Exploiting Dead Value Information

Amir Roth

Milo M. Martin, Charles N. Fischer

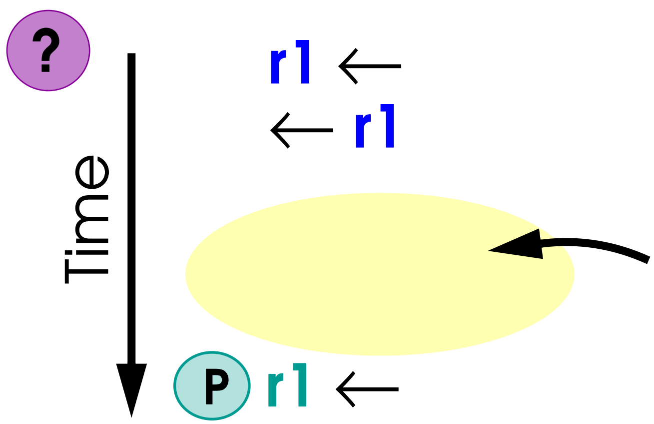
amir,milo,fischer@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison

Motivation - Dead Values

Dead Value : no longer needed for computation

Processors waste resources managing dead values



Definition to r1
Last use of value in r1

No computation using value in r1

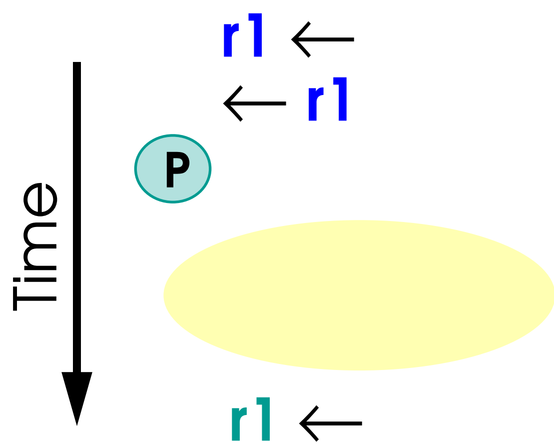
New definition to r1

value in r1 becomes dead after last use
processor discovers at new def P → too late

This work: Reduce waste for dead register values

Motivation II - Exploiting Dead Values

? What could processor do if it knew r1 is dead before ?



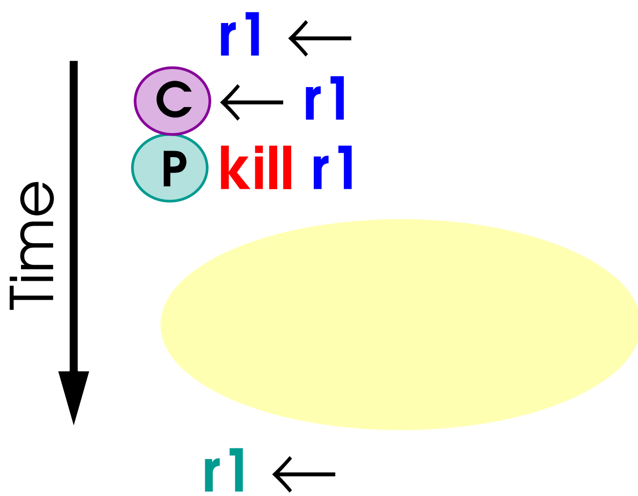
Dead values do not need to be stored
→ free physical registers for dead values

Dead values do not need to be spilled to memory
→ eliminate saves/restores of dead values in procedure calls/context switches

Motivation III - Exploiting Dead Value Information

compiler knows last register uses statically

→ mark last uses to provide processor with Dead Value Information (DVI)



New kill instruction says value in r1 is dead

Kill instruction often not necessary (later)

