Path-Sensitive Value-Flow Optimizations

Rastislav Bodík

University of Pittsburgh

Motivation

Value-reuse redundancy:

- **Instructions recompute previously computed values**

  optimization: do not recompute the value, reuse the old one !

  - **benefit 1:** fewer instructions to execute
  - **benefit 2:** instructions ready to execute earlier

- **How much redundancy?**

	instructions	conditional branches
Compilers remove:	**30 %**	**10 %**
Processors predict results:	**80 %**	**95 %**

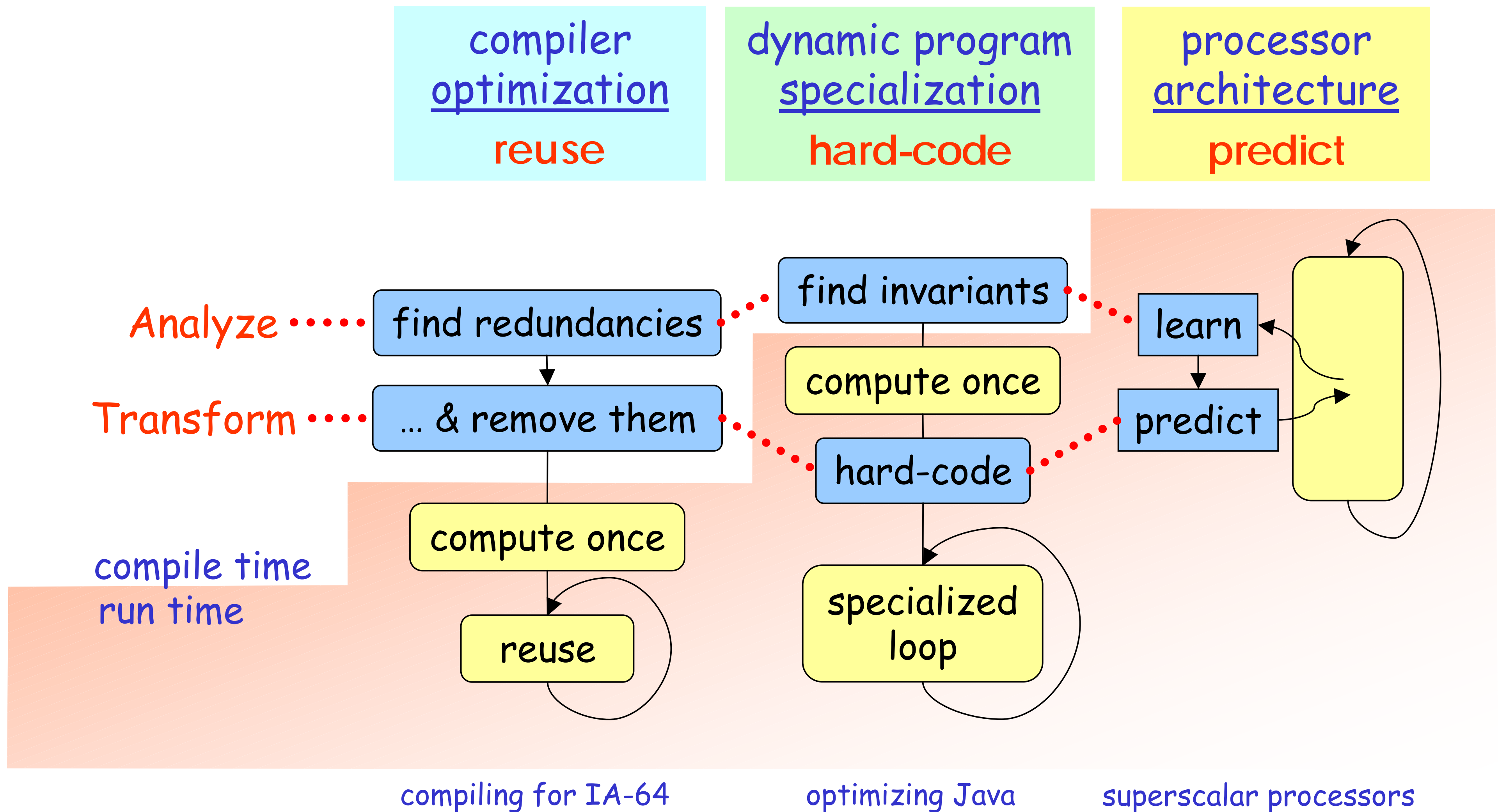
Why so much redundancy?

a fragment from Merge Sort:

```
while   ...   do
    if  compare( A[i], B[k] )
        C[ top++ ]  :=  A[ i++ ]
    else
        C[ top++ ]  :=  B[ k++ ]
end while
```

```
compare( x, y )
{
    if  x.f < y.f
        return 1
    else
        return 0
}
```

The optimization spectrum

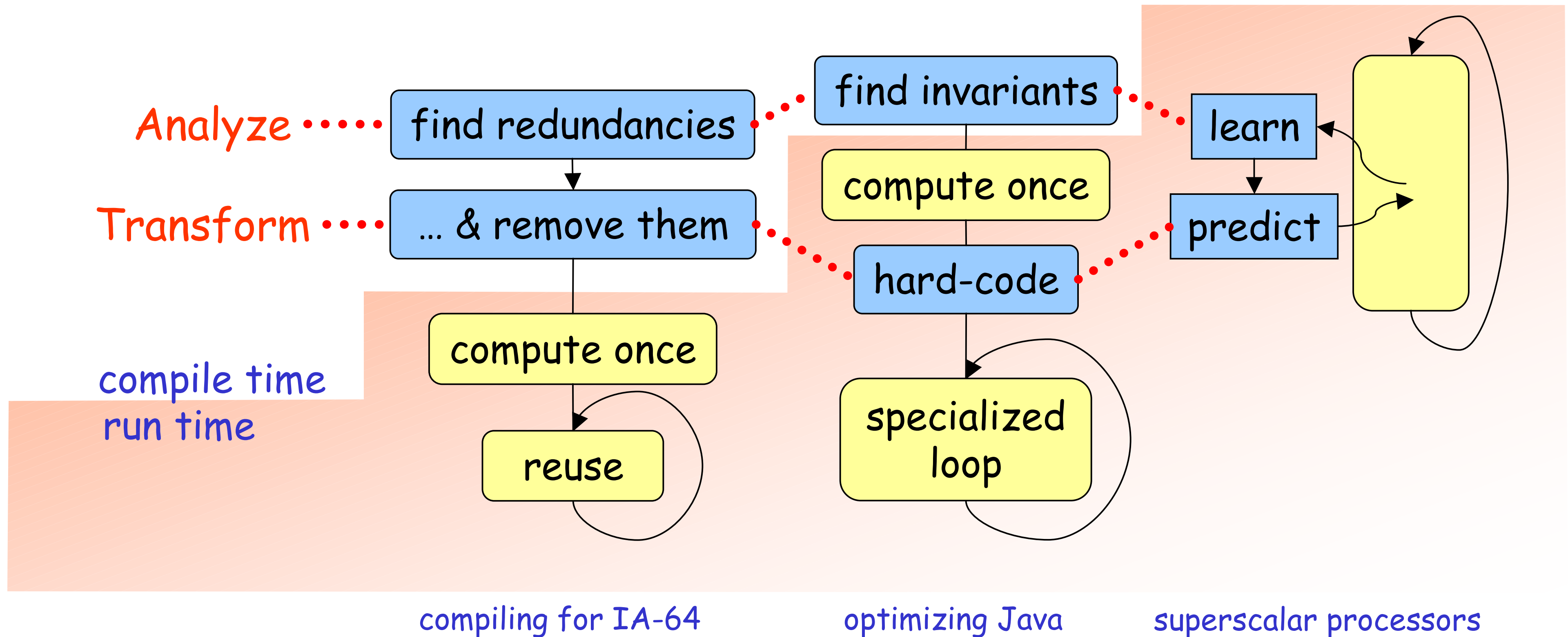


The optimization spectrum

My thesis: →

+ no run-time cost
- blind to run-time values

compiler optimization	dynamic program specialization	processor architecture
reuse	hard-code	predict

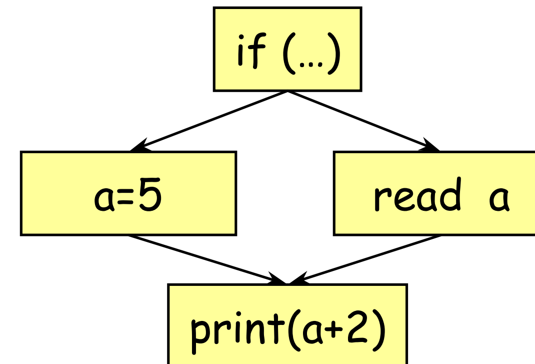


Path sensitivity

Optimization opportunity may be:

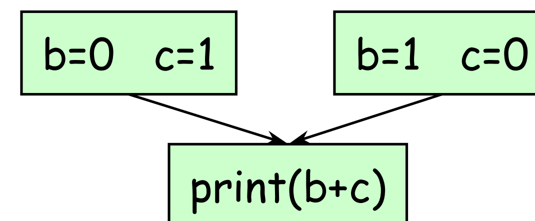
partial

1. possible only along <u>some</u> control flow paths



diluted

2. visible only when paths are examined <u>separately</u>



Path sensitivity

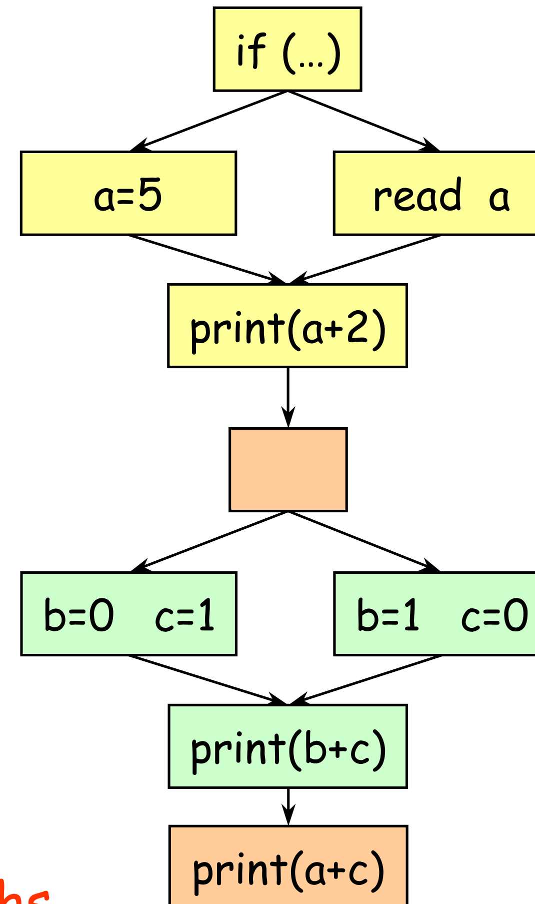
Optimization opportunity may be:

partial

1. possible only along some control flow paths

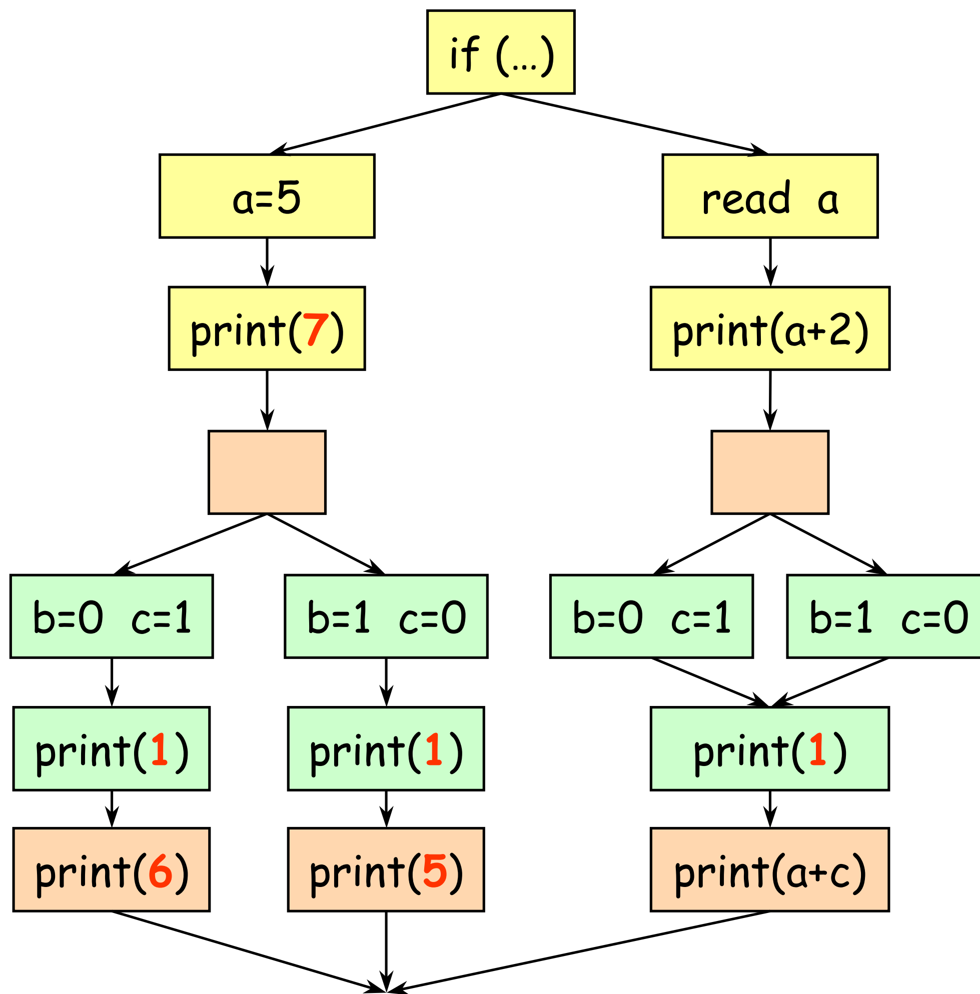
diluted

2. visible only when paths are examined separately



Problem: exponential number of paths

Exponential path explosion



Explosion in

- analysis
- transformation

Goal:

- exploit paths as much as is practical

Path-sensitive optimizers exist, but

- room for improvement
- I can double the gains

My thesis

- **Claims:** path-sensitivity can be
  - effective: improve the optimizer
  - practical: "immune" to exponential blowup
  - broad: cover many redundancy optimizations
- **Result:** Pathfinder, a framework for

path-sensitive value-flow optimizations

How? What computations?

Handle both flavors:

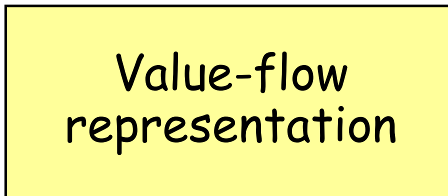
partial:
exists on some paths

diluted:
different on some path

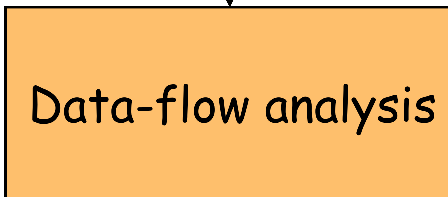
Elimination of: common sub-expressions, loop-invariants, partial redundancies, array bound checks, conditional branches, loads and stores, constant propagation, dead value elimination, etc.

Pathfinder components

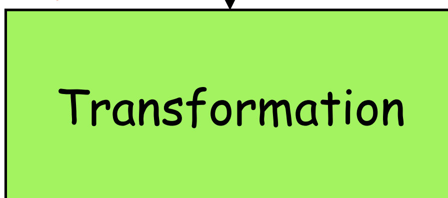
expose



collect



exploit

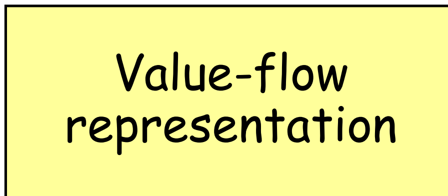


diluted

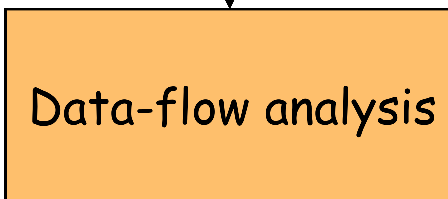
partial

Pathfinder components

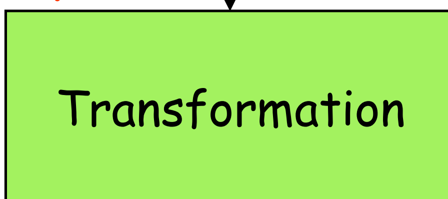
expose



collect



exploit



diluted

partial

model the flow of values

- which computations are equivalent ?
- along which paths ?

traverse the representation

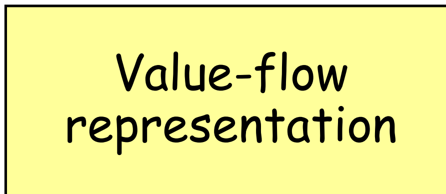
- can a value be reused on some path ?
- how often? (profiling)

remove redundancies

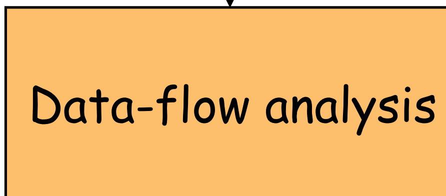
- separate paths
- move instructions

Tradeoffs

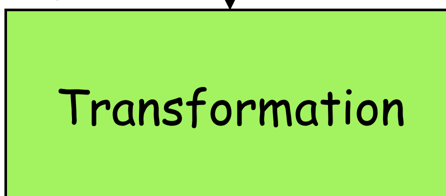
expose



collect



exploit



precision: avoid dilution

cost: do not separate all possible paths!

precision: collect all exposed reuse

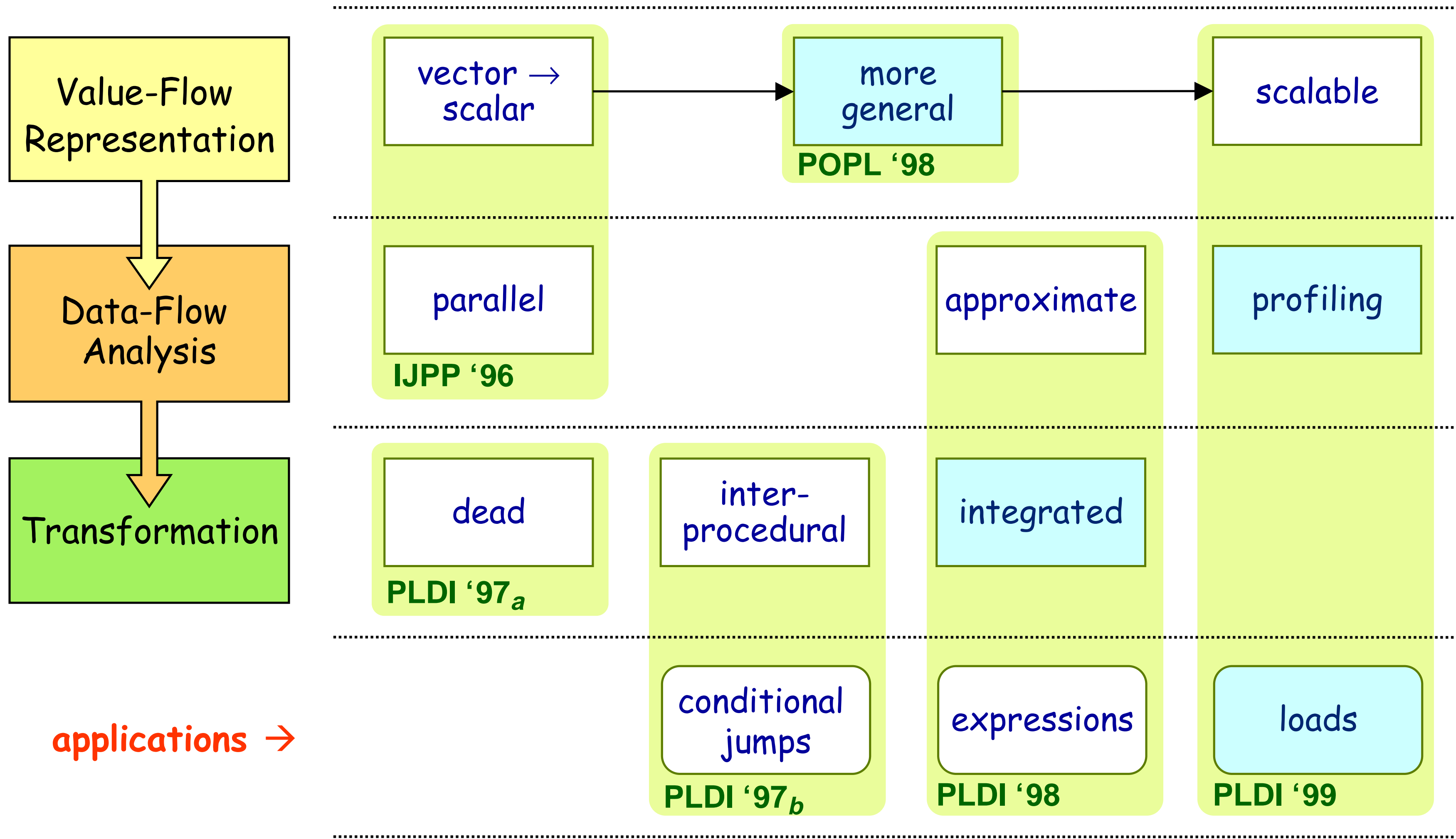
analysis cost: representation may be large!

profiling cost: number of paths to profile!

completeness: remove all redundancies

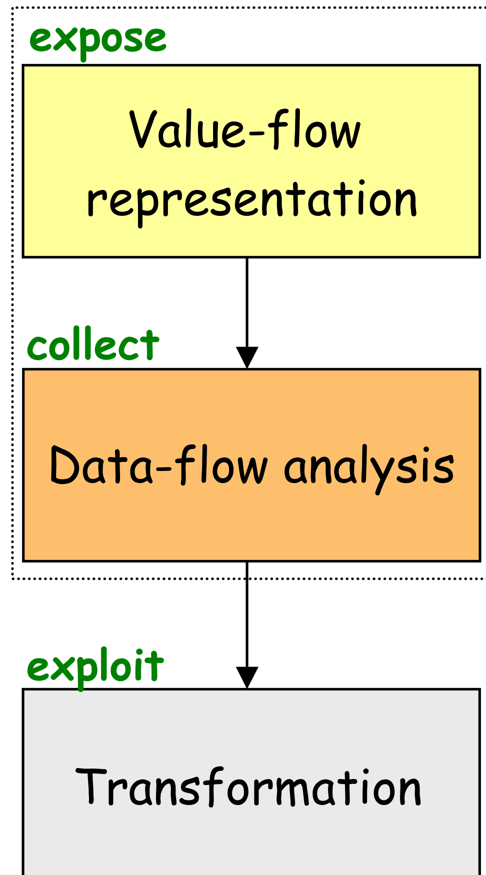
code growth: careful with restructuring!

Thesis results



Representation + Analysis

R
A
T



Tradeoffs

precision: expose "many" equivalences

- **my approach**: each path separately

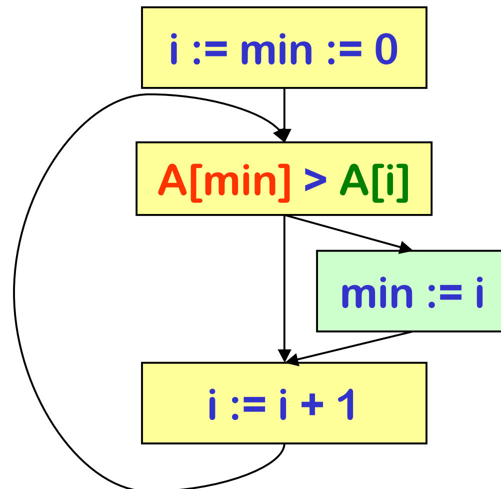
cost: number of paths to be examined

- **my approach**: separate a path only to prevent dilution

Value-flow representation

Must answer two questions:

- which instructions are value-equivalent ?
- along which control flow paths ?

