

Complete Removal of Redundant Expressions

Rastislav Bodík Rajiv Gupta Mary Lou Soffa

University of Pittsburgh

R Intro:

In this talk, we are going to deal with one the earliest problems of compiler optimization.

Notes:

The problem is to eliminate expressions that are redundant because they re-compute values that could instead be reused from other, identical expressions.

Conclusion:

Despite its long history, the problem has not been completely solved. So far, no practical algorithm that could remove from the program all redundancies was available.

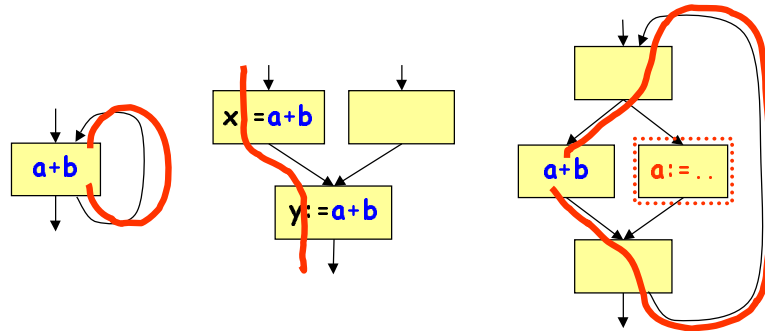
Transition:

Misc: imperative programs

Partially redundant expressions

2

Partially redundant: computed on some incoming path.



Goal: remove redundancy from "reuse" paths.

R Intro:

The problem are expressions that we call partially redundant.

Notes:

For these, opportunity to reuse a value exists only along some (but not necessarily all) paths.

The bottom $a+b$ can be optimized only along the left path.

This loop-invariant $a+b$ is also PR, the only difference is that value reuse originates in the expression itself.

The notion of partial redundancy includes conditionally executed loop invariants, and also conditionally killed invariants.

Conclusion:

What we have here are paths where the same value is computed repeatedly.

Our goal: completely remove this redundancy from each path, without changing program semantics. (by reusing the value).

The challenge: find a practical technique.

Transition:

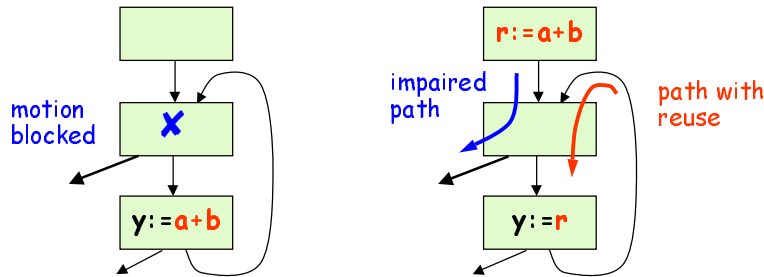
Practical algorithms for removing PR exist, but they are too conservative and therefore not complete: as they leave many opportunities for value reuse unexploited.

PRE via code motion

[Morel, Renviose '79] 3

Hoist partially redundant expression until it becomes:

- fully redundant \Rightarrow remove it
- not redundant \Rightarrow insert it



PRE fails on 70% of loop-invariants

Intro:

All algorithms for Partial Redundancy Elimination (or PRE) used in practice are improvements on the original MR technique. Ours is, too.

Notes:

The algorithms are based on code motion. If we keep hoisting the expression, then on each incoming path we get to an edge where the expression becomes either fully redundant or fully required.

PRE simply inserts the expression where it is needed and removes it where the reuse is available along all incoming paths.

The result: at the insertion point we initialize a temporary to carry the reused value. It's like a compensation code for changing PR into FR.

Unfortunately, this approach breaks down easily under more complex control flow. It's enough to change the repeat-until loop into a while loop and the insertion becomes control speculation. Inserting the expression here, we are speculating the control will enter the loop. If it does not, the blue path is impaired and the optimization is counterproductive. So, PRE disables this speculation as unsafe.