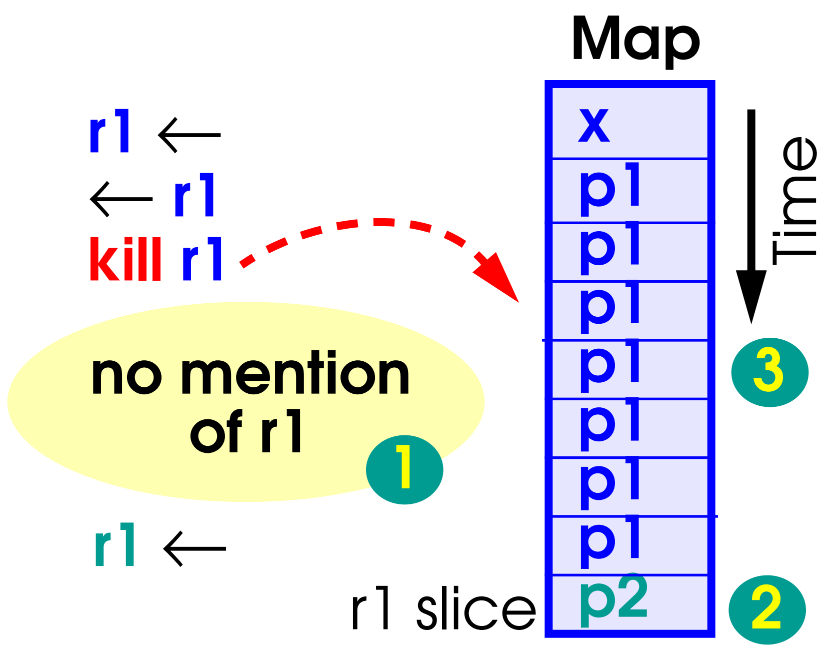
Compiler and Processor Cooperate

Outline

- ~~Motivation~~
- Applications of DVI
  - Reducing physical register requirements
  - Eliminating dead procedure saves/restores
  - Eliminating dead context-switch saves/restores
- Implementation cost of DVI
- Summary

Application 1 - Early Physical Register Reclamation

R10K style OOO machine



1 value in r1 not used

→ r1 doesn't need a physical register in

2 normally reclaim p1 when r1 ← commits

3 reclaim p1 when **kill r1** commits

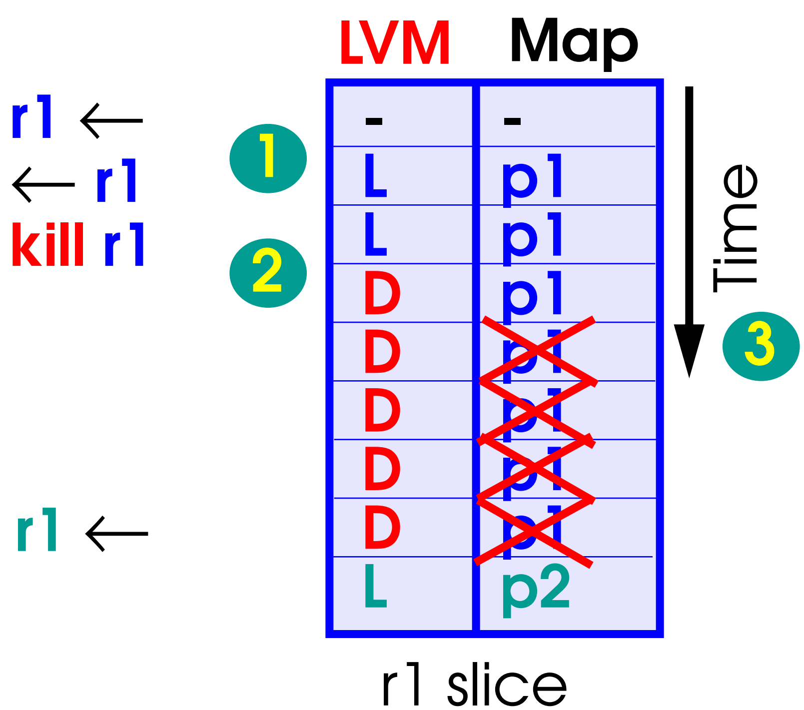
Use DVI to reclaim dead physical registers early

+ Reduce physical register requirements

→ Build smaller register files/bigger OOO windows

Mechanism - Physical Register Reclamation

- Live Value Mask (LVM) tracks value live/dead state
  - One bit per architectural register (map table entry)



At decode stage:

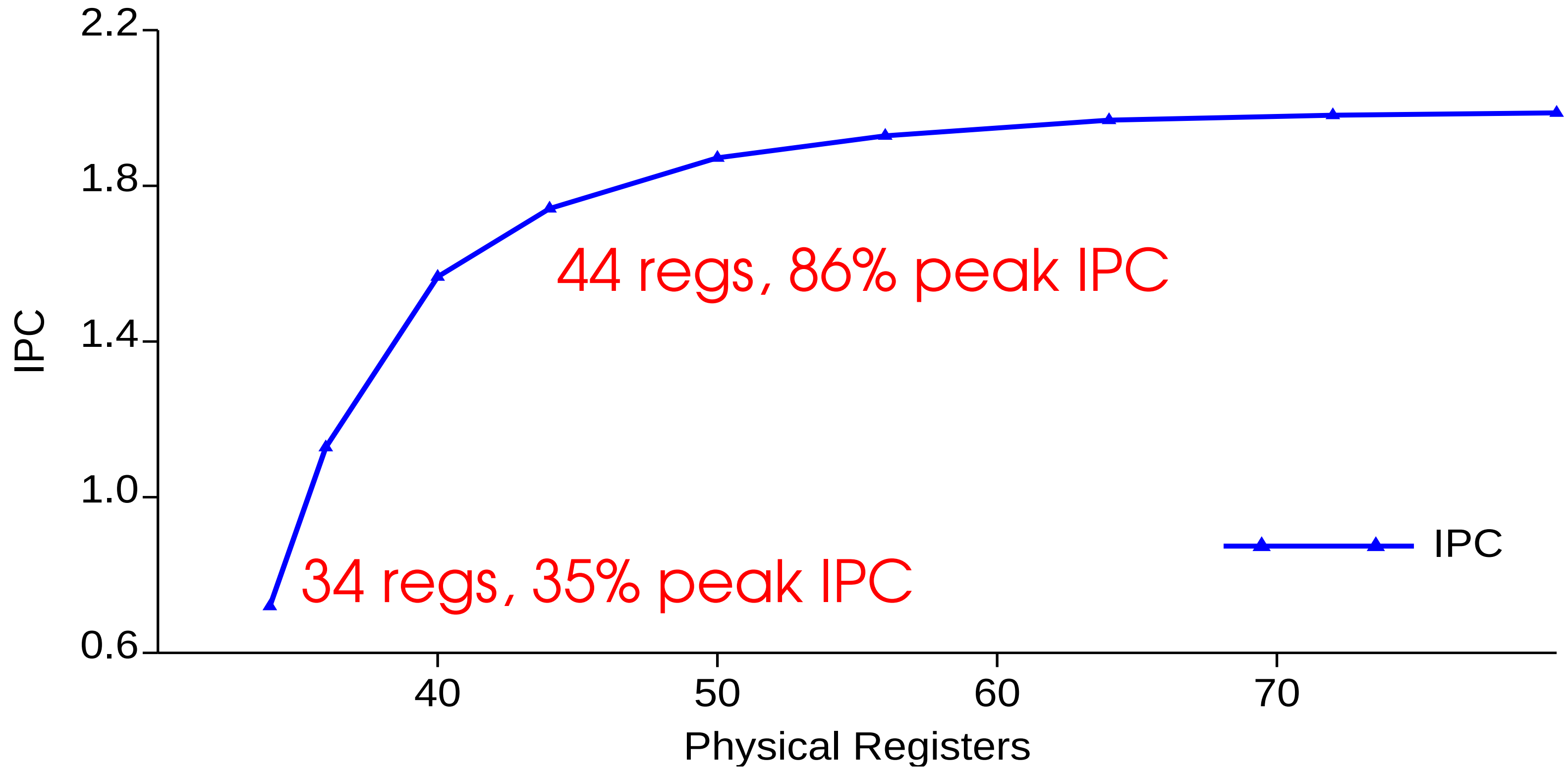
1 mark Live on r1 ←

2 mark Dead on **kill r1**

When **kill r1** commits

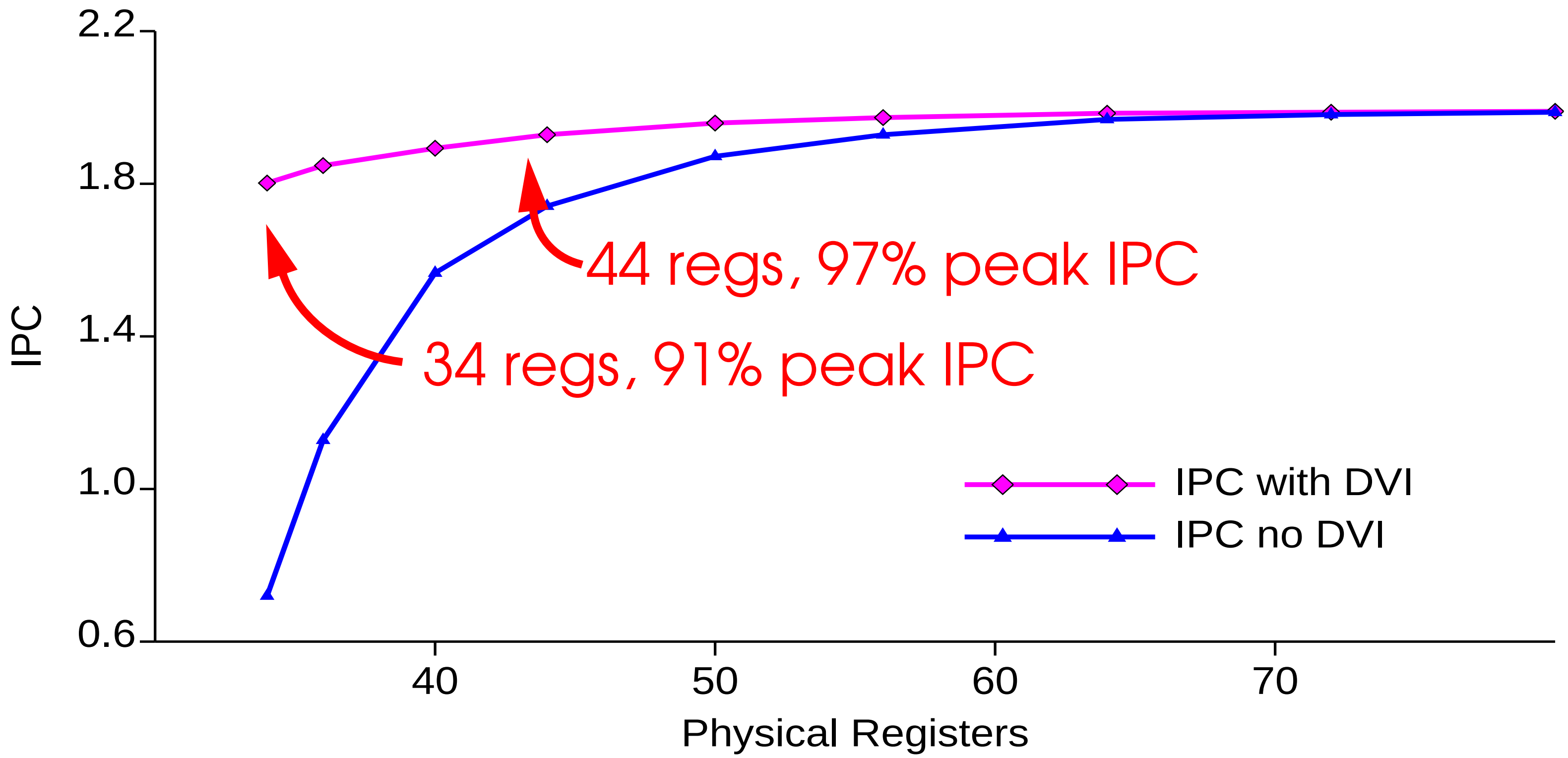
3 free p1

Experiment - Effect of Physical Registers on IPC



- 4 wide superscalar, OOO (unconstrained window)
- Aggregate IPC for SPECint95
  - IPC peaks at 60-80 physical registers (typical)
  - IPC degrades sharply below ~45 registers

Results - Register Usage Reduction



- Repeat with DVI and early reclamation
  - Achieve near peak IPC with few physical registers

$\rightarrow$ μArch with ~40 physical registers more appealing

Application 2 - Procedure saves/restores

← r16
kill r16

call proc1
save r16
r16 ←
← r16
restore r16
return

r16 ← 1

1 value in r16 dead before **proc1**

→ don't save and restore it

? Why does this happen?