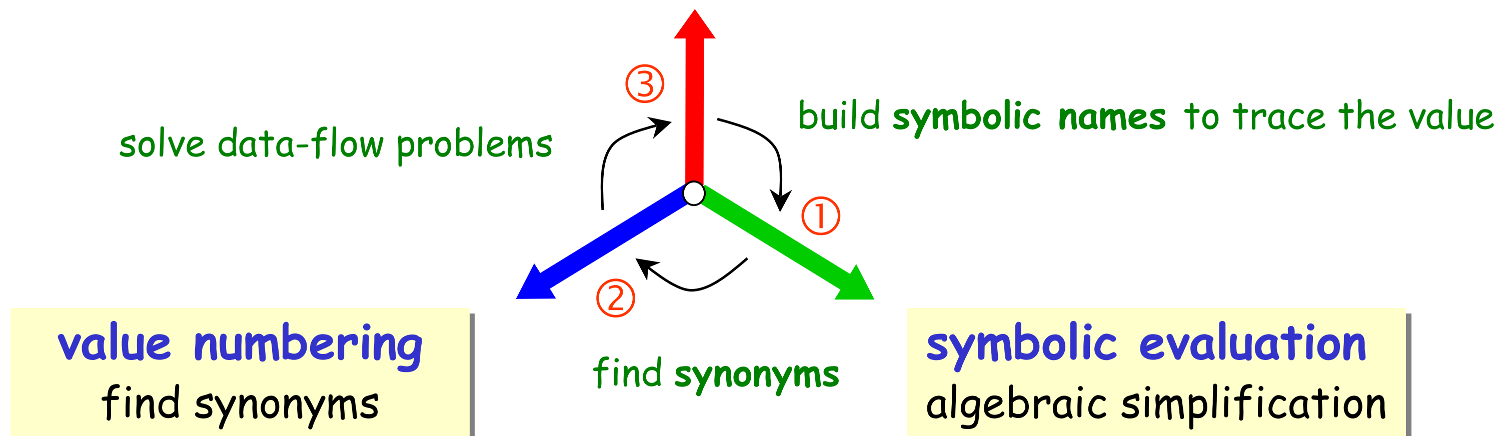


Value Name Graph (VNG)

R A T

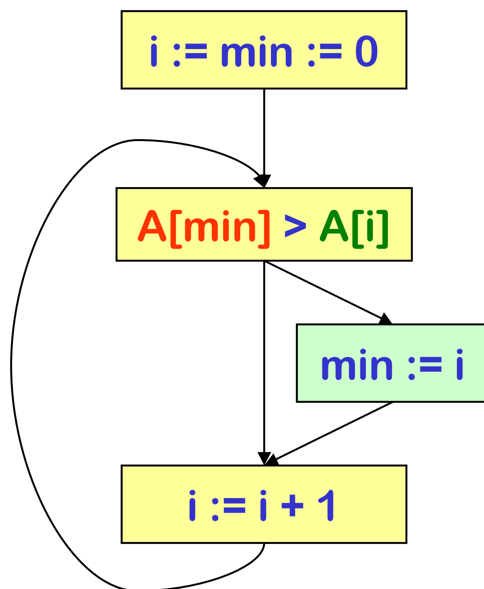
Combines three orthogonal techniques:

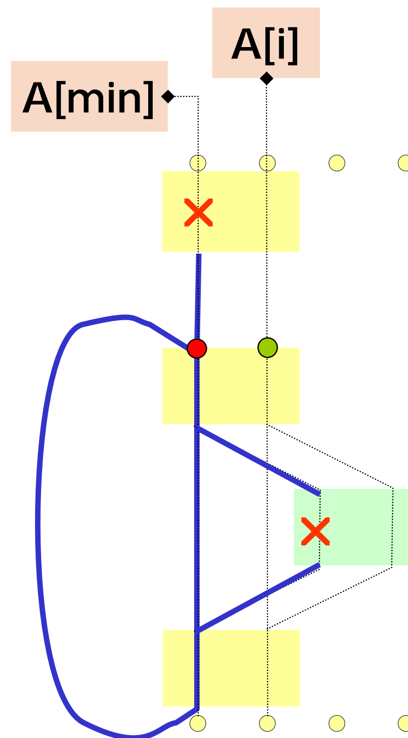
- integration removes their individual limitations



