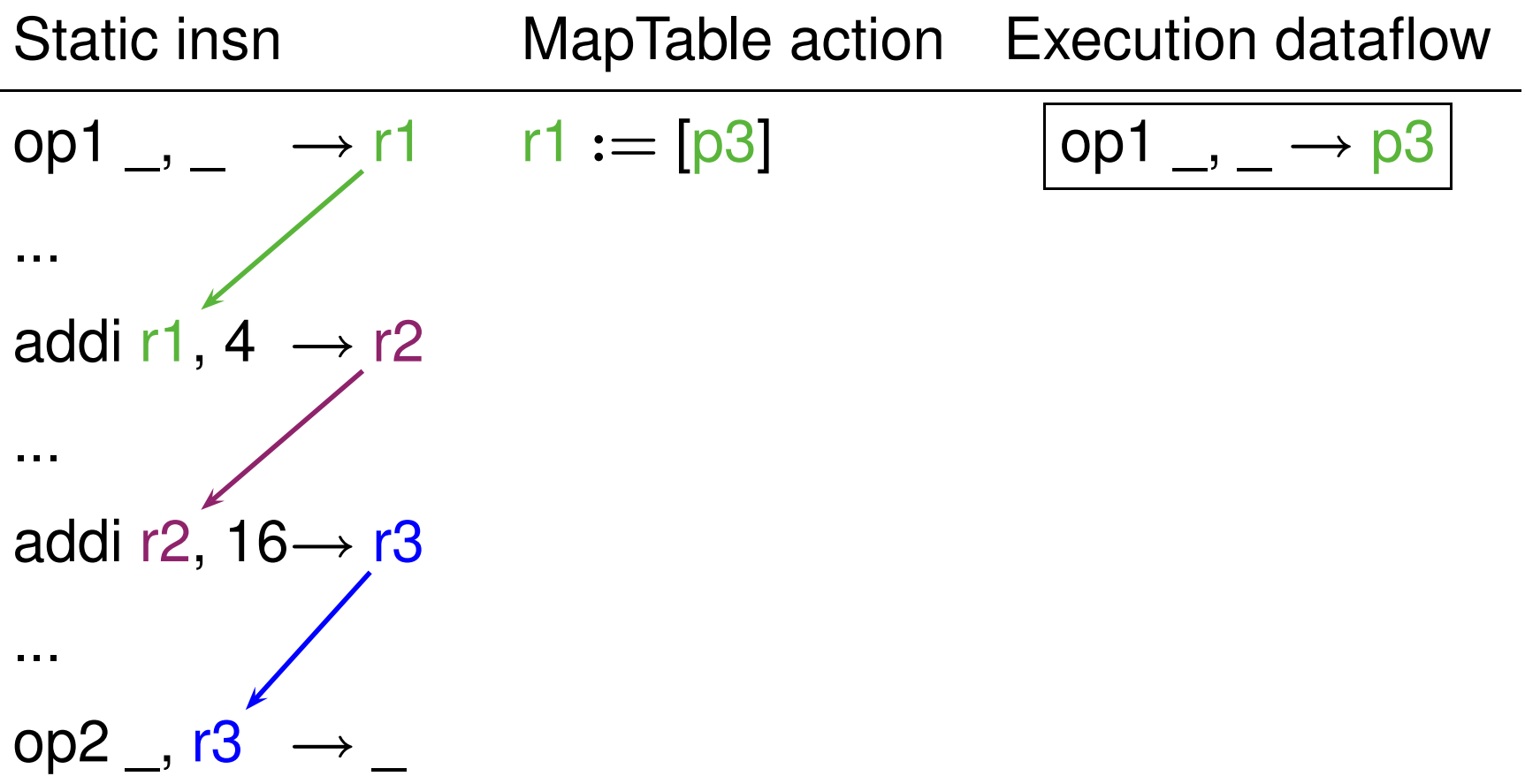
- Extended map table: LREG := [PREG, DISP]
- +DISP implicitly fused to any consumer

Conventional Processing

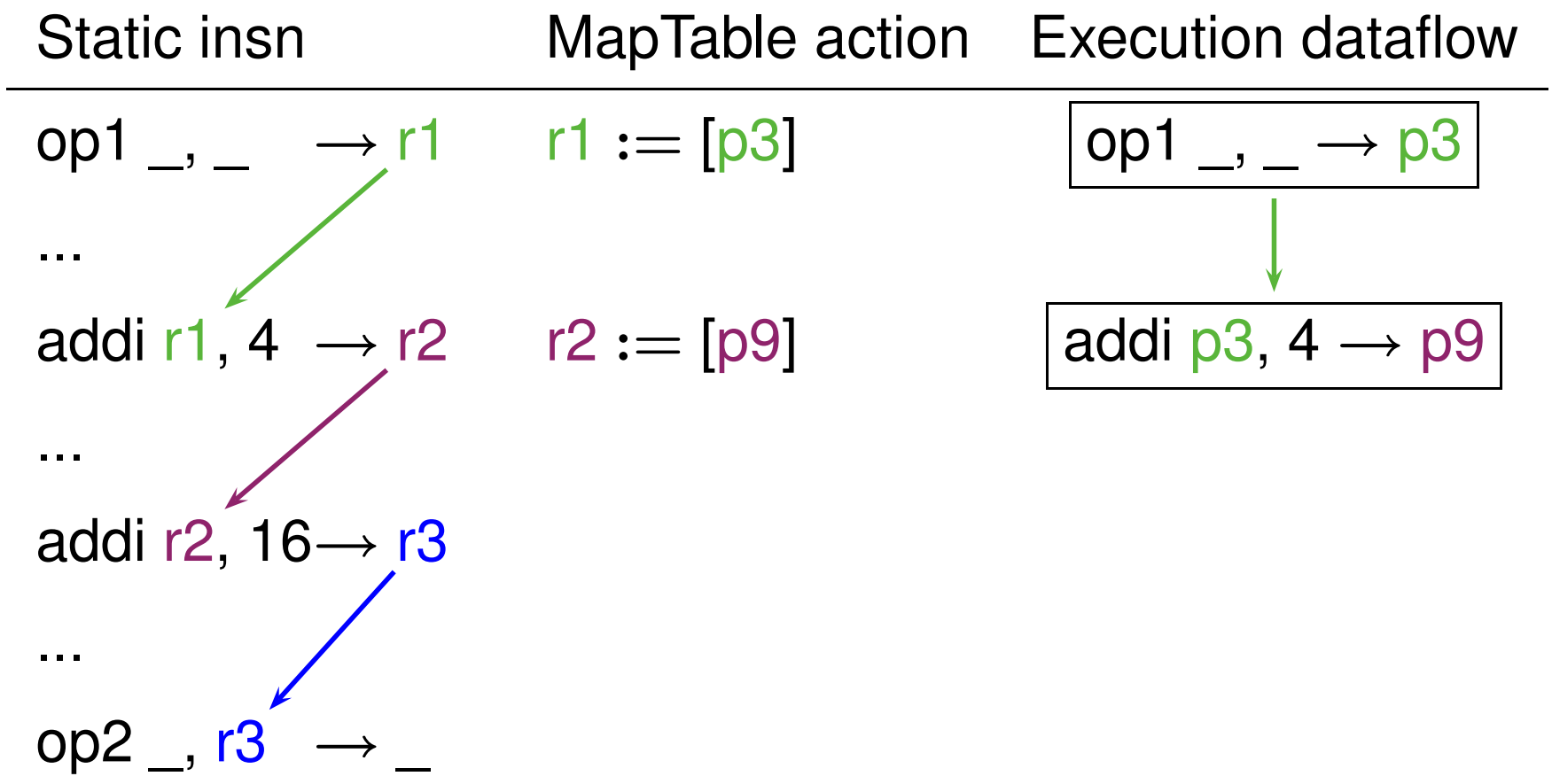
Static insn	MapTable action	Execution dataflow