proc1 and caller compiled separately

multiple paths to call

→ interprocedural opt's not completely effective

Use DVI to eliminate dead callee saves/restores

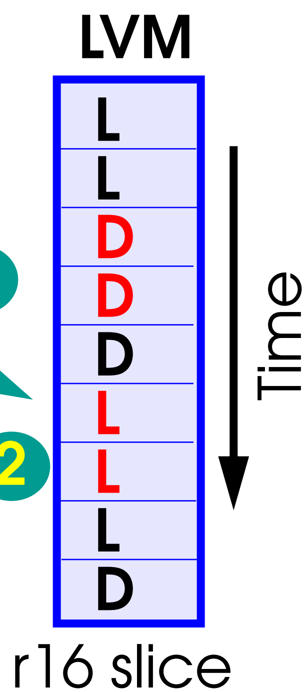
+ Reduce loads/stores executed

→ Amplify effective data cache bandwidth

Mechanism - Procedure Save/Restore Elimination

- Implement saves/restores as predicated instructions
  - Predicated = potentially not executed
- Use LVM as predicate

← r16
kill r16
call proc1
    ~~save r16~~ 1
    r16 ←
    ← r16
    restore r16 2
    return
r16 ←



1 eliminate save

2 restore not eliminated

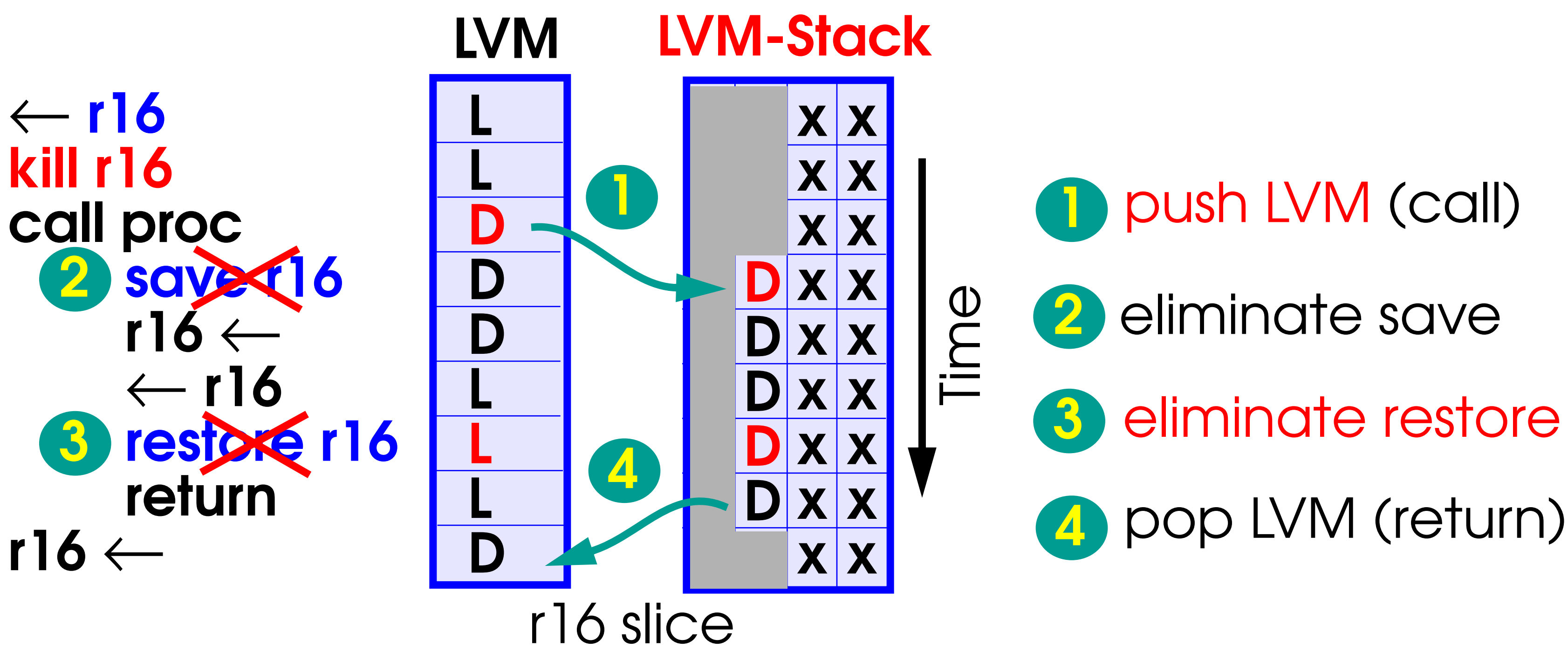
? Why not?