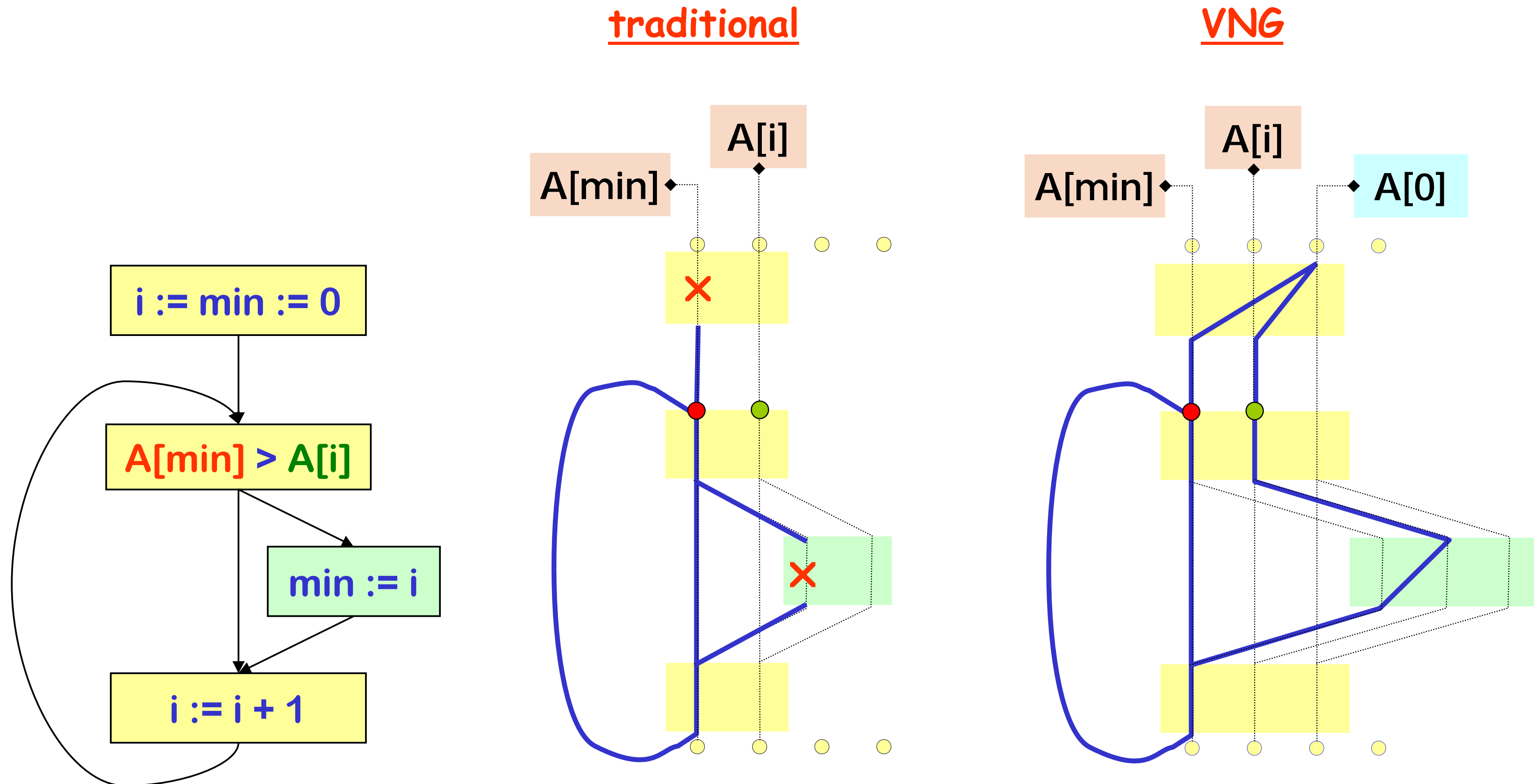
VNG: example

traditional



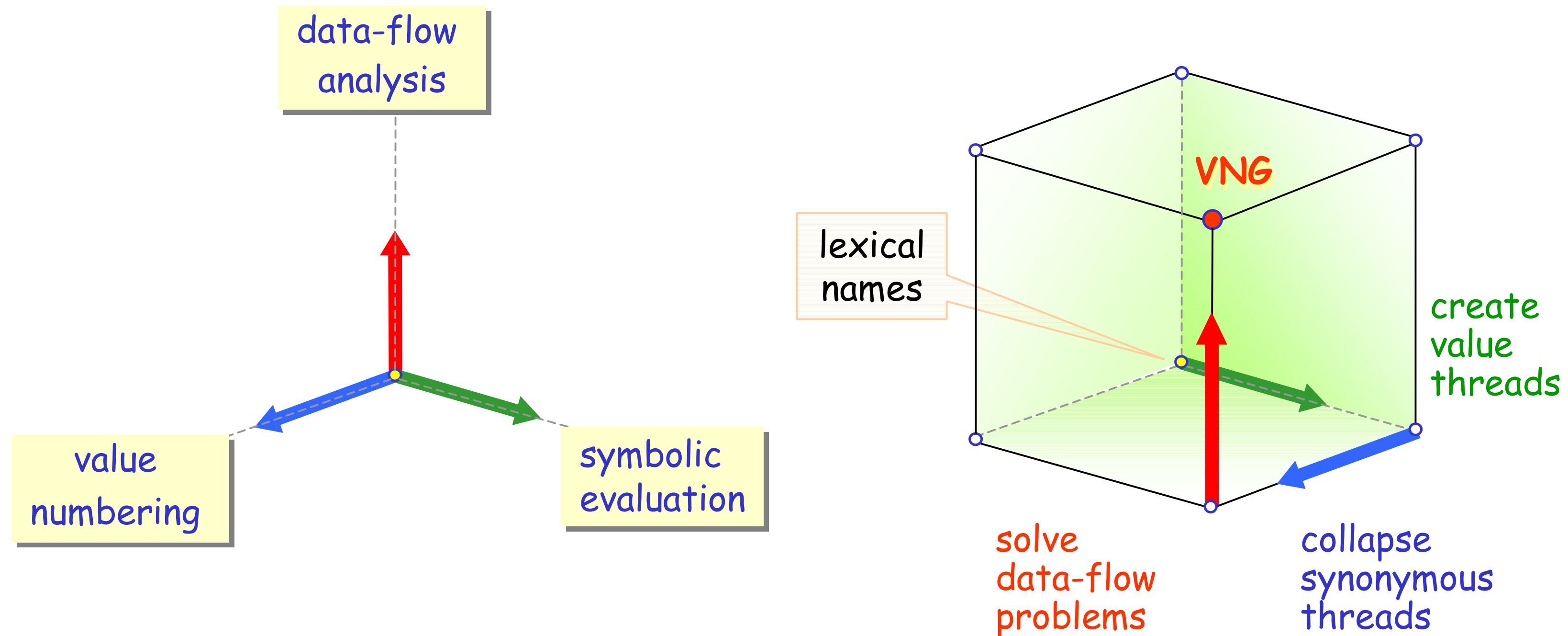


VNG: example



Constructing the VNG

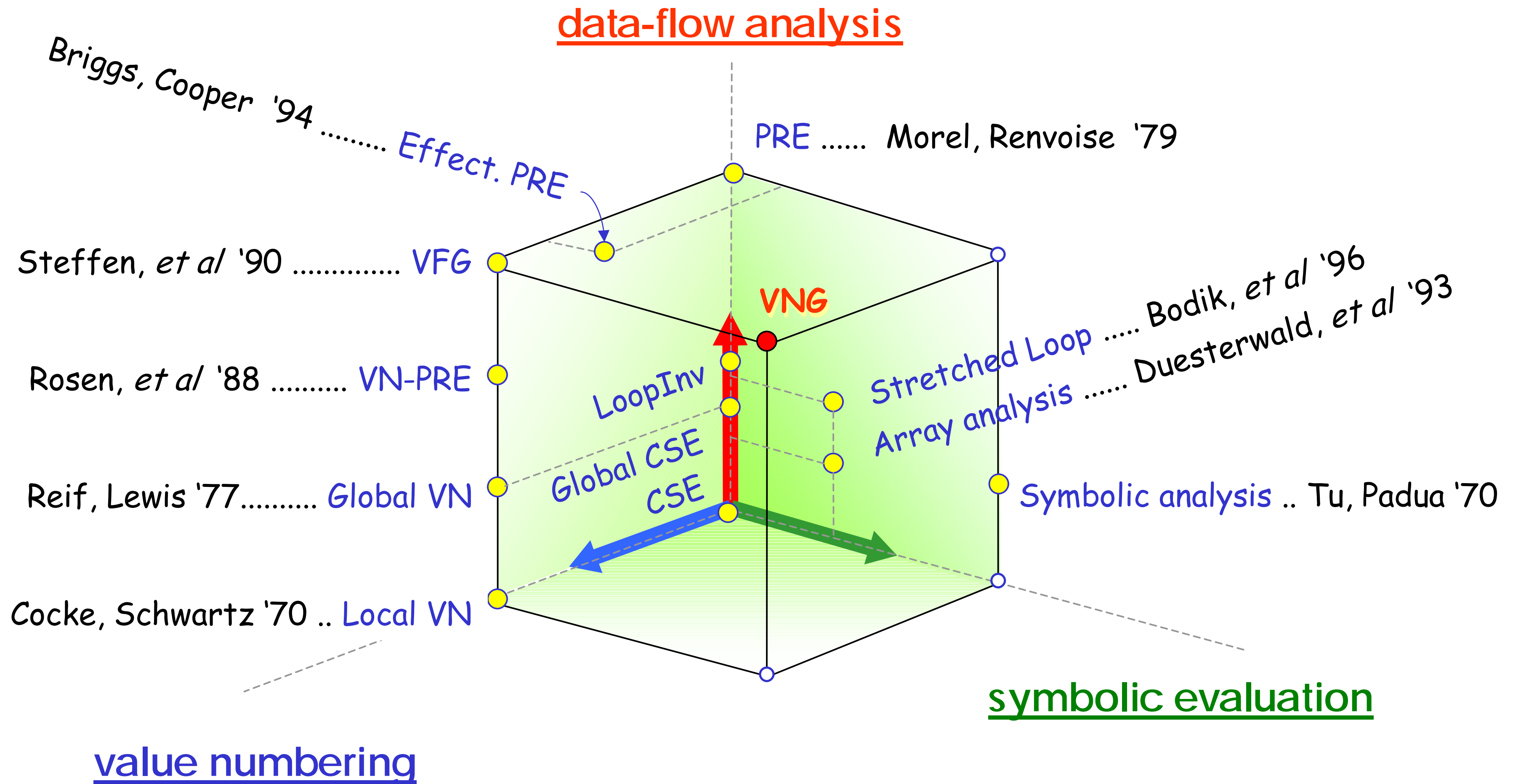
R A T



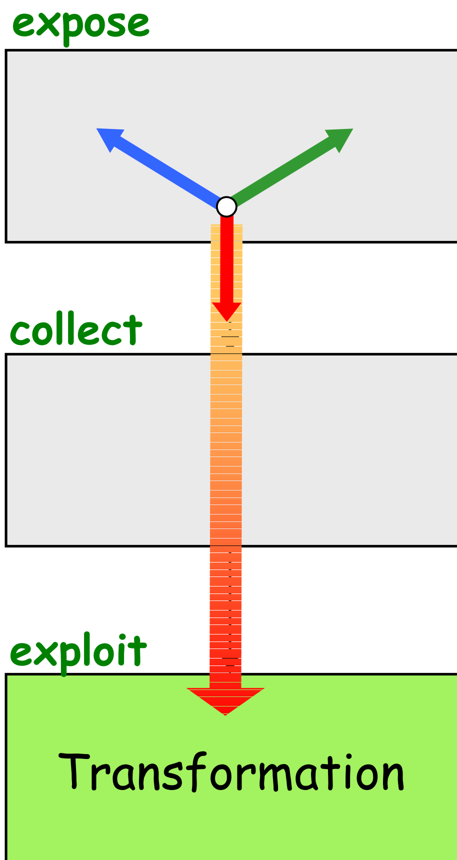
Value threads created on demand:

- VNG separates only paths on which a value has different names
- still, exponential number of threads (but moderate in practice)

VNG: related work



Transformation



Tradeoffs

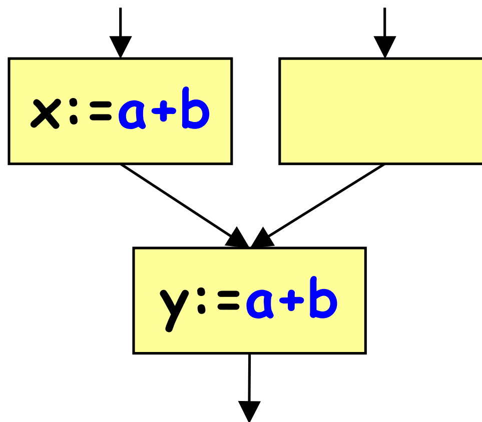
completeness: remove all redundancies collected by analysis

code growth: due to path duplication

- impairs subsequent compiler stages
- impairs cache

PRE: partial redundancy elimination

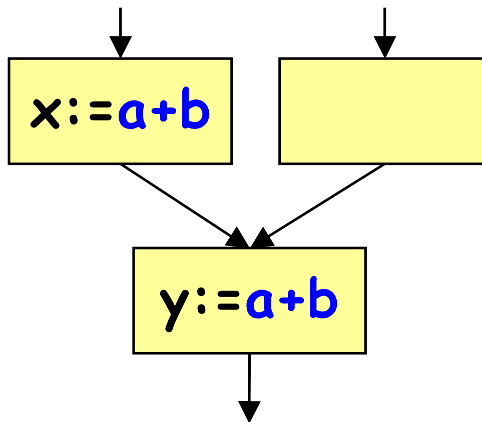
Partially redundant: computed on <u>some</u> incoming path.

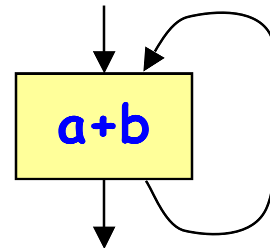


PRE: partial redundancy elimination

RAT

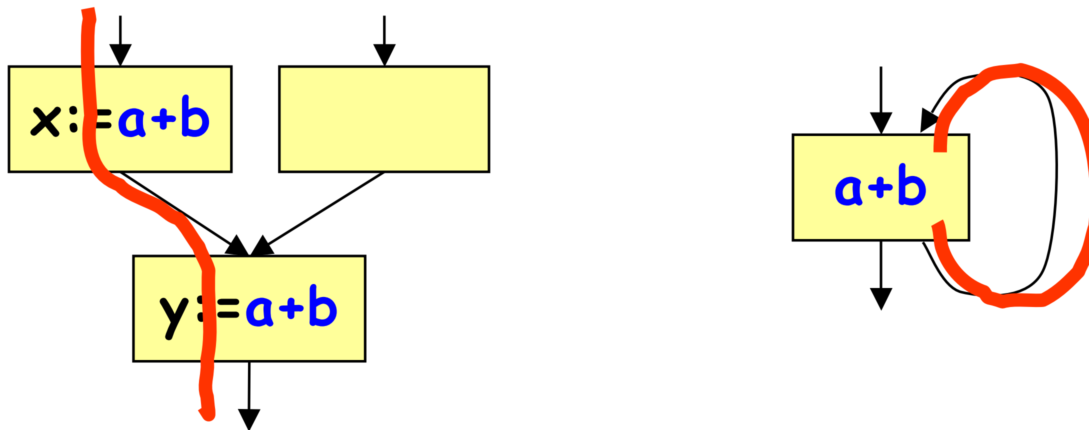
Partially redundant: computed on <u>some</u> incoming path.





PRE: partial redundancy elimination

Partially redundant: computed on some incoming path.

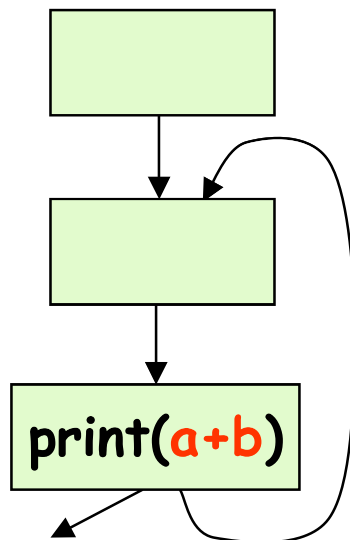


Goal: remove redundancy from "reuse" paths.

PRE via code motion [Morel, Renviose '79]

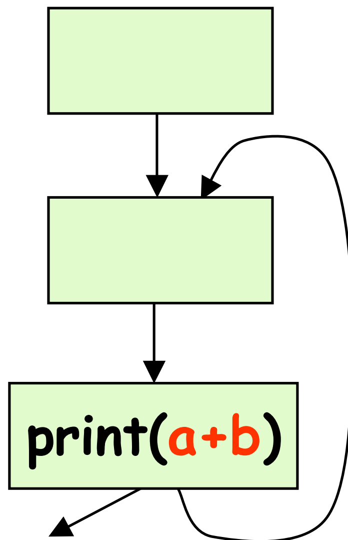
R
A
T

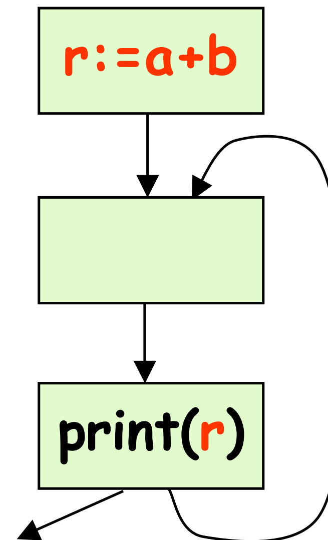
- **Hoist partially redundant expression until it becomes:**
  - fully redundant ⇒ remove it
  - not redundant ⇒ insert it



PRE via code motion [Morel, Renviose '79]

- **Hoist partially redundant expression until it becomes:**
  - fully redundant ⇒ remove it
  - not redundant ⇒ insert it

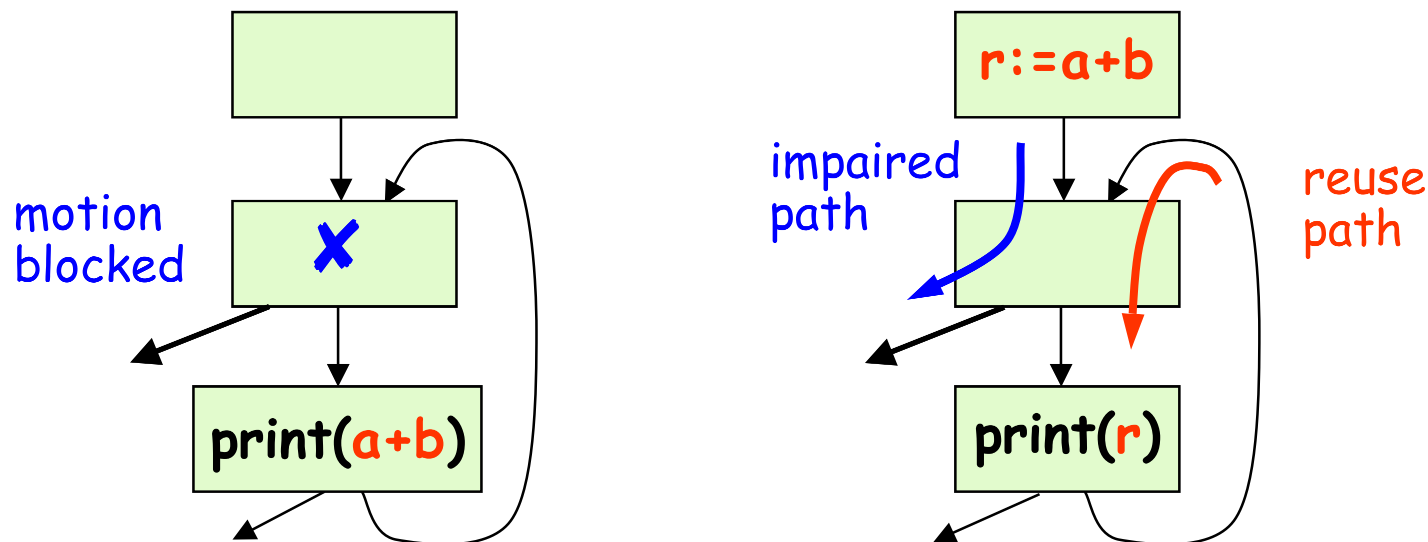




PRE via code motion [Morel, Renviose '79]

R A T

- **Hoist partially redundant expression until it becomes:**
  - fully redundant ⇒ remove it
  - not redundant ⇒ insert it



PRE fails on 70% of loop-invariants !