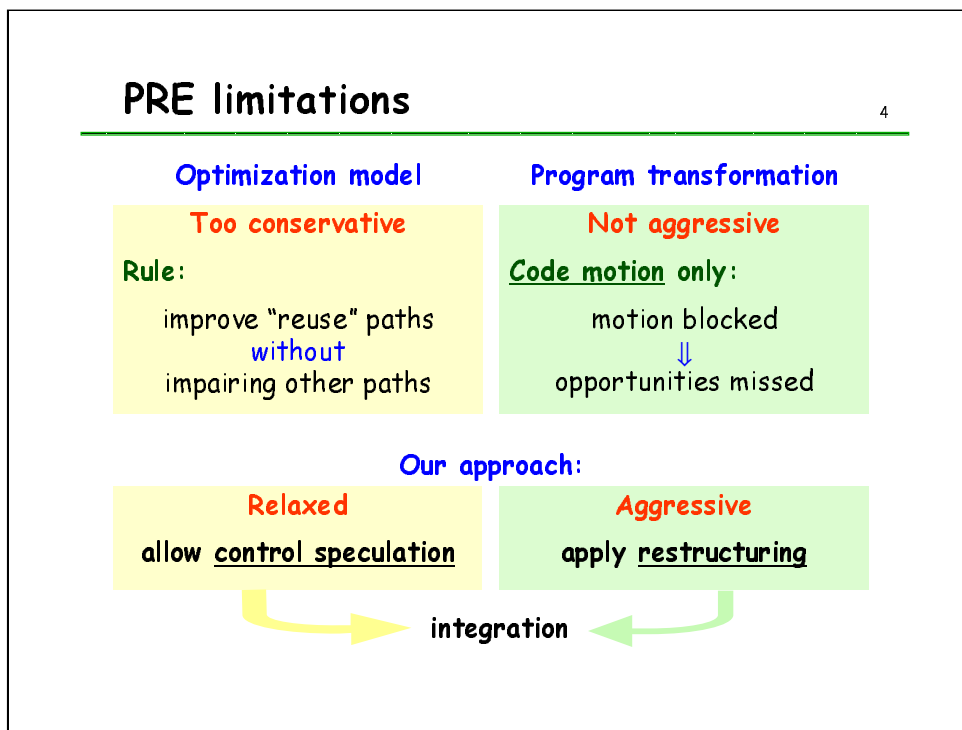
Conclusion:

As a result, 70% of what we classified as LI cannot be fully hoisted.

Transition: Why traditional PRE misses so many opportunities, it all boils down to two independent reasons.

PRE limitations

4



Intro:

Notes:

The first reason is the overly safe optimization model: to guarantee that the program will be optimized and never impaired, PRE never increases the number of computations any path, which corresponds to disabled control speculation.

Second, code motion is the only transformation. Given the safe model, it fails to deal with motion obstacles presented by the control flow.

To achieve a complete, practical PRE, we attack both problems. First, we supplement CM with a more aggressive trafo, CF restructuring. This trafo duplicates some CF paths to enable the desired code motion. The cost is code growth.

Second, we relax the safe model and allow speculation. Because speculation impairs, we don't consider it complete. But there is no code growth. We use profile to navigate speculation for optimal results.

Conclusion:

The two strategies are orthogonal and we integrate them to get near-complete PRE at affordable code growth.

Transition:

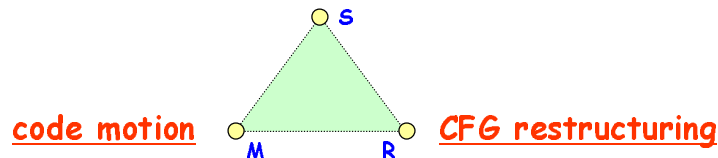
Altogether, there are three trafos.

Three PRE transformations

5

- impairs some paths
- no code growth

control speculation



- misses opportunities
- no code growth

- complete
- code growth

Intro:

Each of the three trafos has different properties.

Notes:

CM and speculation only move expressions around, so they cause no code growth as restructuring does. They are very economical.

While CM can be blocked and misses opportunities, restructuring can completely remove all redundancies, but is costly. Speculation is tricky, because we need to know the profile and how to use it.

Conclusion:

It turns out that a PRE algo can be a combination of all three trafos. The triangle is an algorithm design space. Depending on where the algorithm is in the design space, it has different biases and different properties.

Trans:

In this talk, I will show how to integrate and how to control the properties.

Misc: power corrupts!