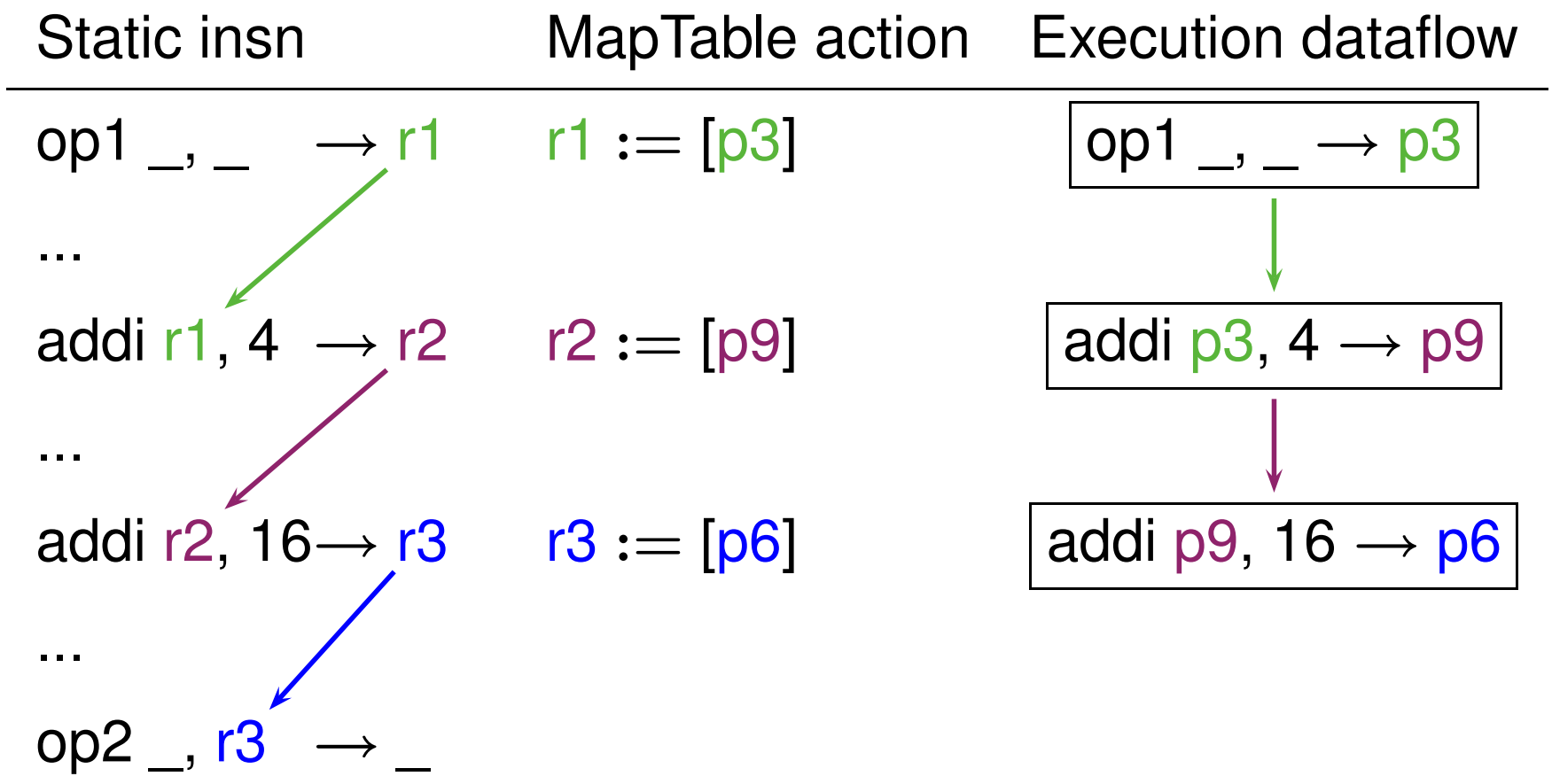
Conventional Processing



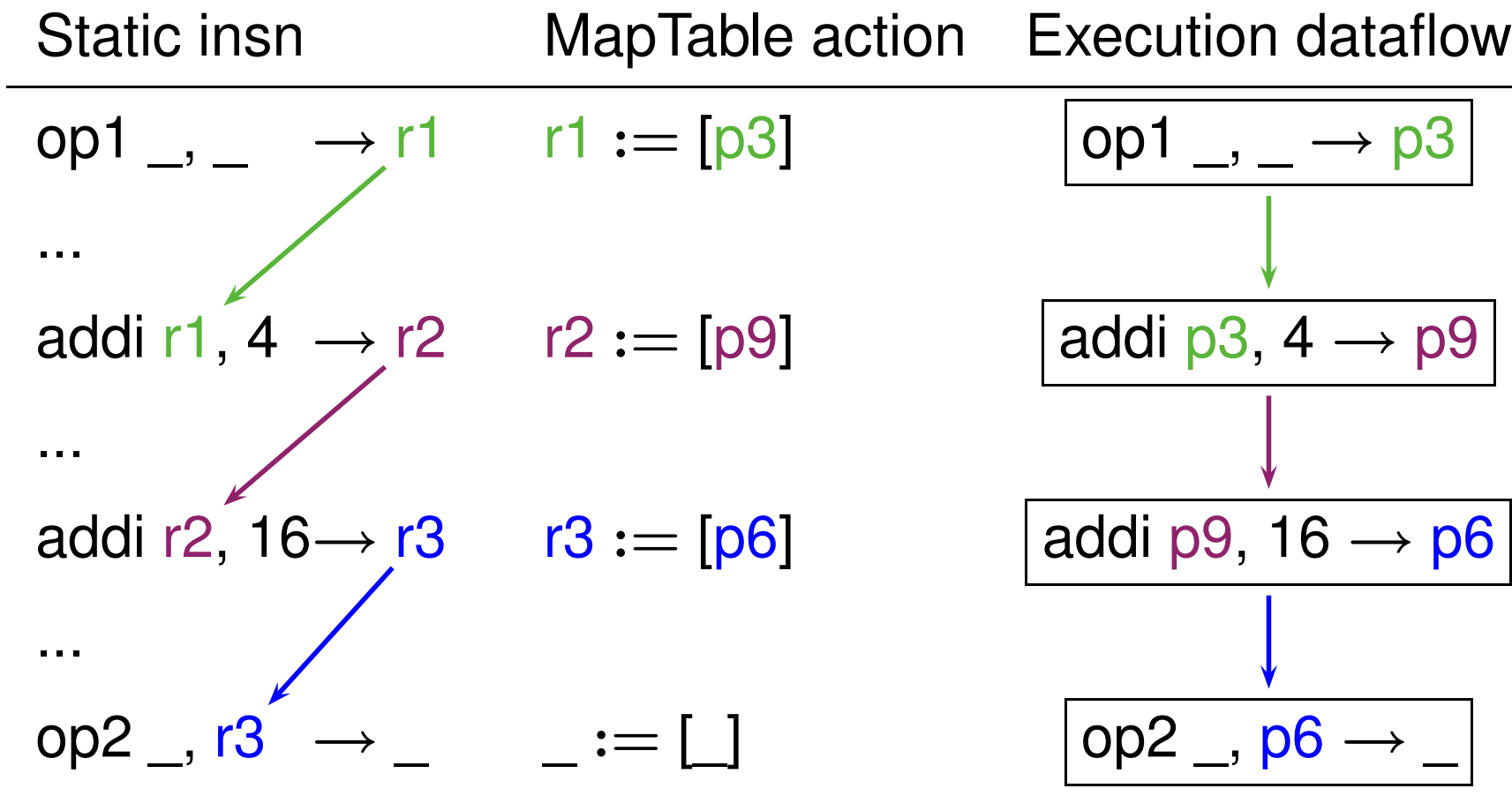
Conventional Processing



Conventional Processing

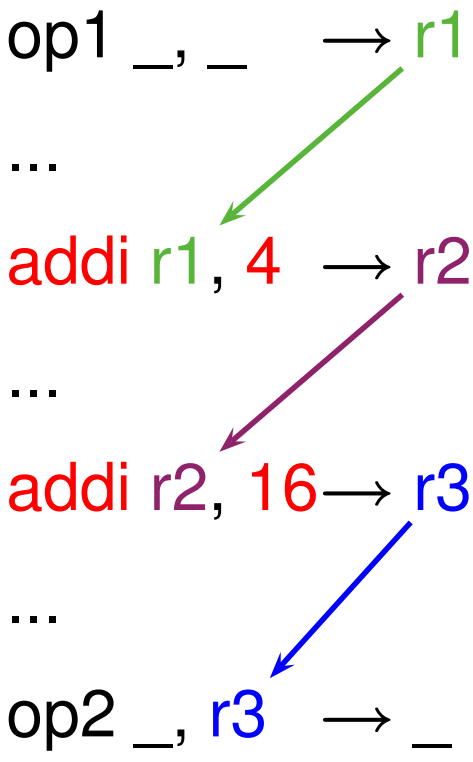


Conventional Processing

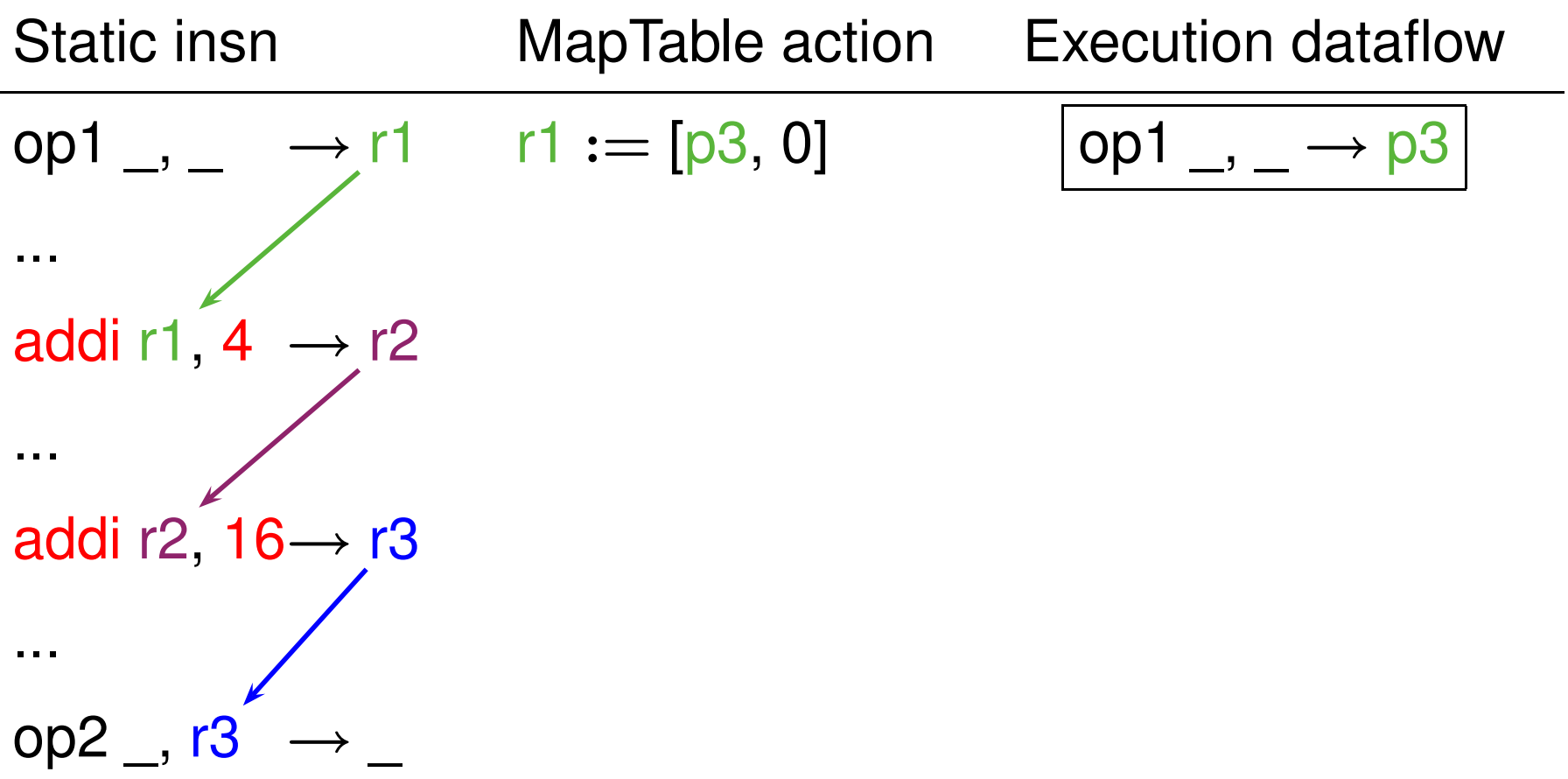


RENO$_{CF}$: Constant Folding

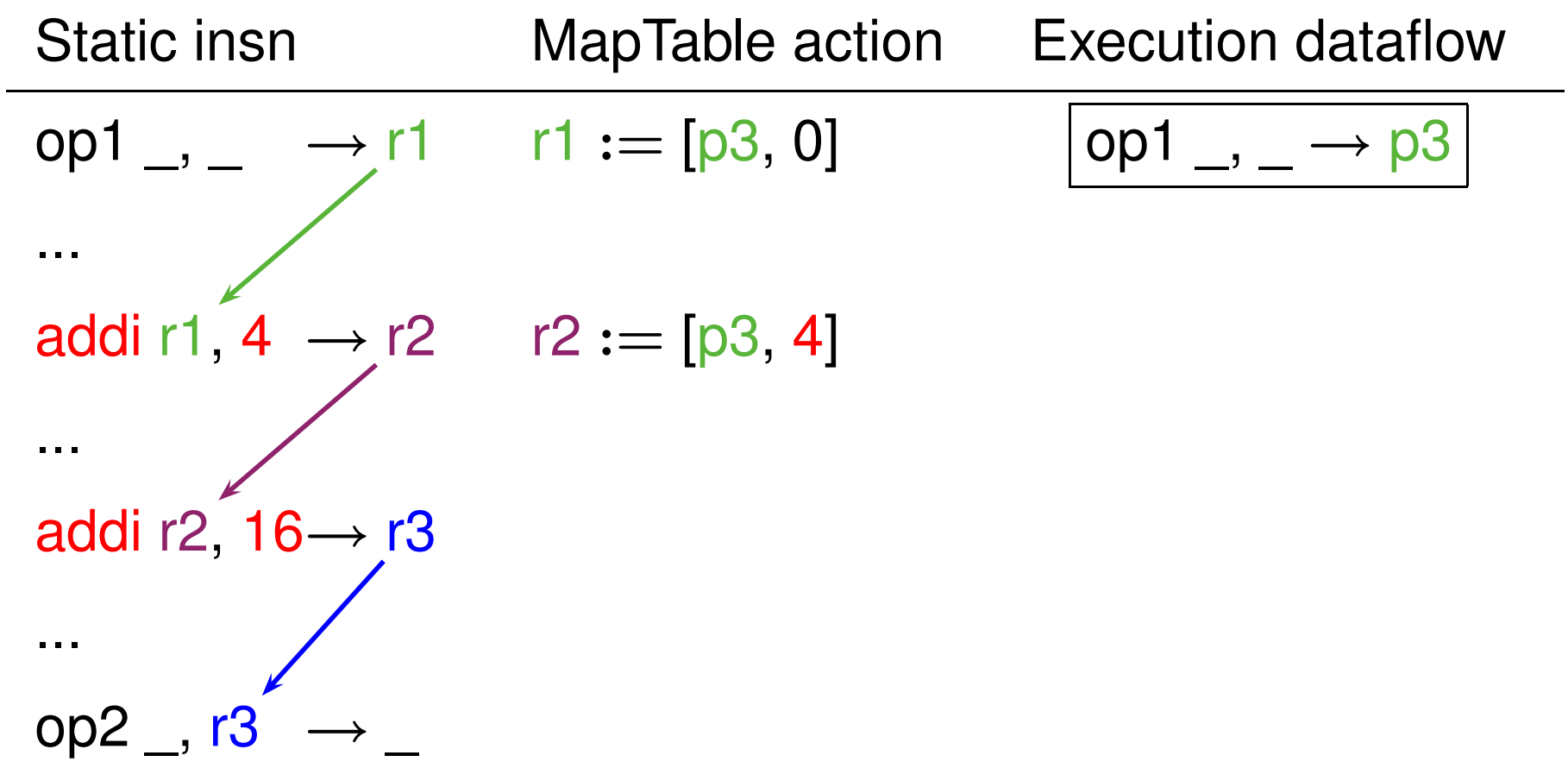
Static insn	MapTable action	Execution dataflow