Morel-Renviose limitations

R A T

Optimization model
too conservative

Rule:
improve "reuse" paths
without
impairing other paths

Program transformation
not aggressive

Code motion only:
motion blocked
⇓
opportunities missed

My approach:

Relaxed
allow control speculation

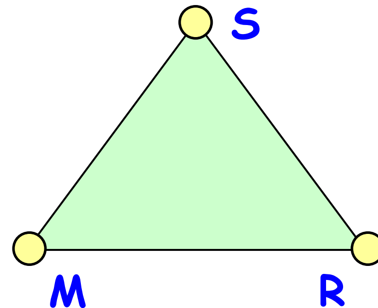
Aggressive
apply restructuring

Combining the transformations

R
A
T

+ no code growth
- impairs some paths

control speculation



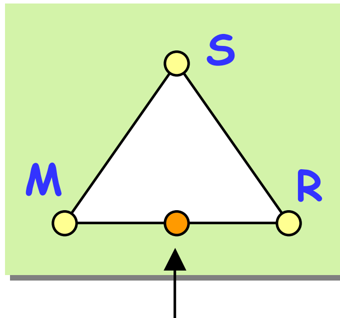
code motion

restructuring

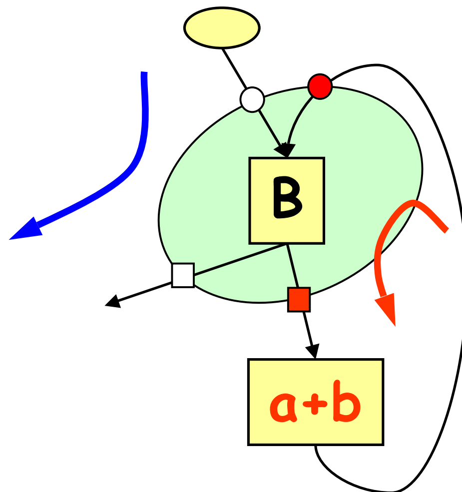
+ no code growth
- misses opportunities

+ complete
- code growth

CMP: code motion preventing region

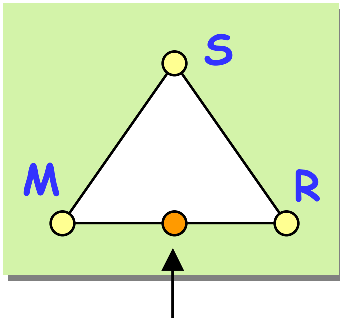


- duplicate only nodes that block code motion

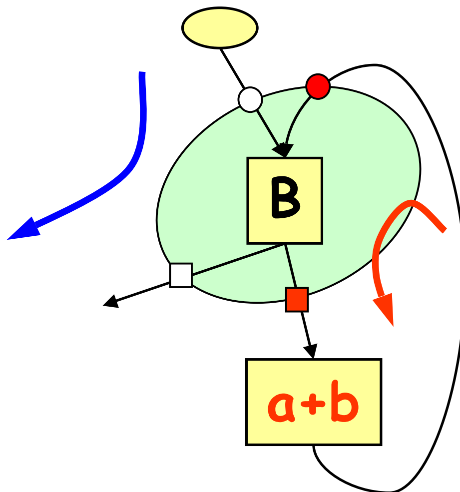


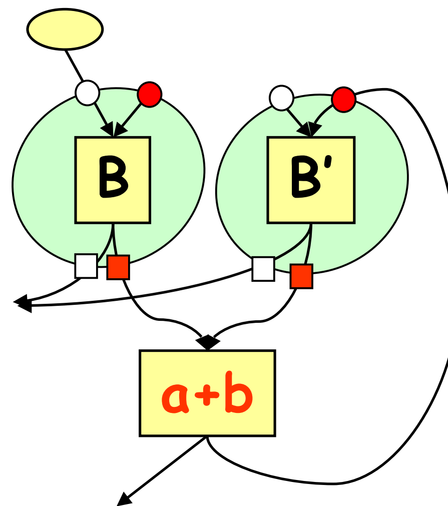
CMP: code motion preventing region

R
A
T

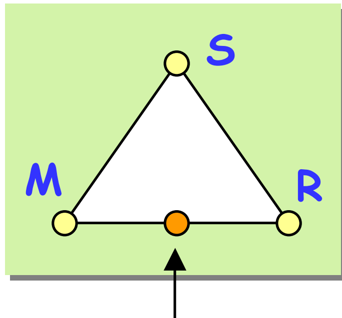


☛ duplicate only nodes that block code motion

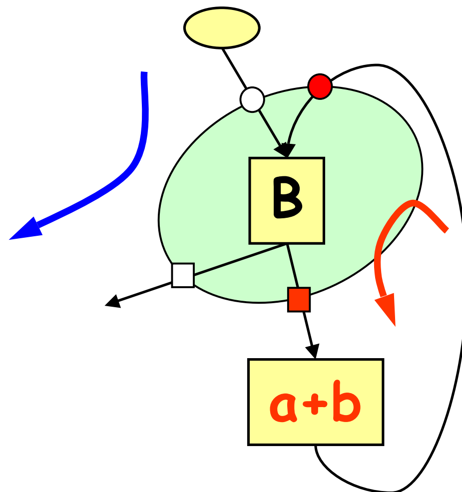


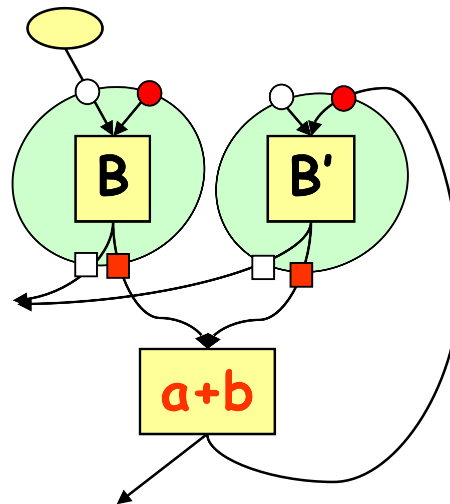


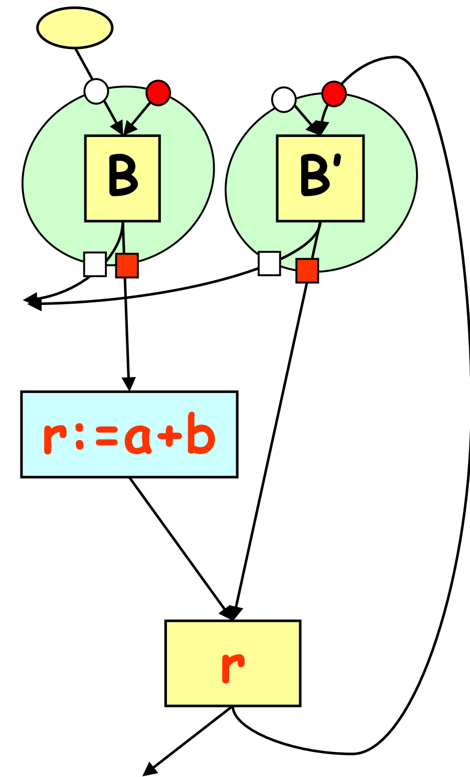
CMP: code motion preventing region



- duplicate only nodes that block code motion



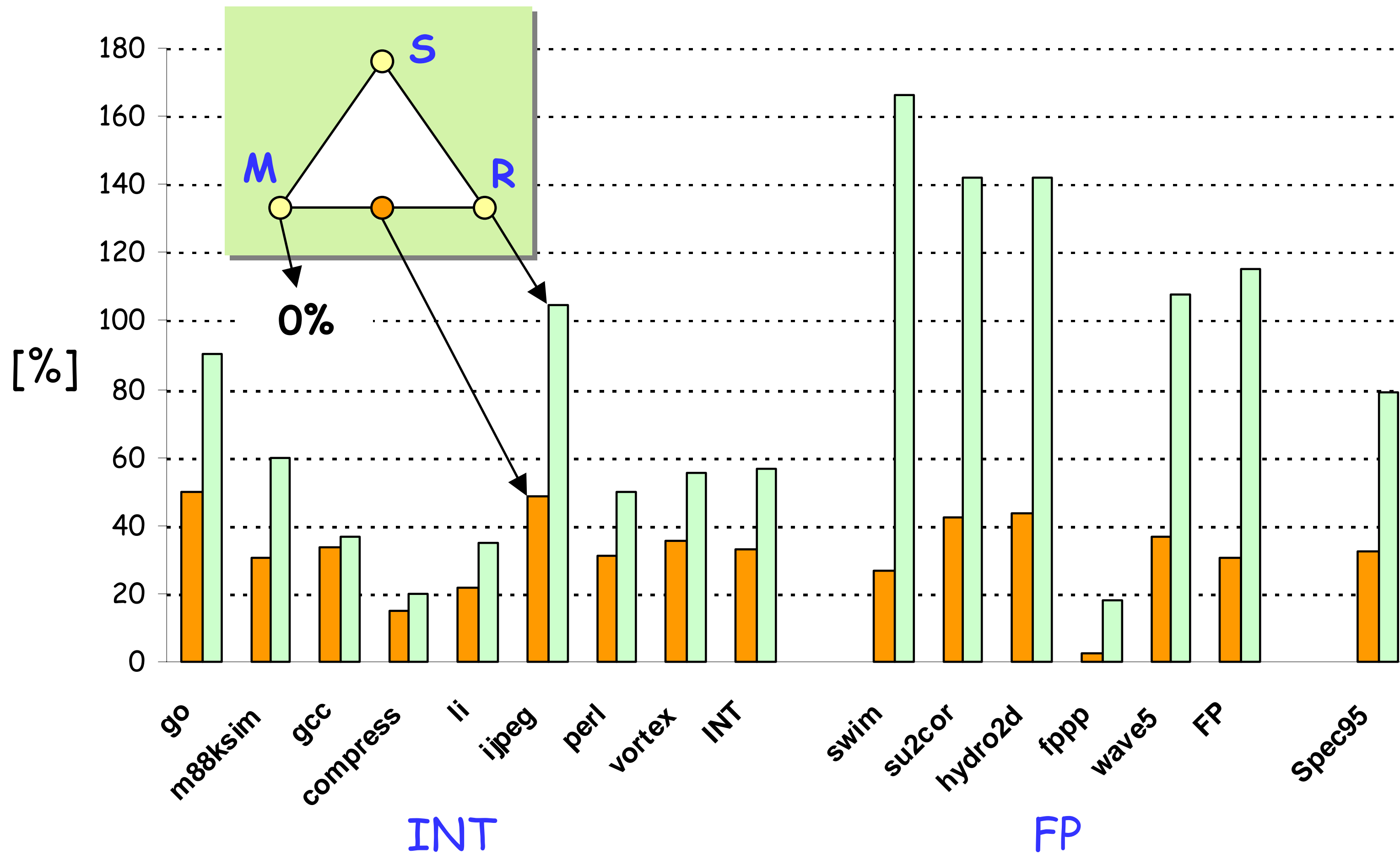




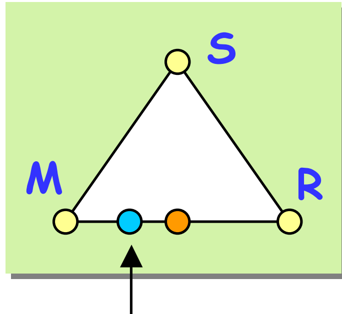
Code growth

R A T

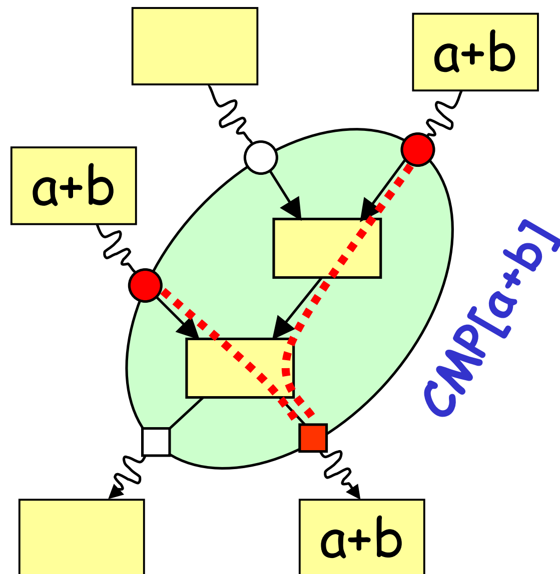
motion+restructuring vs. restructuring-only



How profitable is a CMP ?