Interference in LVM

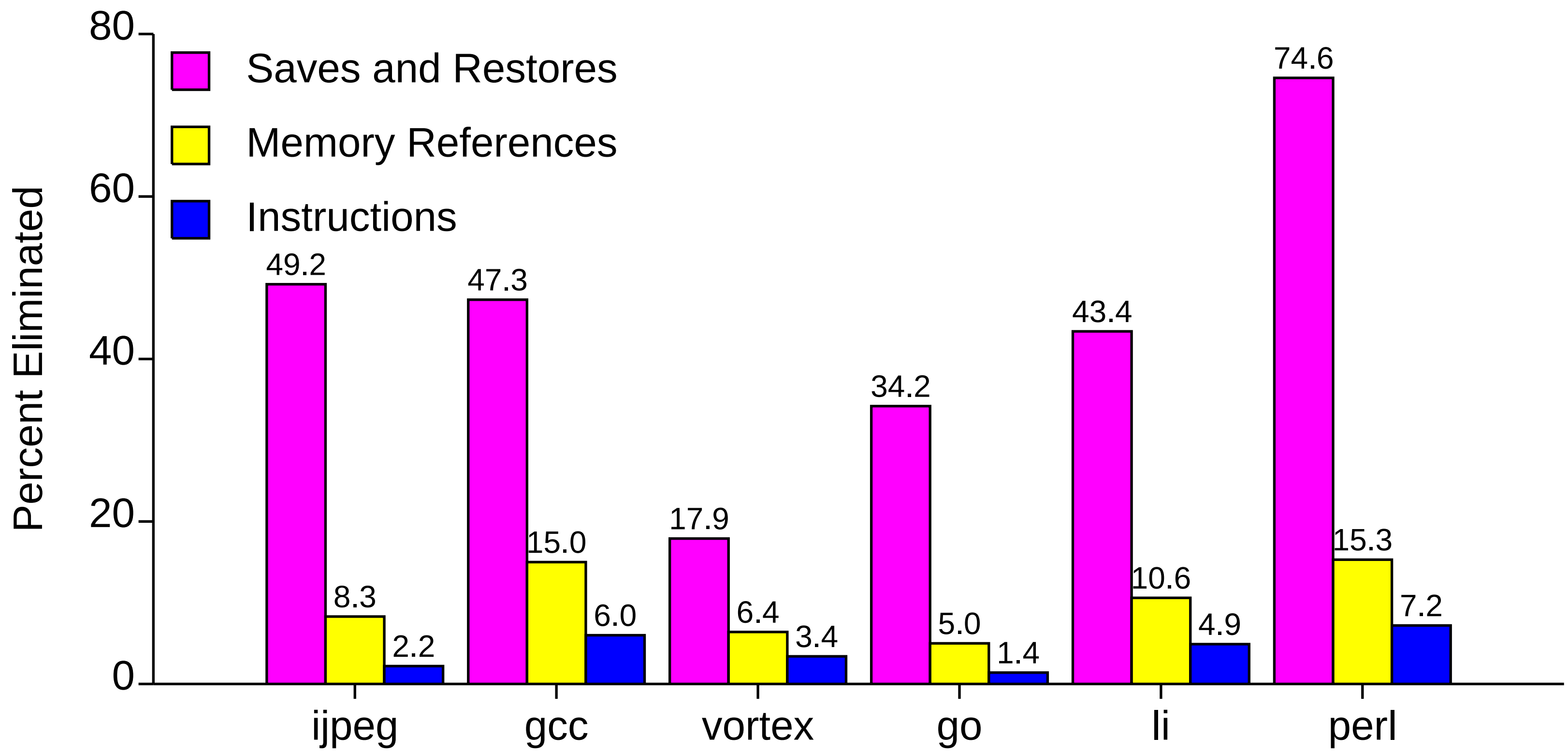
→ Save copy of DVI from **kill r16** on the side