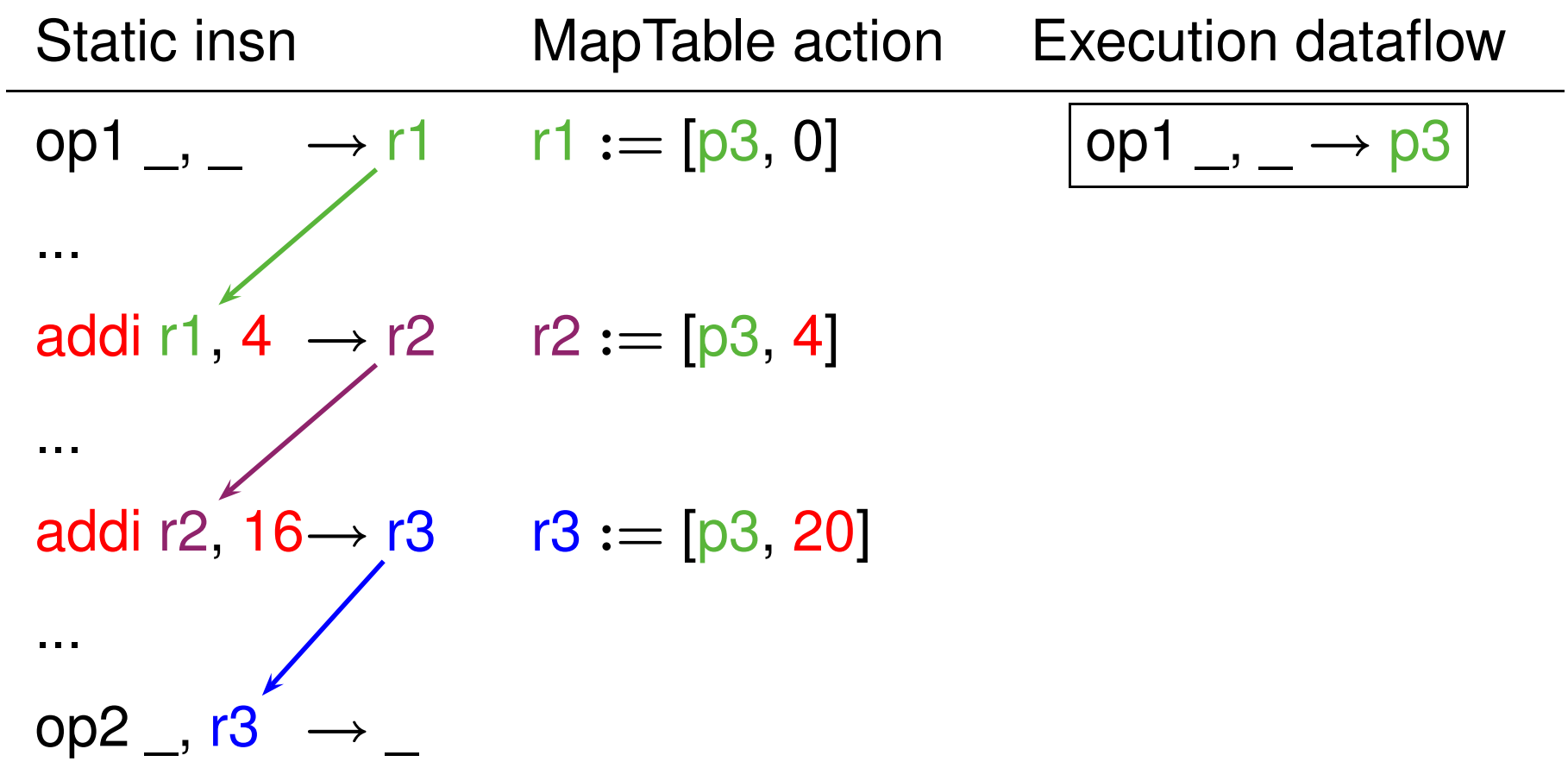
RENO$_{CF}$: Constant Folding



RENO$_{CF}$: Constant Folding

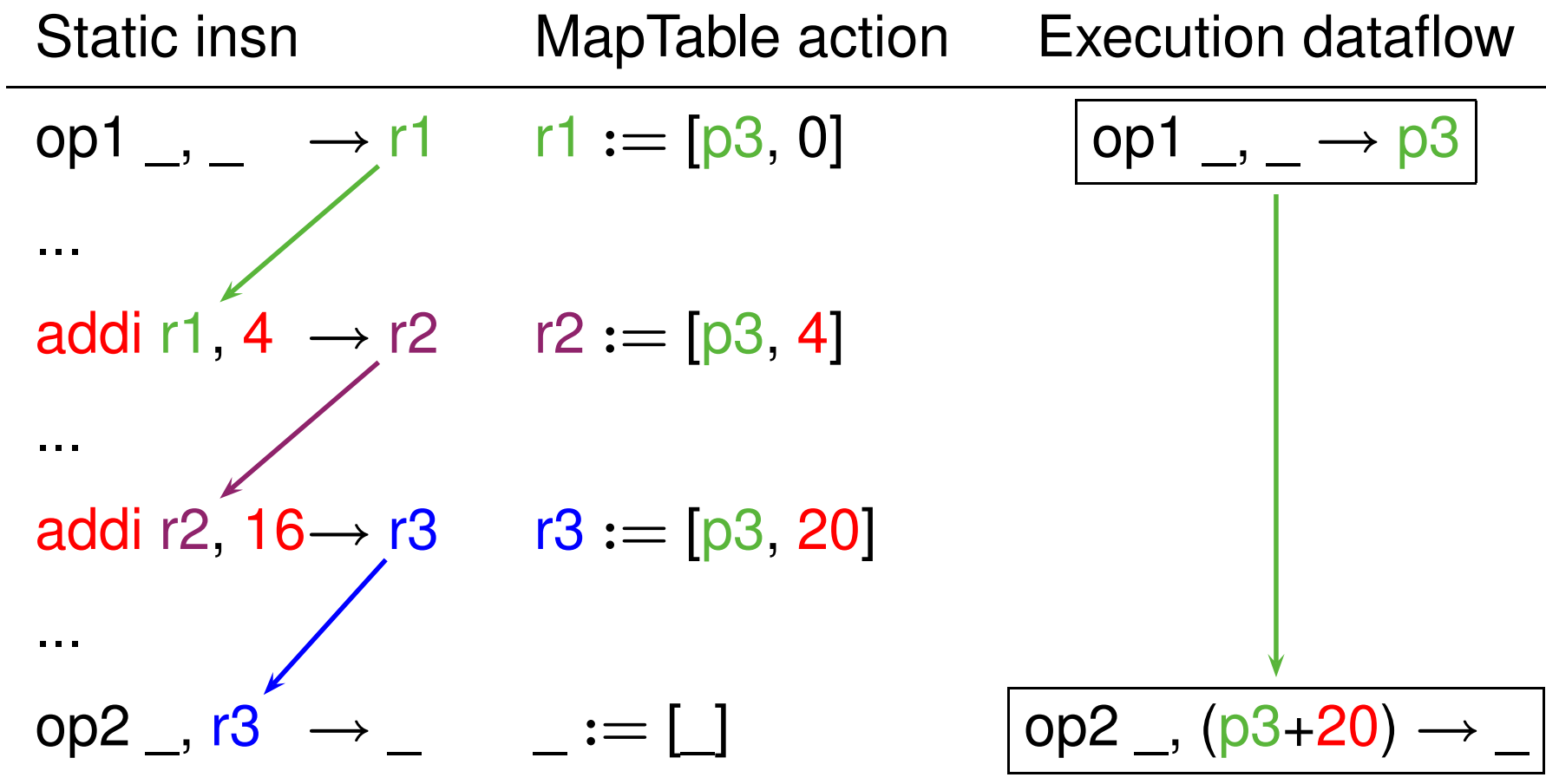


RENO$_{CF}$: Constant Folding



Renamer accumulates displacements

RENO$_{CF}$: Constant Folding



Renamer accumulates displacements

Consumer performs fused addi-op2 operation

RENO Implementation

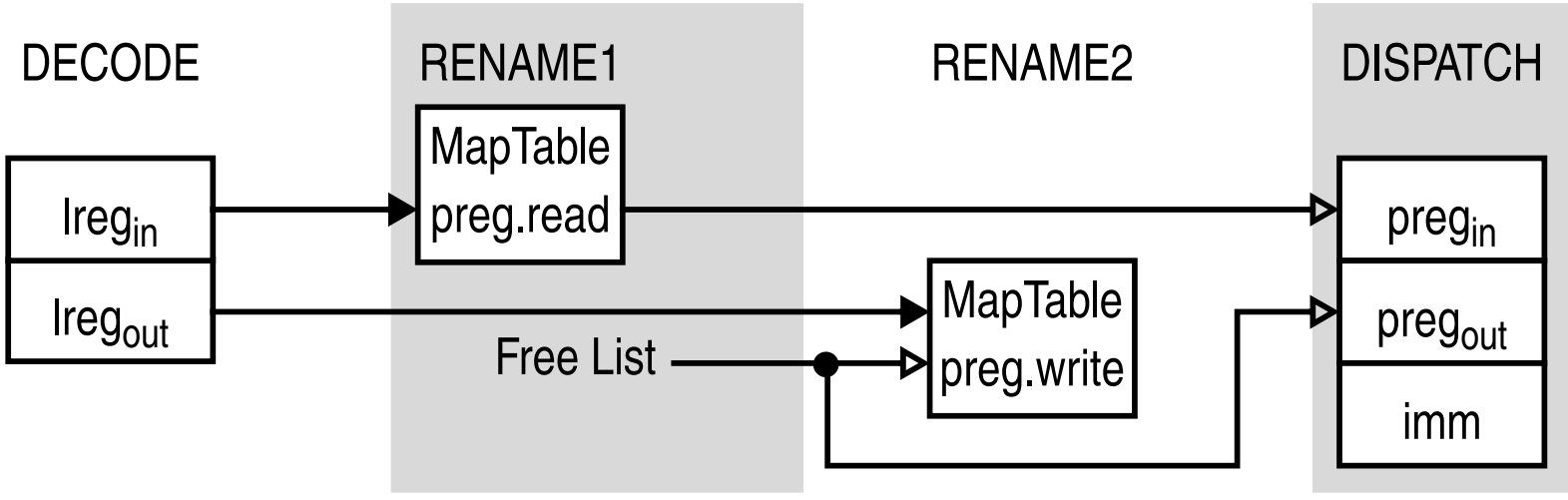
Rename

- 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

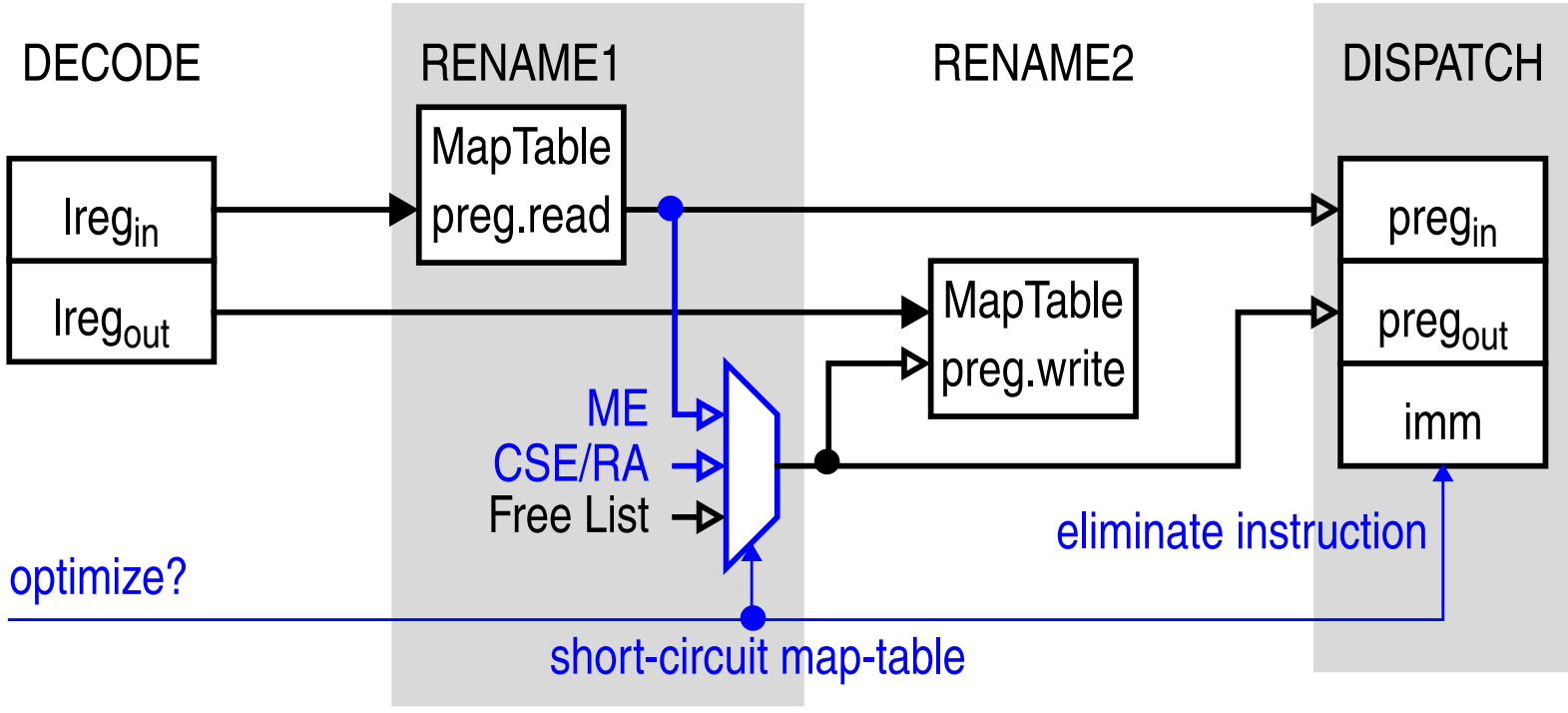
Conventional Renaming



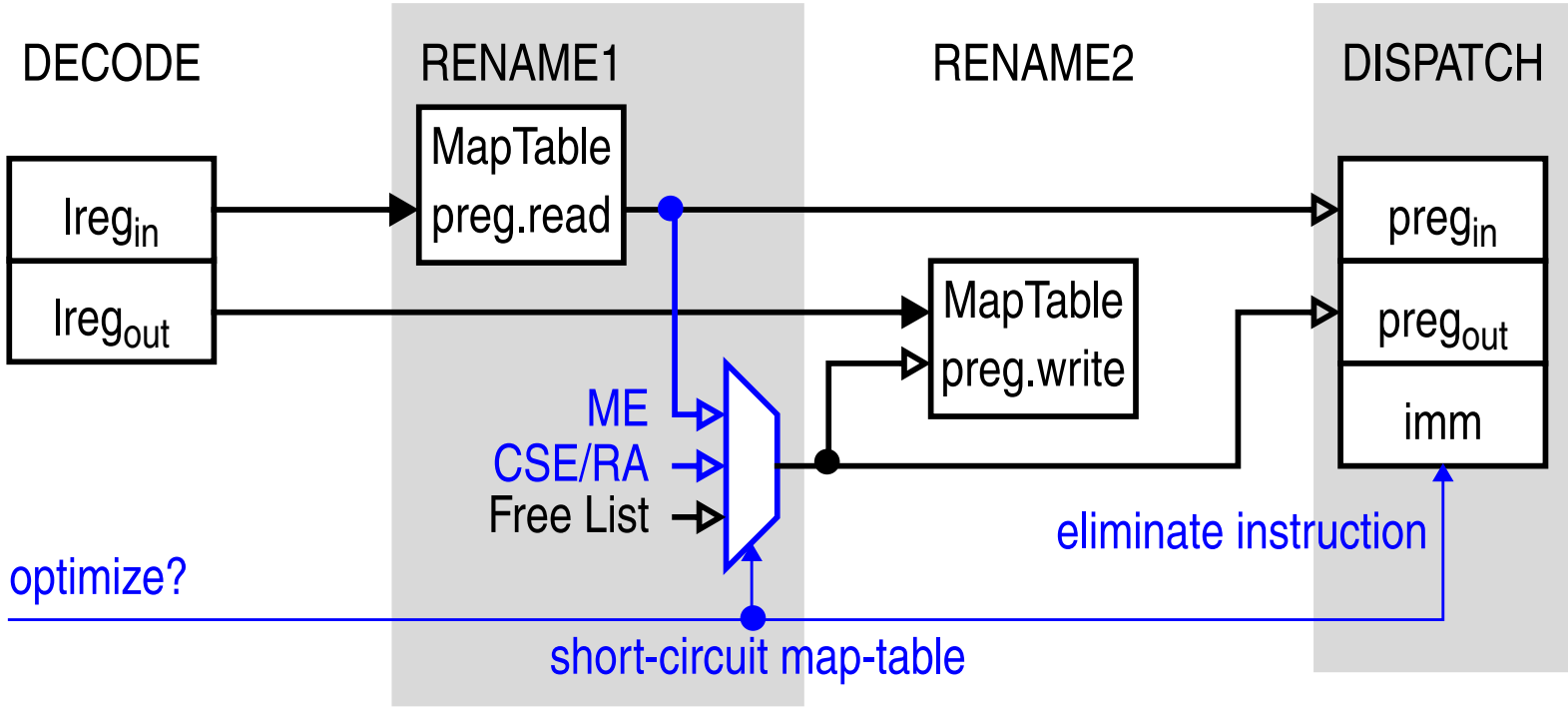
Example: scalar, one register input per instruction