- We need a profile!



path profile:

- yet another exponential factor

edge profile:

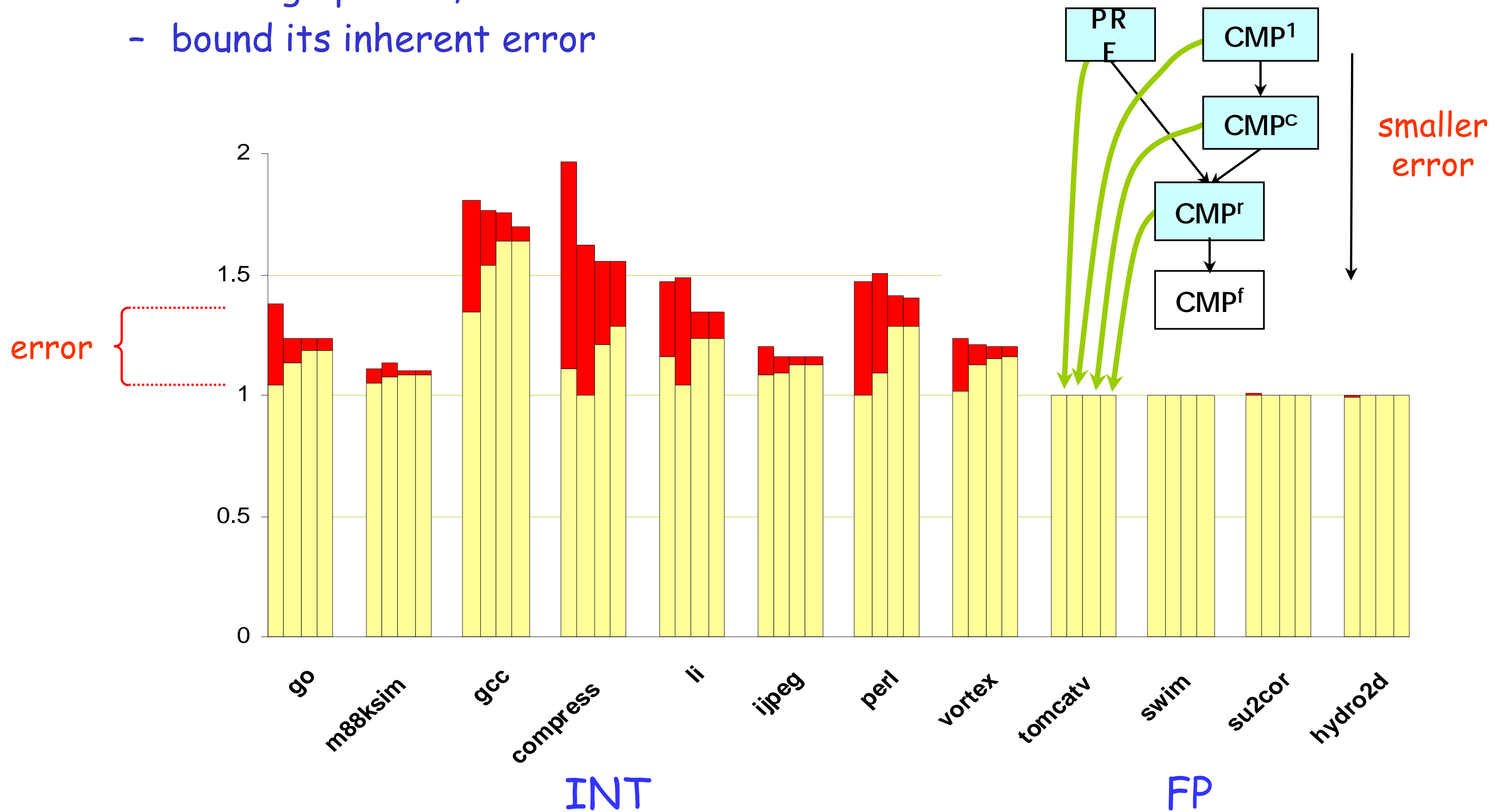
- imprecise, but commonly used
- can we use it with confidence?

edge profile < path profile

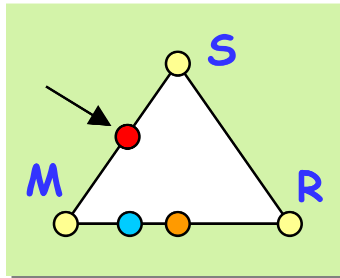
Estimators

Data-flow info + profile = dynamic amount of reuse

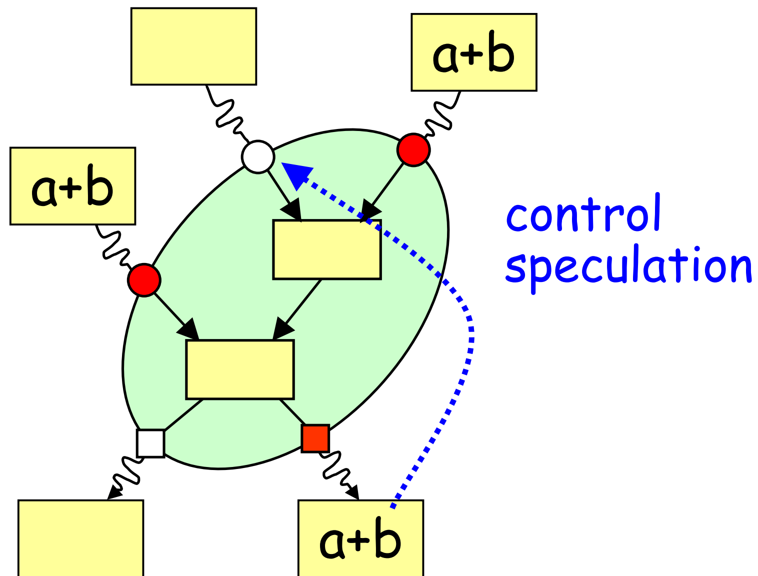
- use edge profile, but
- bound its inherent error



Motion + speculation



- relaxed model: can impair paths

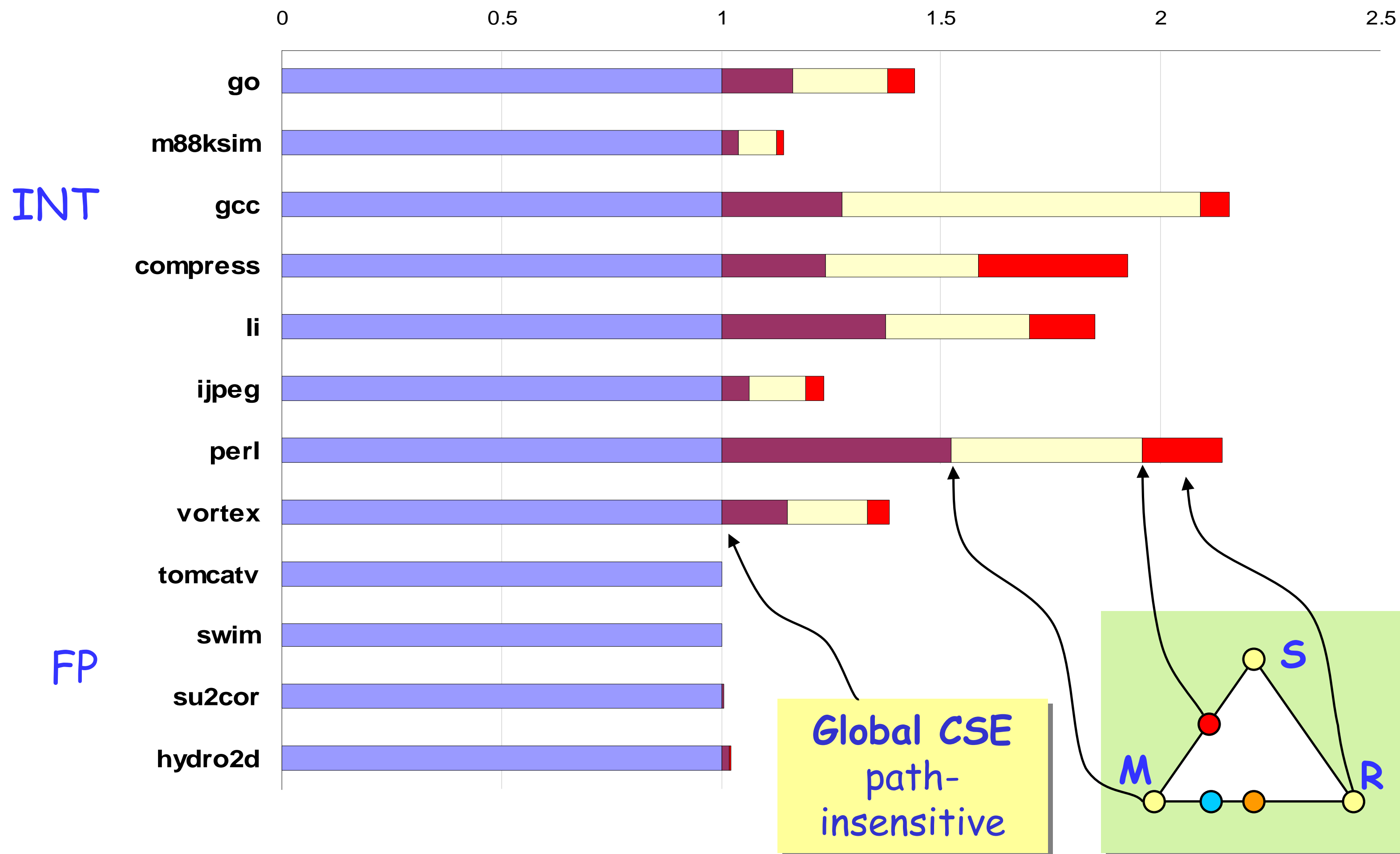


Speculate if:

removal - insertion > 0

edge profile = path profile

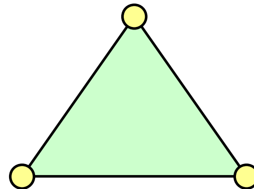
Relative removal power



The 3 transformations in concert

R
A
T

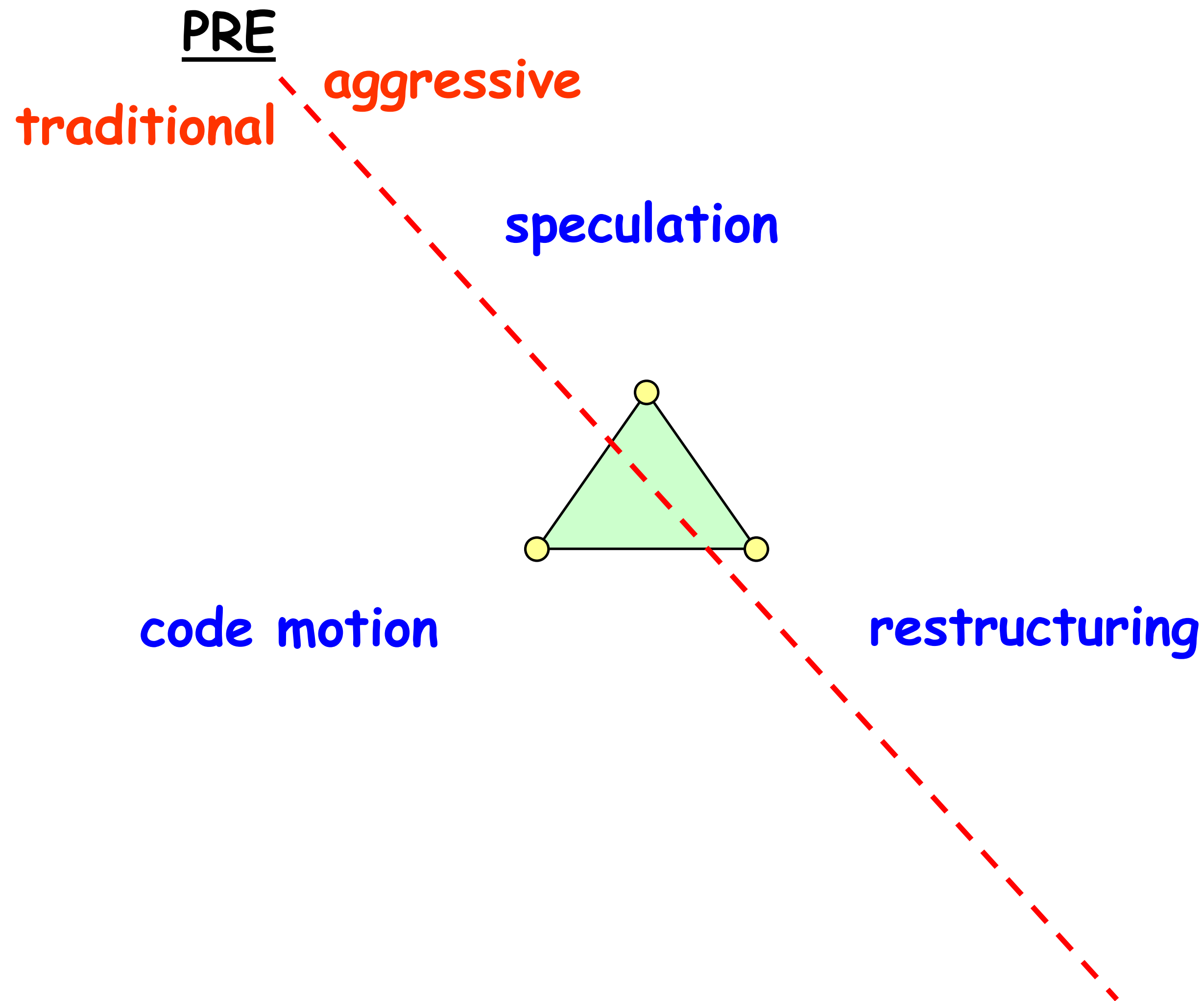
speculation



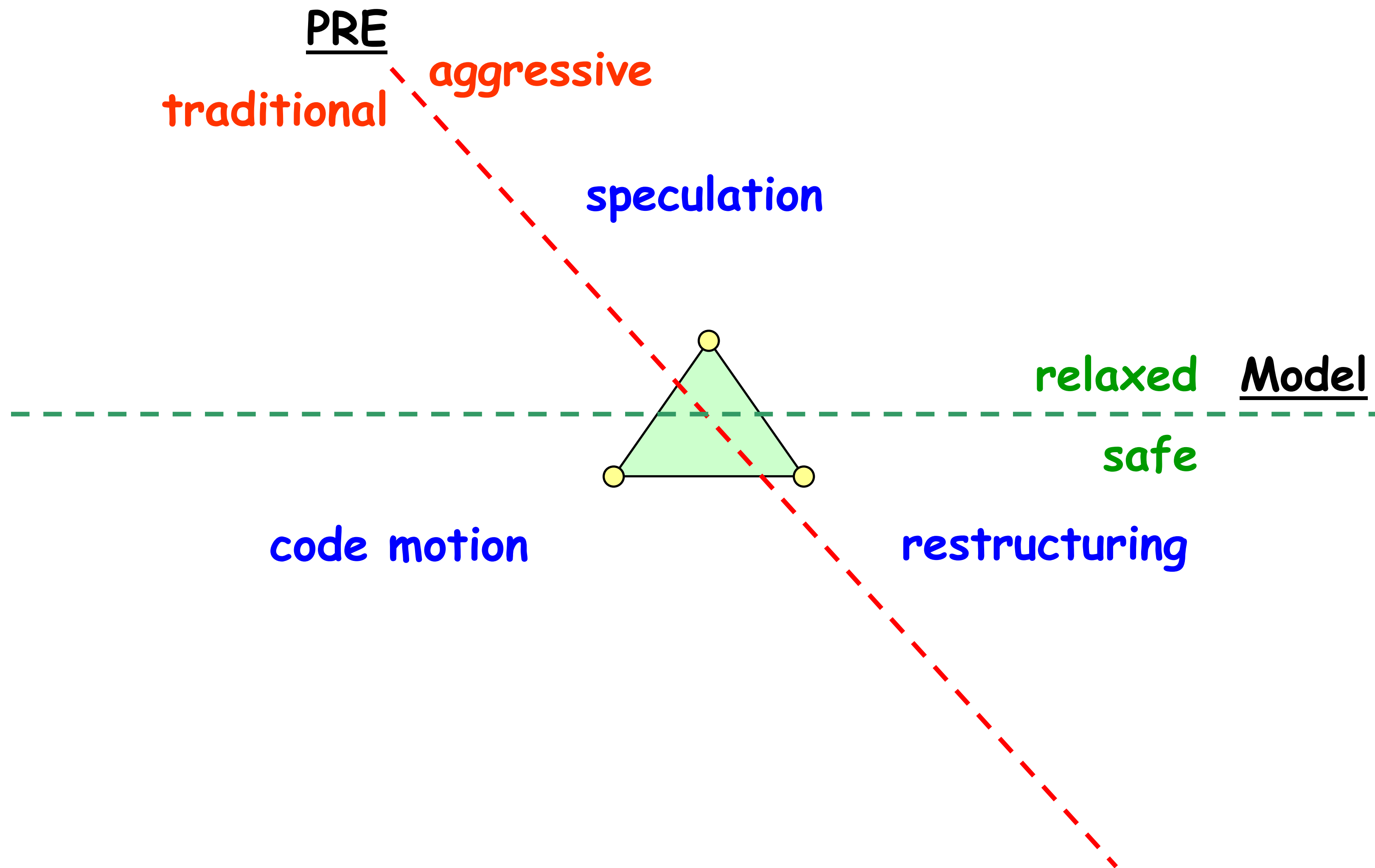
code motion

restructuring

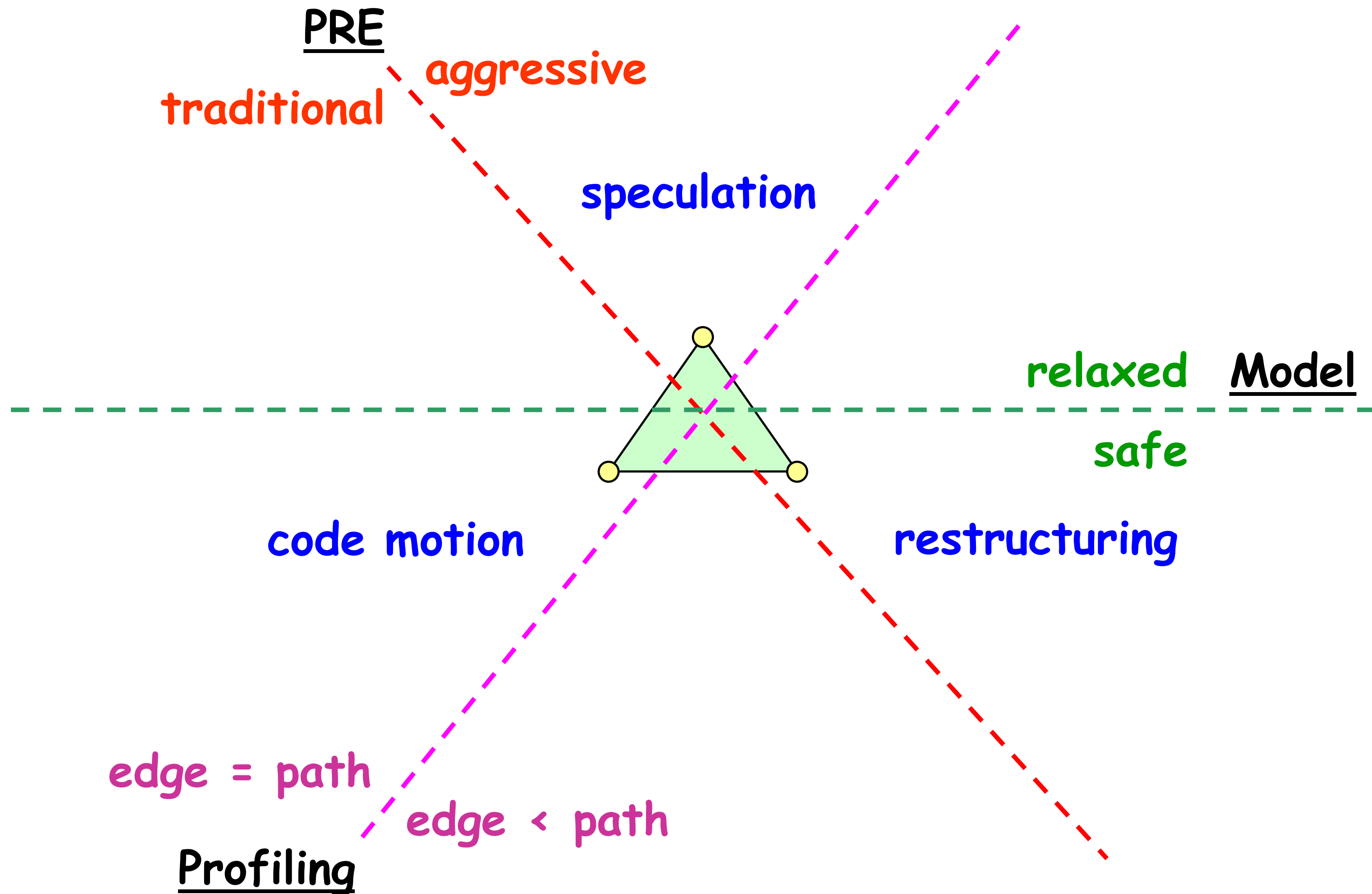
The 3 transformations in concert



The 3 transformations in concert



The 3 transformations in concert



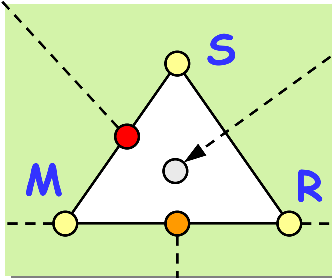
CMP: related work, contributions

R A T

Horspool, Ho '97
Lo *et al* '98
Gupta *et al* '98
heuristic

Maximal PRE at no code-growth
- optimal, using edge profile

Balanced PRE
- near-complete,
- small code growth



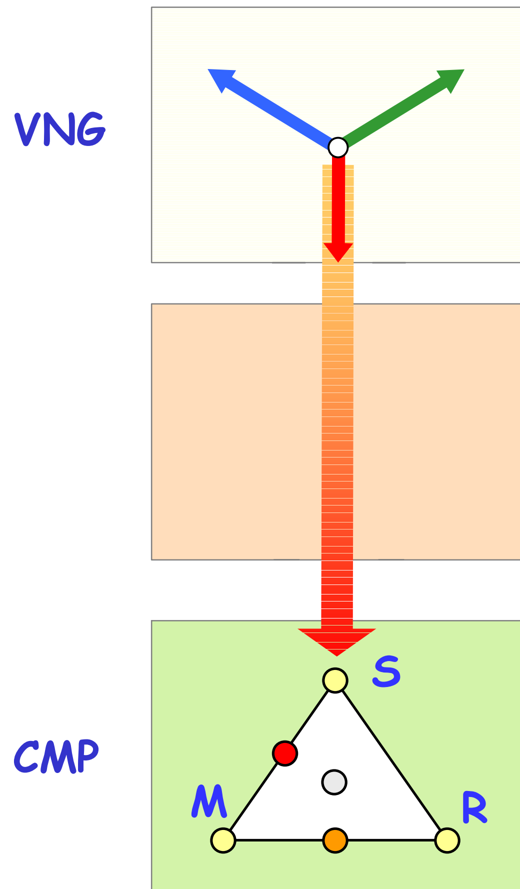
Morel, Renviose '79
Dhamdhere, ...
Knoop *et al* '92
"lazy" code motion

Wegbreit '75
Wegman '75
Steffen '96

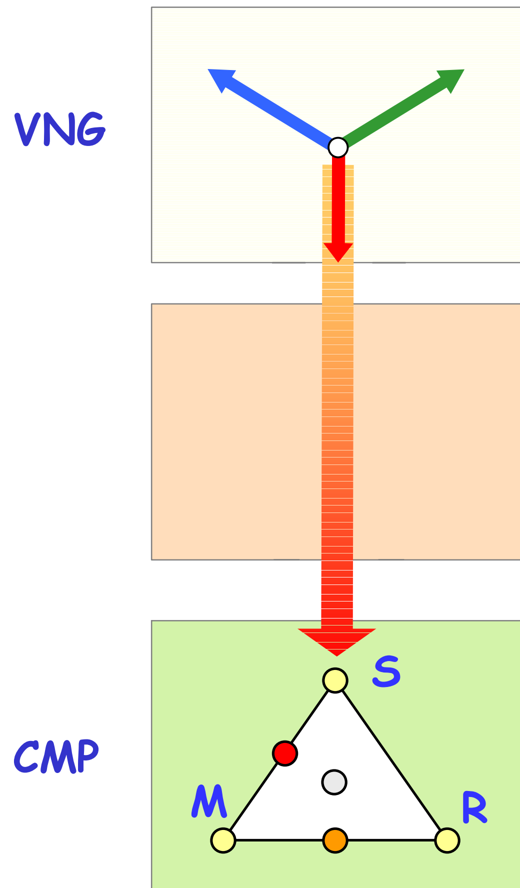
Intuitive formulation
- without being "lazy"

Complete PRE with minimal code-growth
- in the absence of profile

The Pathfinder framework



The Pathfinder framework



Pathfinder (etymology :)

framework stages:

VNG: 3 orthogonal techniques

CMP: 3 orthogonal techniques

hence

3x3 -- just like 4x4

so it's an

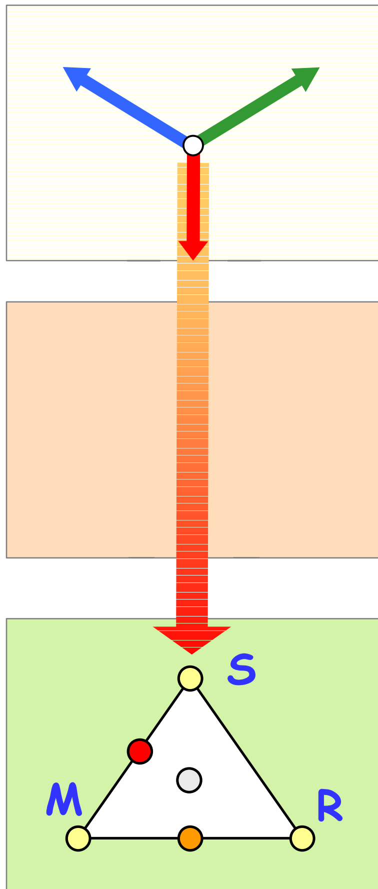
all-wheel-drive optimizer

called

(Nissan*) **Pathfinder.**

* research funded by NSF, HP, Intel.