Mechanism II - Procedure Save/Restore Elimination

- LVM-Stack tracks DVI from procedure entry to exit
- Use top of LVM-Stack as predicate
- Push/pop on call/return (implicitly)

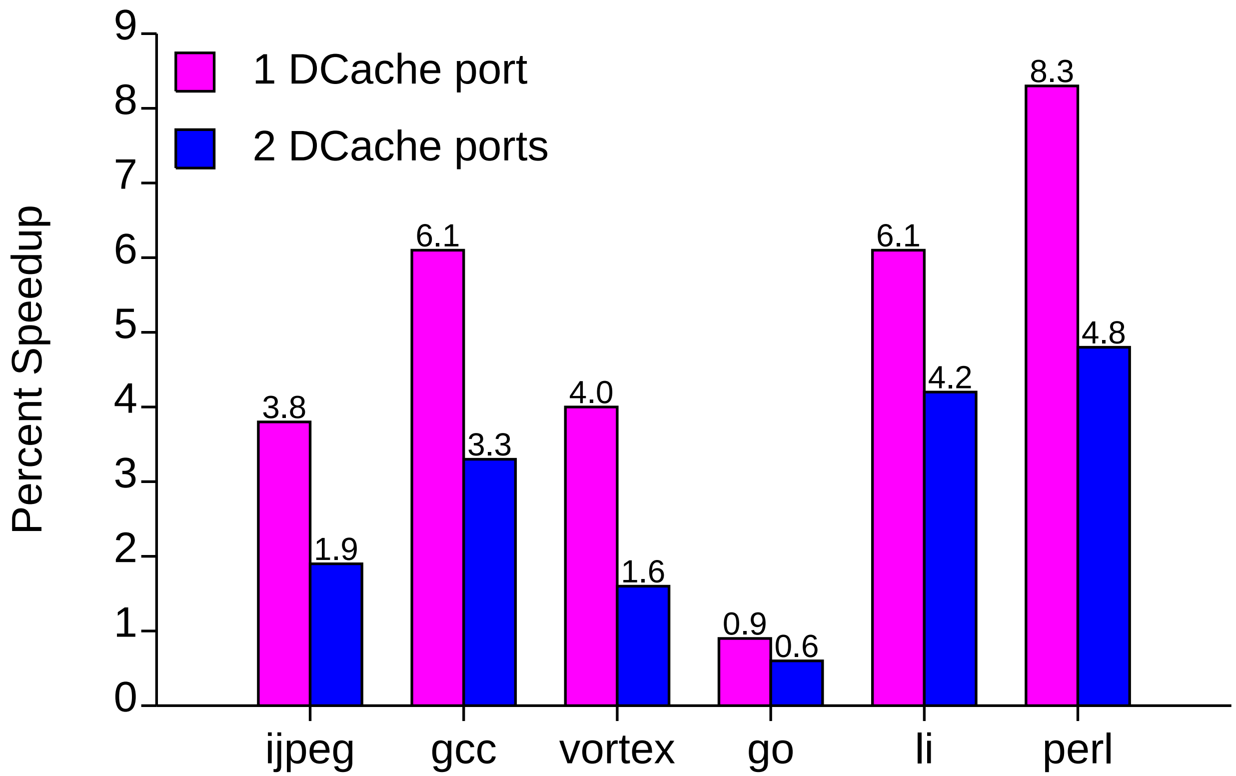


Results - Procedure Saves/Restores Eliminated



- 46% of callee save/restore code not executed
- Data cache accesses reduced 11%
- Instructions executed reduced 5%

Results - Speedups



- 4 wide superscalar, 64 window OOO
  - 5% speedup with 1 data cache port
  - 3% speedup with 2 data cache ports

Application 3 - Context Switch Save/Restore Code

← r16
kill r16

suspend context
save r16

resume context
restore r16

r16 ← 1

1 value in r16 not used after the switch

→ don't save and restore it

? Why does this happen?