- See paper for superscalar

RENO$_{ME/CSE/RA}$ Renaming



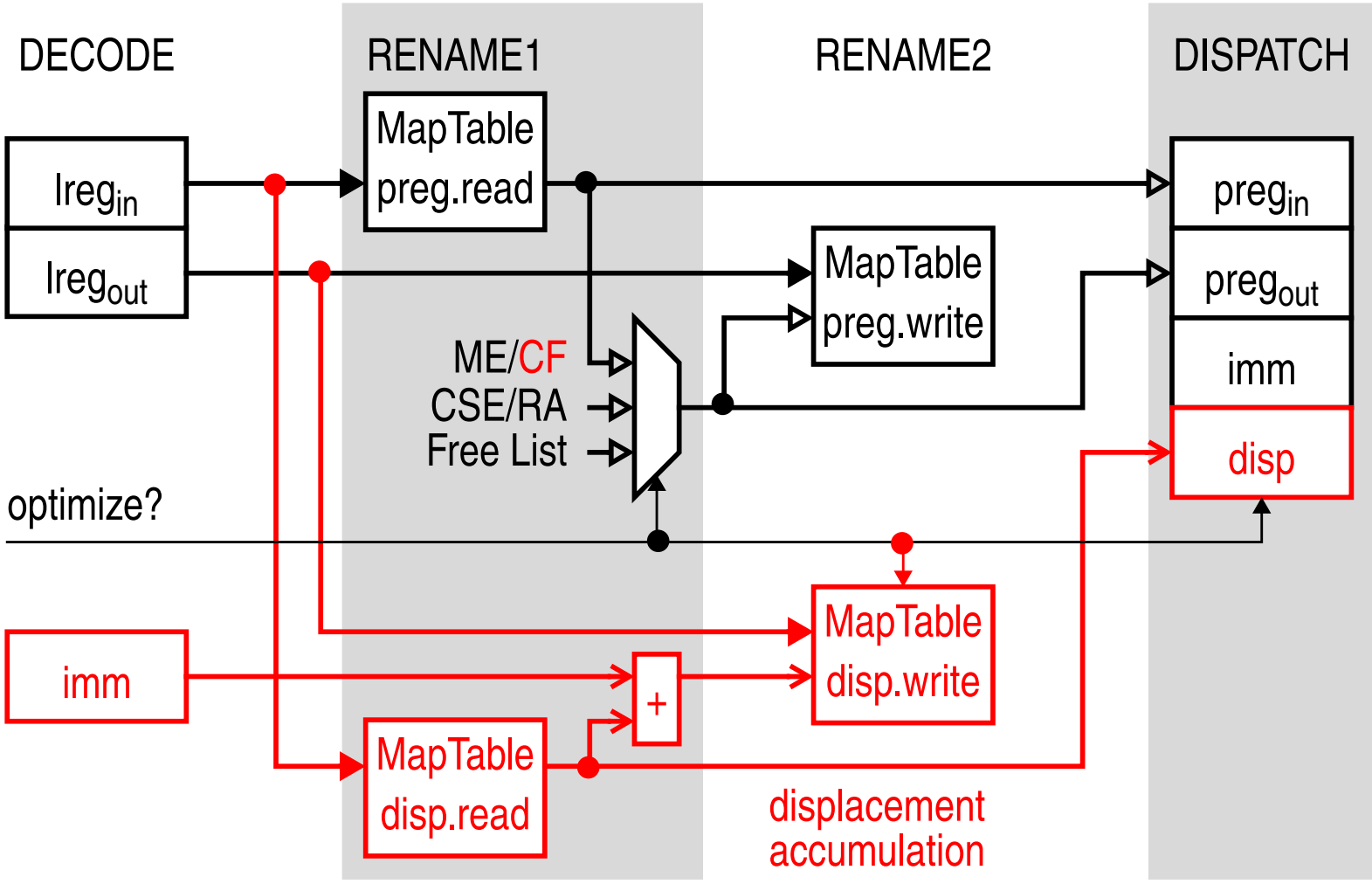
RENO$_{ME/CSE/RA}$ Renaming



Superscalar? see paper

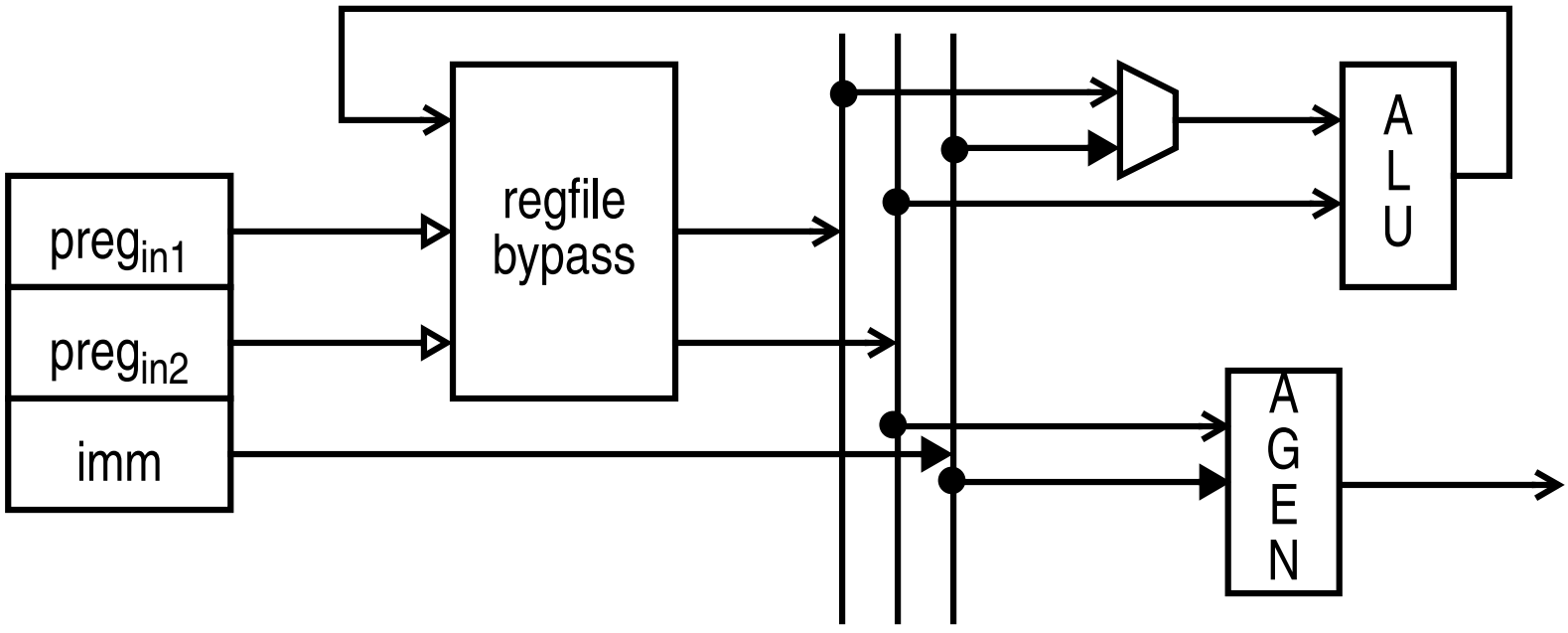
- O(N^2): same as conventional renaming
- Don't optimize dependent insns in same cycle
- We *believe* additional renaming stages are unnecessary