Pathfinder vs Ammons-Larus



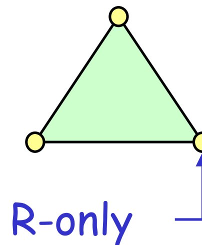
Avoid dilution

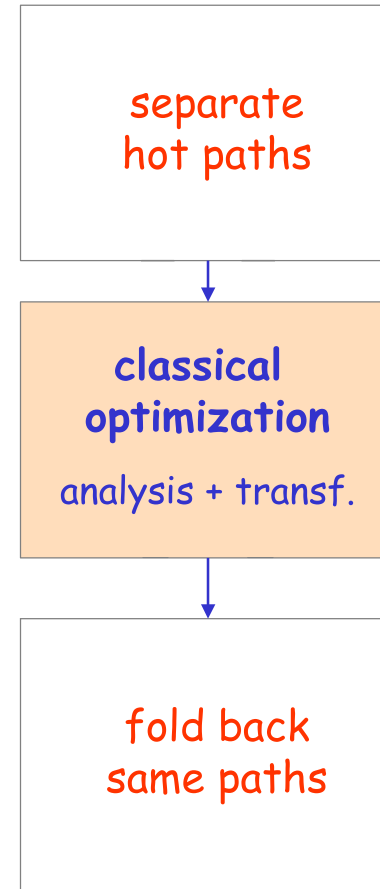
path-separation:

value-based vs profile-based

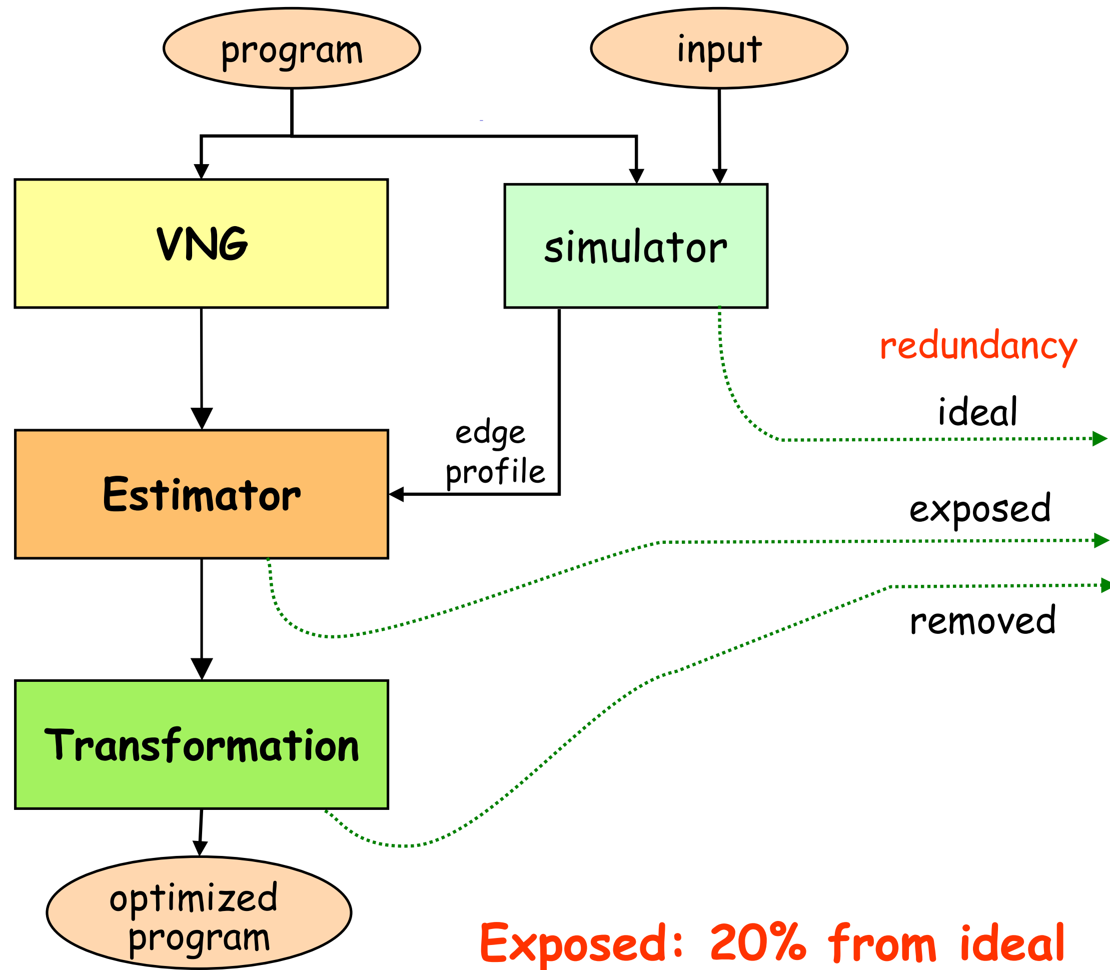
Transformation:

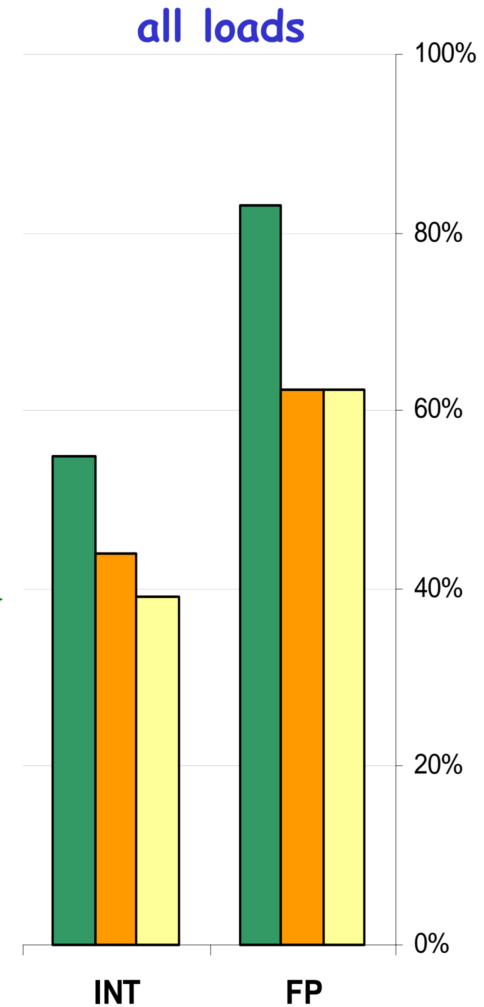
M+R+S vs R-only





Evaluation: load removal



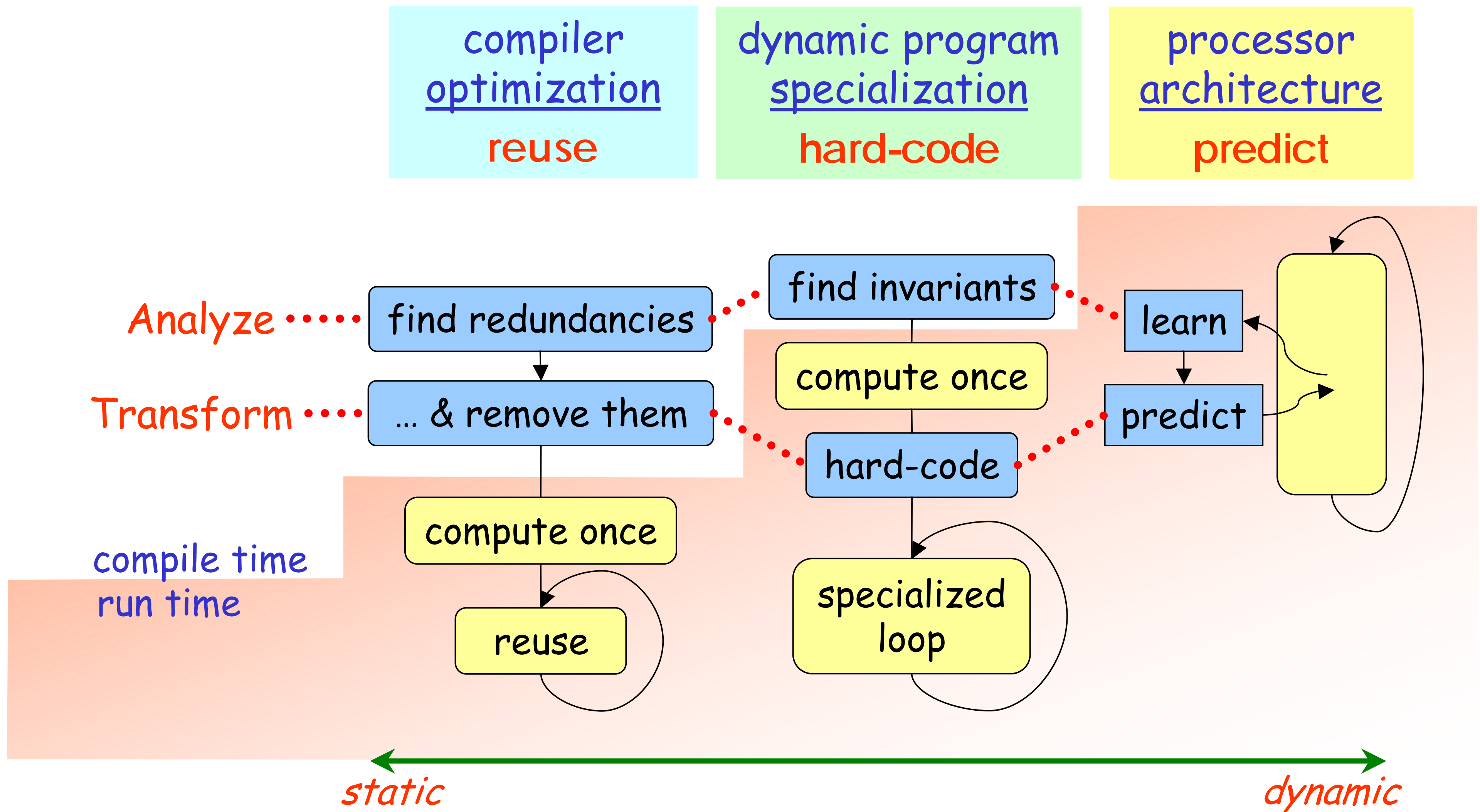


Exposed: 20% from ideal

Major results

- **Pathfinder:** a practical path-sensitive framework
  - 3x3 approach against exponential explosion
    - **VNG:** value threads formed on demand
    - **CMP:** maximize code motion, add speculation
- Profiling: guide the optimization
  - edge profile = path profile: when ?
  - edge profile < path profile: bound the error !
- Performance: load removal
  - 2x more path-sensitive reuse than Code-Motion PRE
  - 20% from its ideal performance
- Future plans: how to get the 20%, and more?
  - hybrid optimizations: hw+sw

The optimization spectrum



SW vs HW: a need for synergy ?