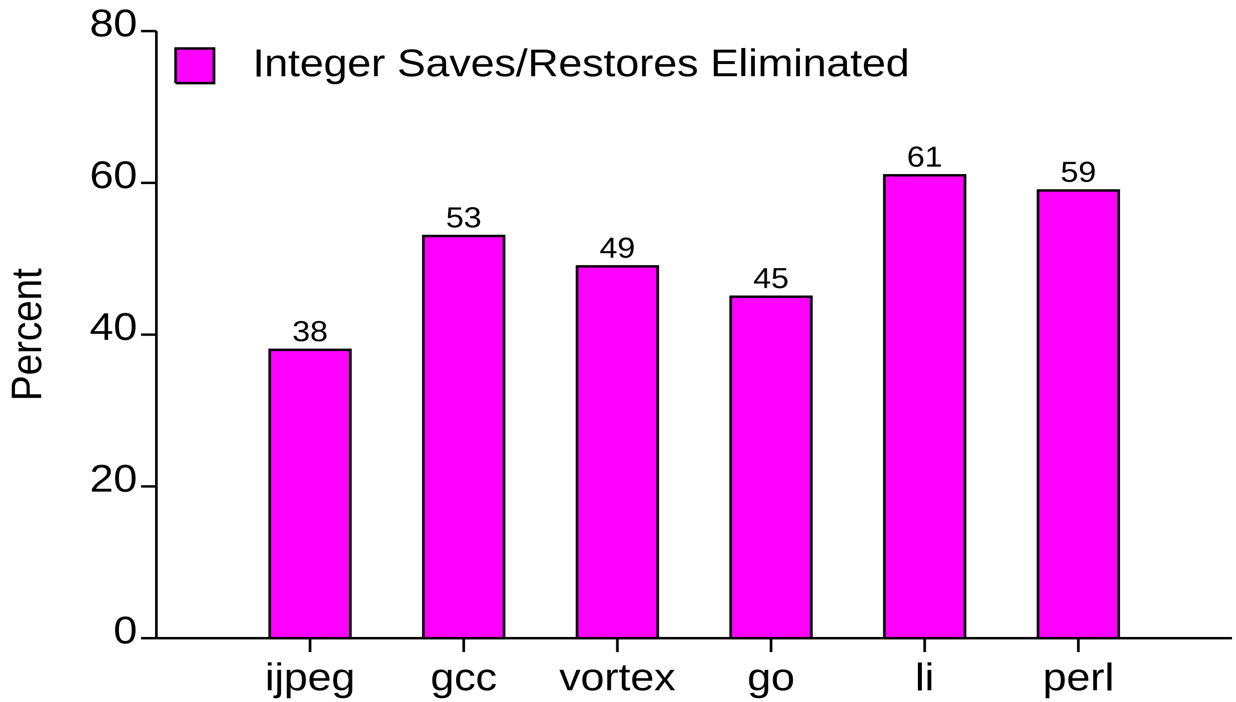
Switch manager cannot know which values are dead

Use DVI to eliminate saves/restores in switch code

+ Potential benefit for multithreaded applications

+ Can handle preemptive switches

Results - Context Switch Saves/Restores Eliminated



- Approximate dead value count at preemptive switch
  - Single-threaded programs (SPECint95)
  - Sample LVM
- 51% of integer save/restore code not executed

Do I really have to add all these instructions?

- So far, DVI given explicitly using kill instructions (E-DVI)
  - Overhead, new kill instruction

- Some DVI implicit in calling convention (I-DVI)
  - Caller-saved registers dead at function entry/exit
  + No overhead
  + No new kill instruction

- Our implementation
  - I-DVI for caller-saved registers
  - Supplement with E-DVI for callee-saved registers
    - One inst before every call (mask kills up to 8 regs)

How effective is I-DVI?

- Very effective for register file optimization
  - + 89% peak IPC at 34 registers (vs 91%)
- Not effective for procedure save/restore elimination
  - - No I-DVI for callee-saved registers
  - + One E-DVI instruction before every call sufficient
- Effective for thread switch save/restore elimination
  - + 41% of integer saves/restores eliminated (vs 51%)
- Intuition?
  - “Housecleaning effect”

Summary

Benefits of DVI

- Physical register requirements reduced (up to 50%)
- Procedure saves/restores reduced 46%, IPC up 5%
- Context switch saves/restores reduced 51%

Implementation Cost of DVI

- I-DVI sufficient for many optimizations and free
- Only a few instructions added to ISA
- Simple and small hardware structures
- No correctness semantics = no legacy problems

DVI = Significant Benefits + Low Implementation Cost