RENO$_{CF}$ Renaming

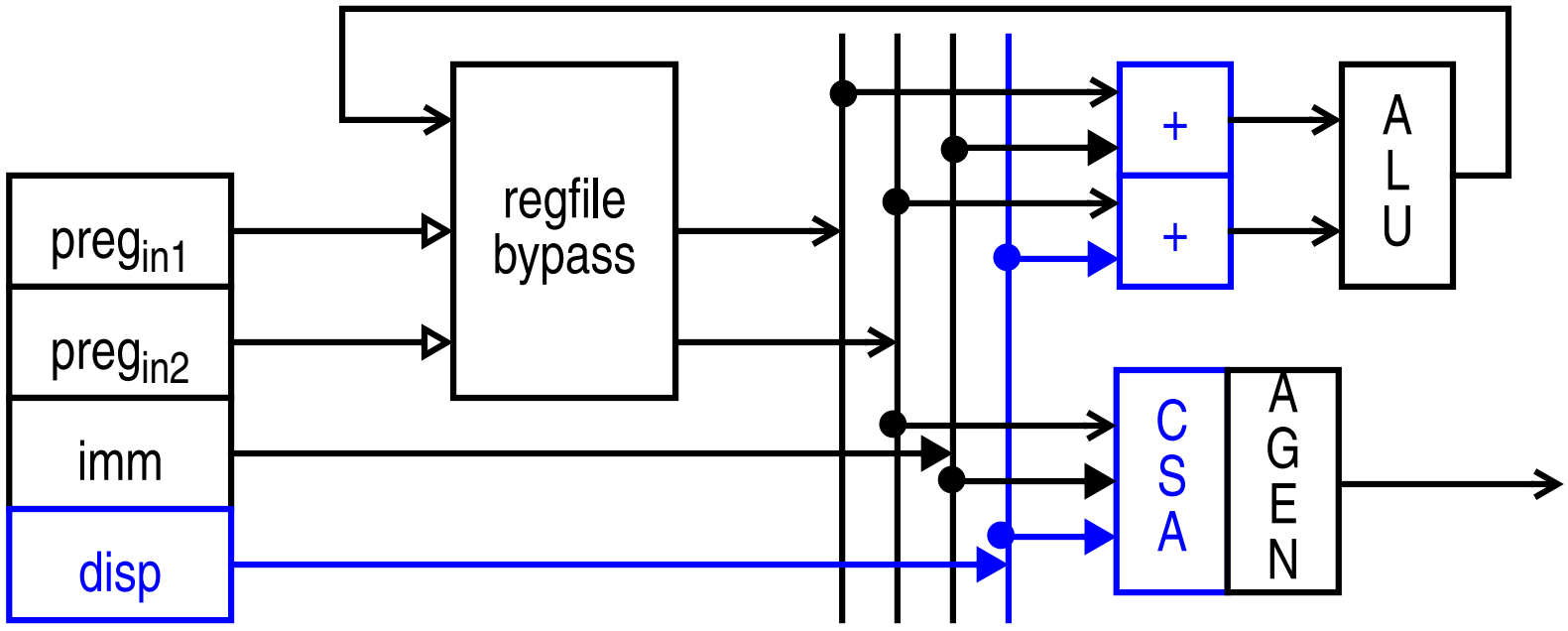


RENO$_{ME}$ + parallel displacement accumulation circuit

Conventional Execution



RENO$_{CF}$ Execution



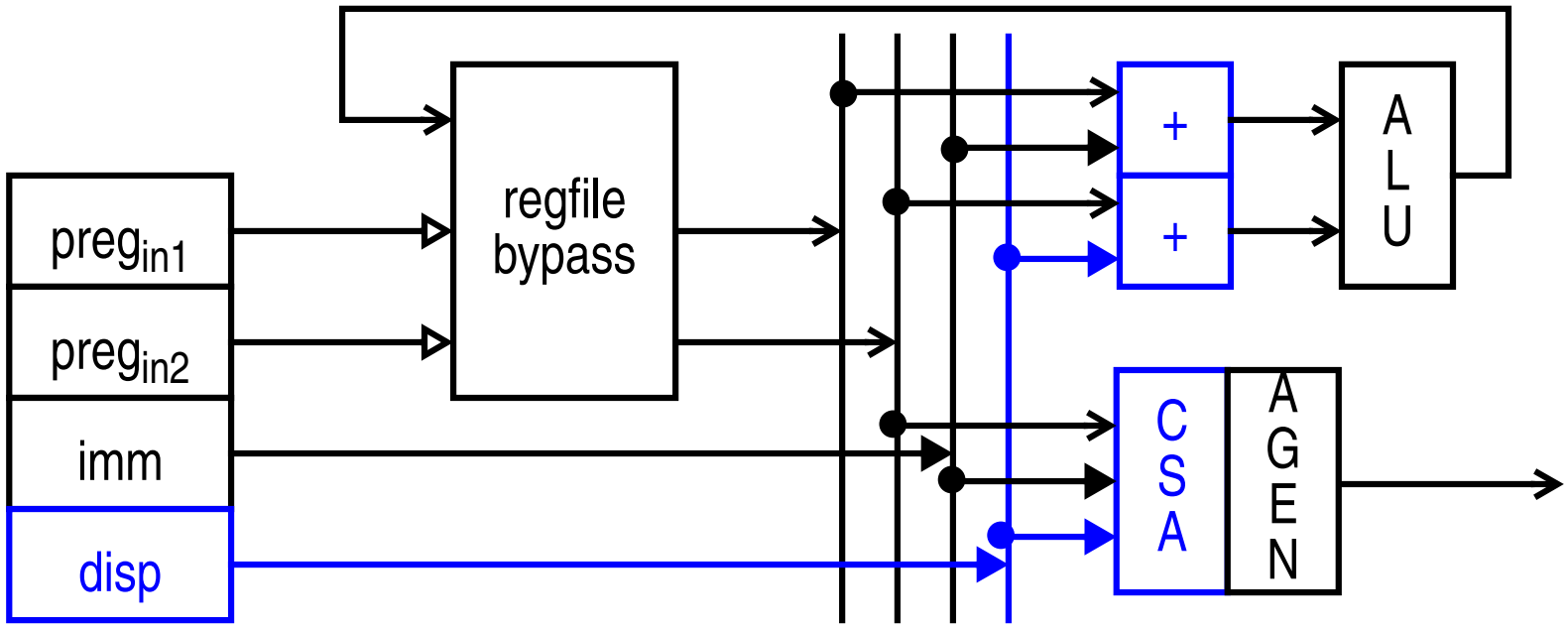
All operations take an additional (immediate) input

– 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.

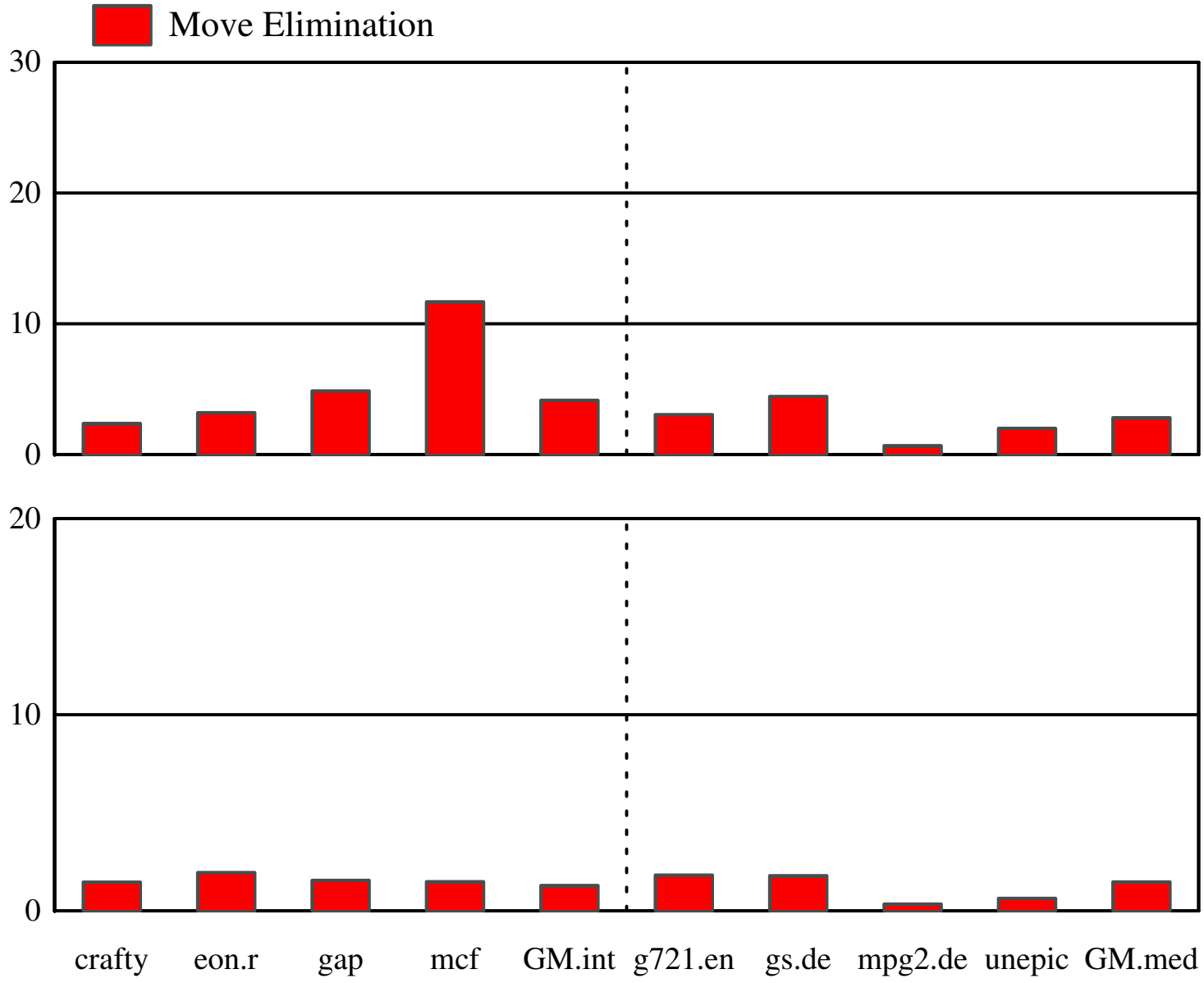
Benchmarks: SPECint2000, MediaBench

- 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



%Instructions eliminated

4%

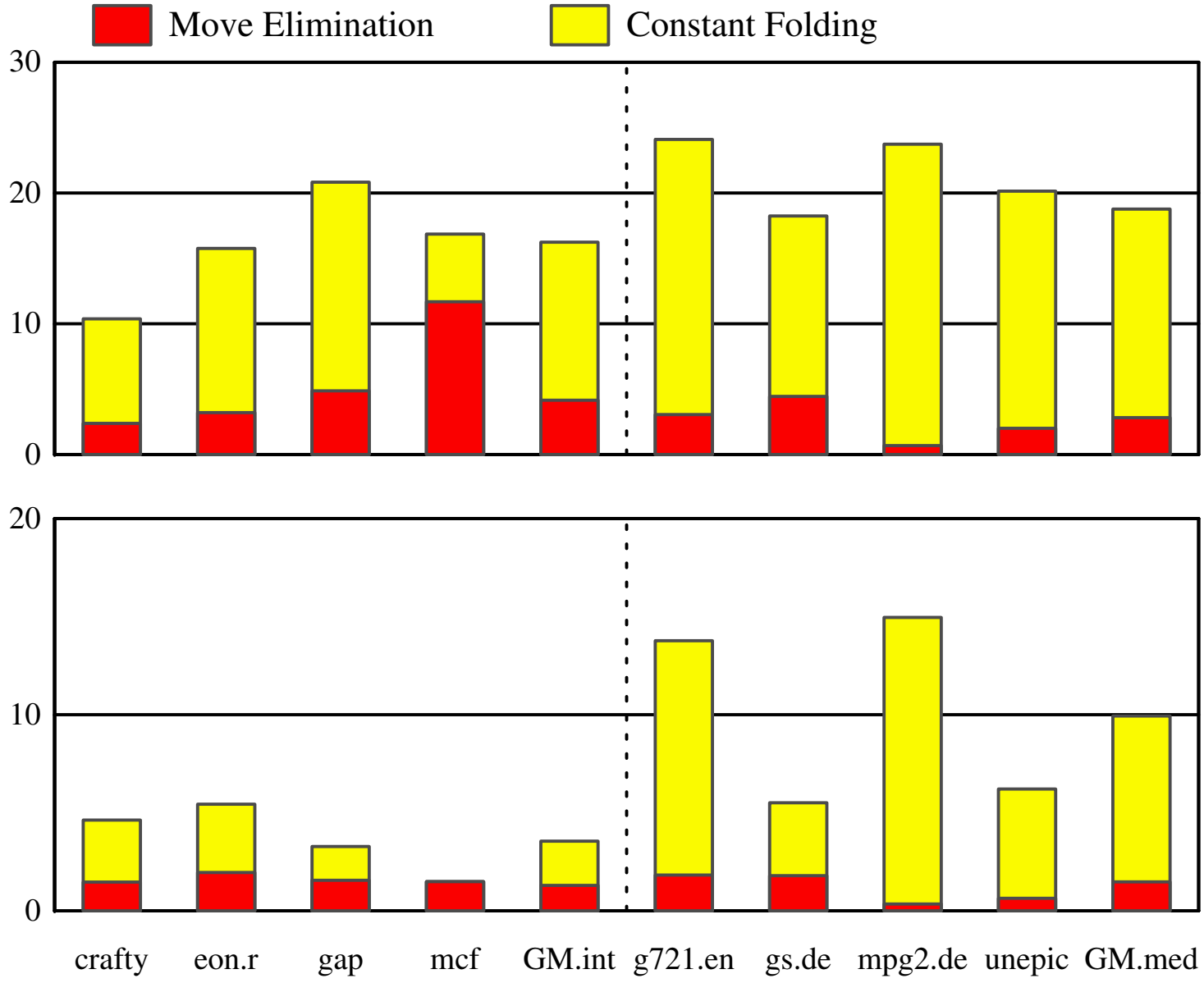
3%

%IPC improvement

1.3%

1.5%

Coverage and Performance



%Instructions eliminated

16%

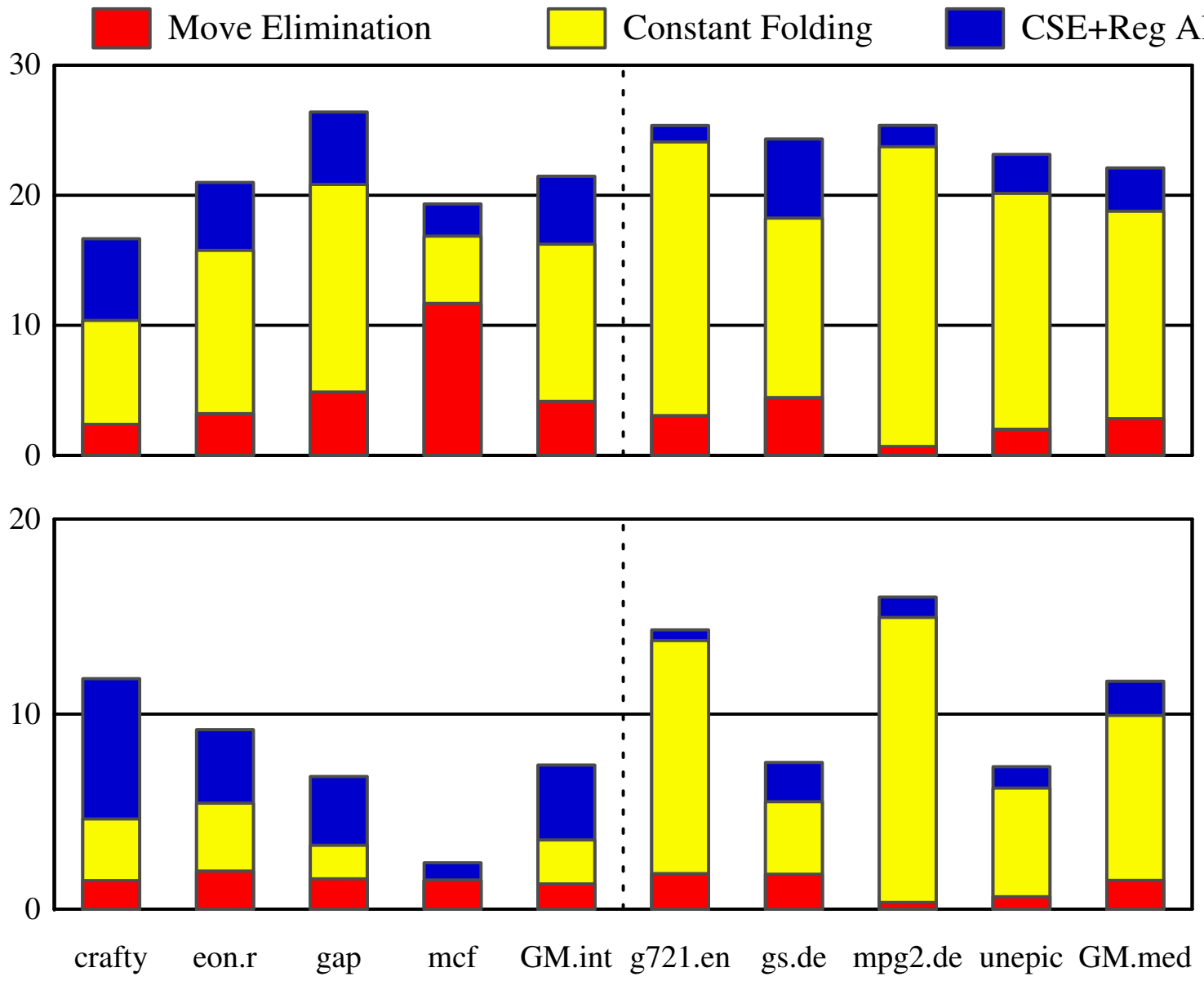
19%

%IPC improvement

4%

10%

Coverage and Performance



%Instructions eliminated

22%

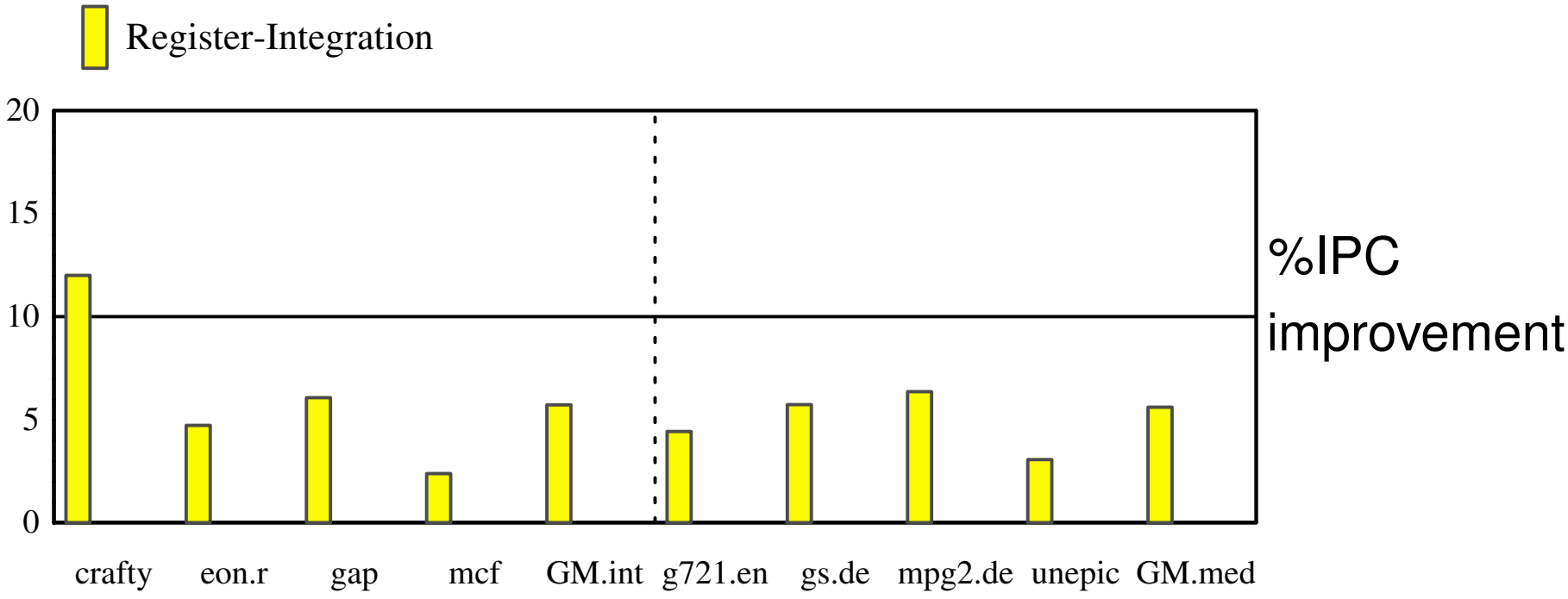
23%

%IPC improvement

8%

12%

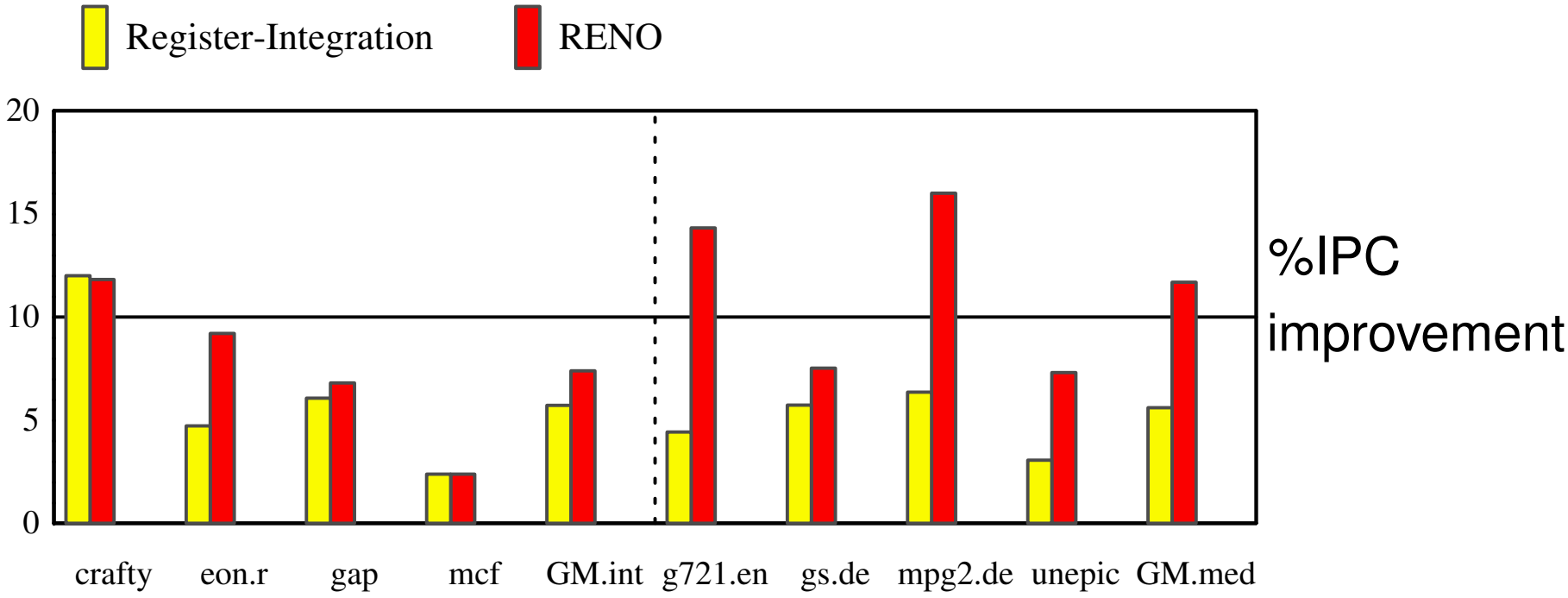
RENO$_{CSE/RA}$ / RENO$_{CF}$ Synergy



Register Integration: CSE for all, no folding

- Operations must be redundant
- Large tables, many lookups

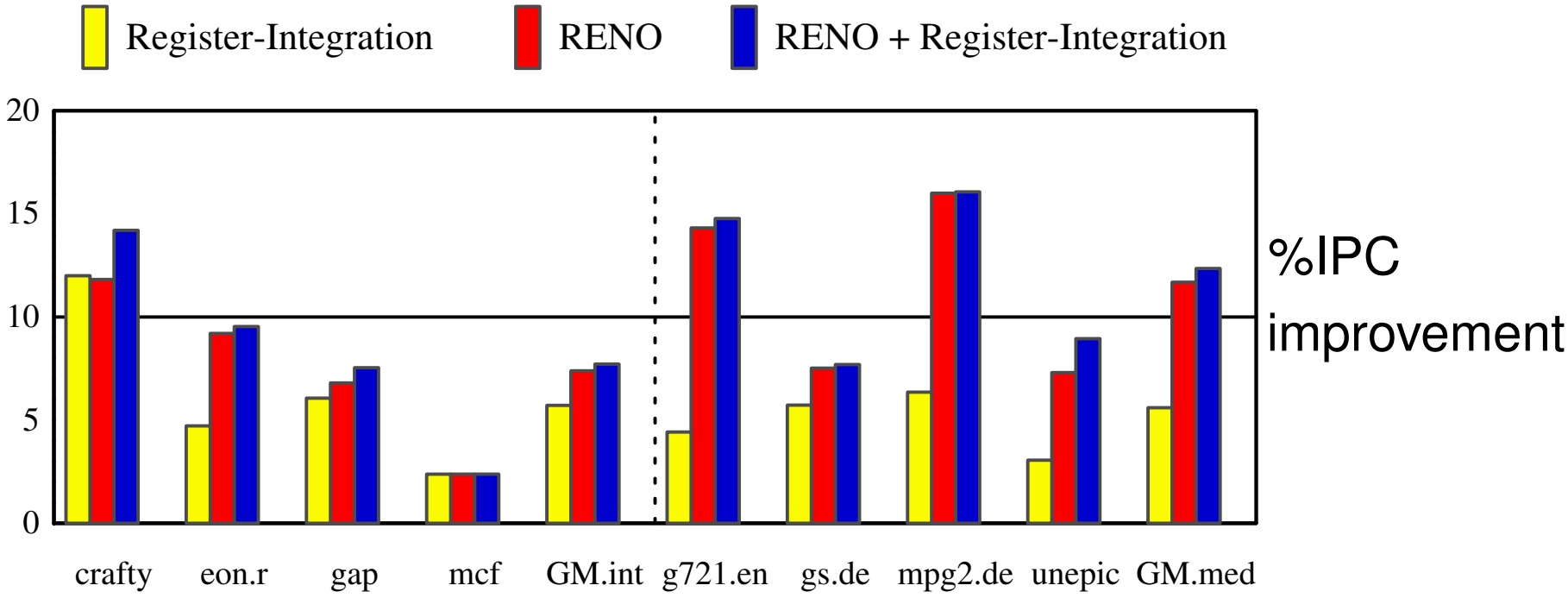
RENO$_{CSE/RA}$ / RENO$_{CF}$ Synergy



RENO: CSE for loads + folding

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

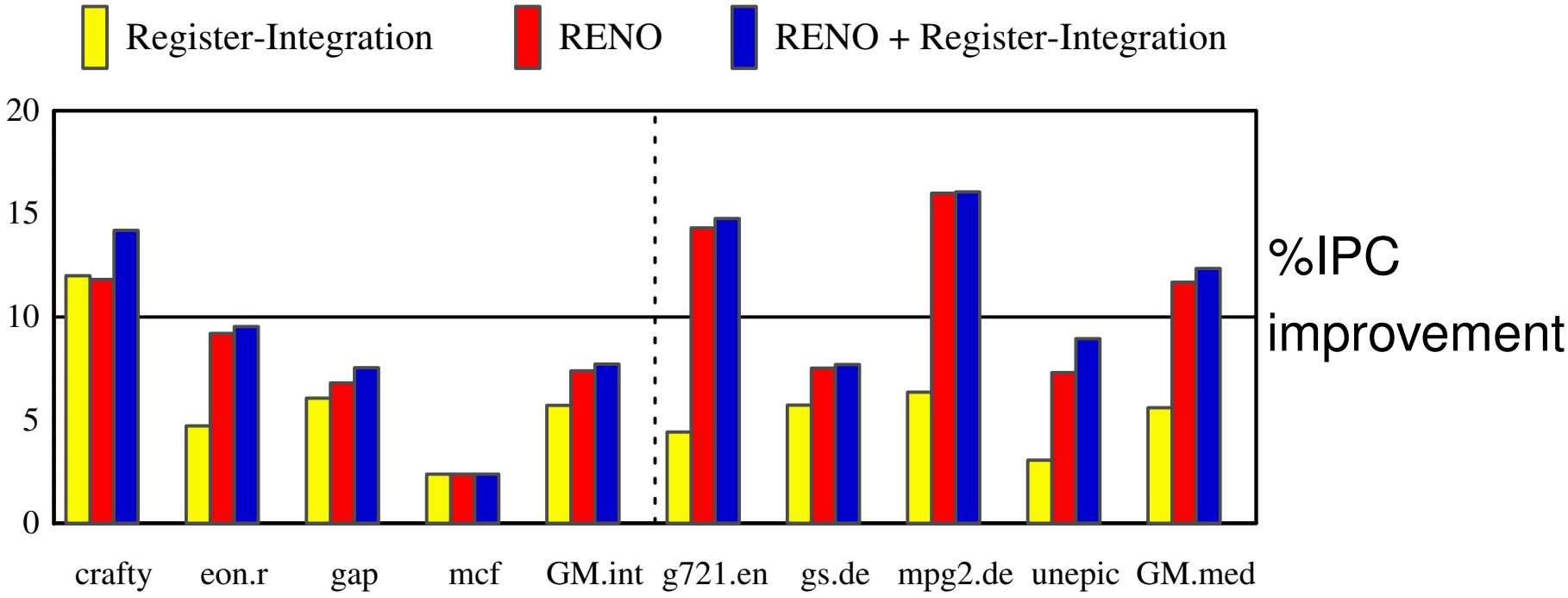
RENO$_{CSE/RA}$ / RENO$_{CF}$ Synergy



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

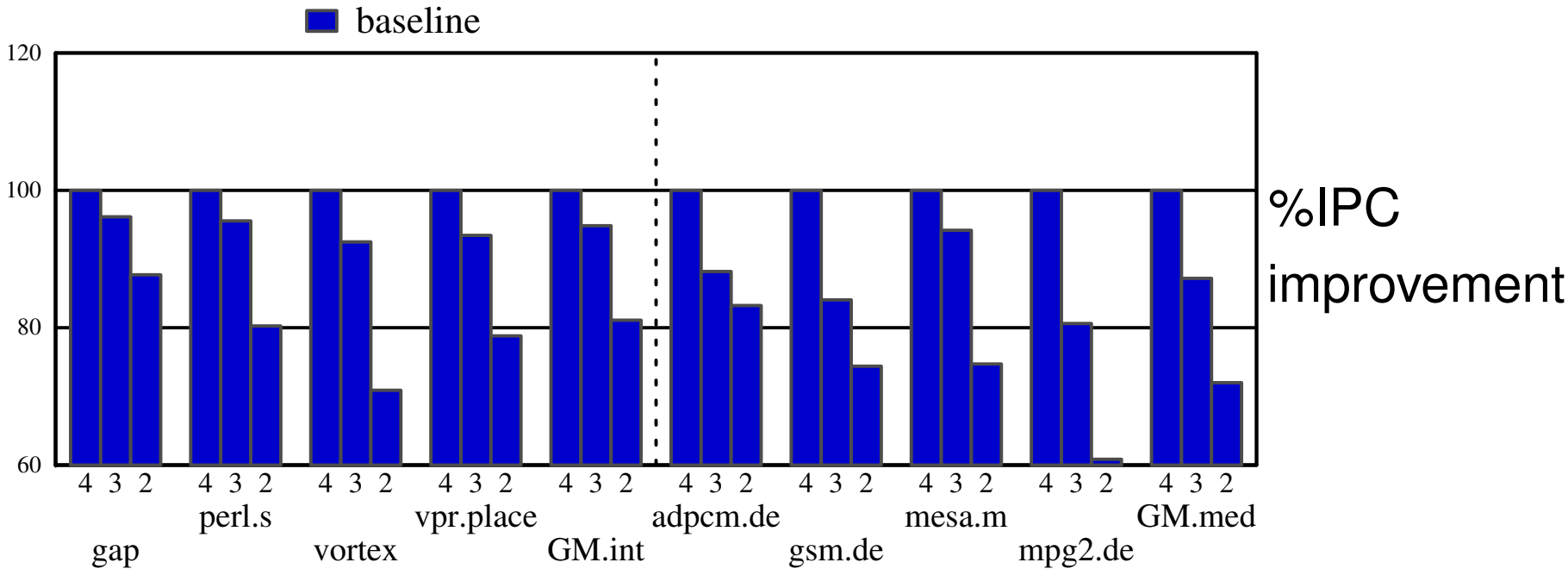
\+ Simpler implementation

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

- Short in-order execution pipeline inside renaming
- Feedback paths from main core register file

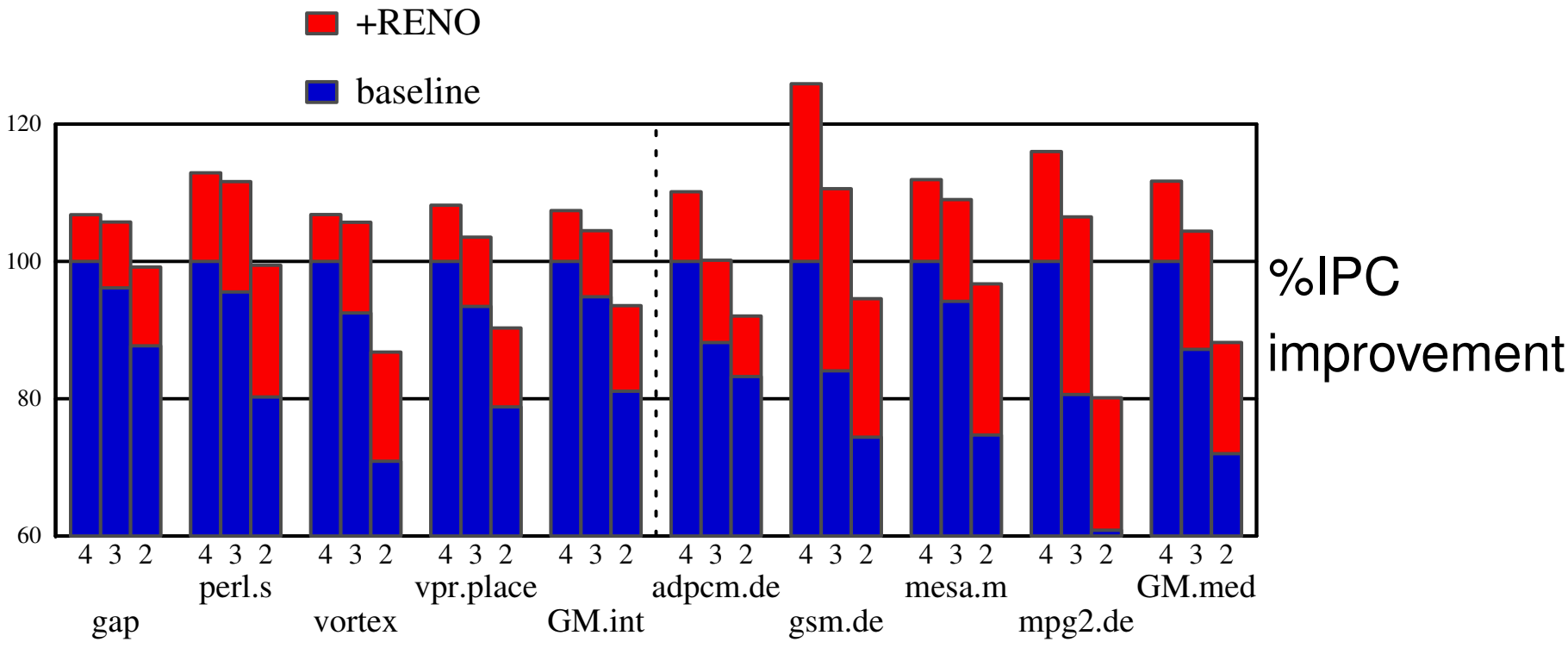
\+ Broader optimization scope

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