Talk outline

6

~ Motion + restructuring

safe model complete PRE/minimal growth no profile
reduce code growth near-complete PRE profile

~ Motion + speculation

relaxed model maximal PRE/no growth

~ Dynamic benefit

large-scale computation

~ Related work & summary

Intro: Notes:

We start without a profile, using the traditional model. I will show how to restructure only enough to do the desired code motion.

Then we switch to world where profile is available and use it to reduce the code size by eliminating unprofitable restructuring.

Then we switch to the relaxed model and use the profile to find optimal speculation.

Because knowing dynamic benefit of the PRE is critical for our profile-directed approach, I will show how to efficiently compute the benefit on large programs.

Conclusion:

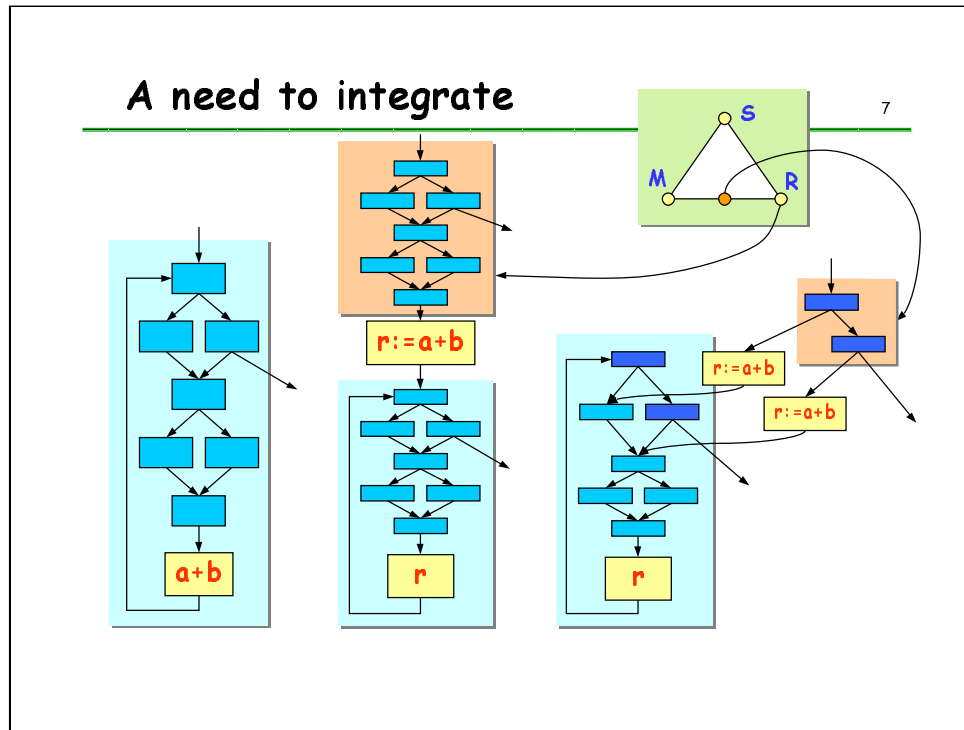
This talk will sketch a few PRE algorithms. First, a complete PRE with minimal restructuring. Then, using profile, a near-complete PRE at reduced code growth. In the process, I will show how to obtain maximal PRE under no restructuring.

Transition:

We start with safe model and no profile. We want to combine CM+R.

R alone can remove all redundancies. All we need to do is duplicate paths where reuse is possible. This will split partial redundancy into full redundancy and no redundancy and the elimination becomes trivial. But, a lot of restructuring is unnecessary because CM can do part of the job. So, we are looking for a balance point where we restructure enough to get a complete PRE but not more than necessary.

And, in our approach, necessary means just enough to enable code motion.



Intro:

This example shows the difference between pure restructuring and integration.

Notes:

$a+b$ is LI but it cannot be hoisted from the loop with traditional PRE because this path would be impaired.

R alone will peel one iteration to separate F red from No red. It leaves $a+b$ where needed, and uses its value where it was redundant.

Instead, our algorithm for complete PRE only restructures these two nodes. This is sufficient for $a+b$ to be hoisted from the loop.

Conclusion:

So, we only duplicated the path preventing CM. This is the minimum set of nodes to enhance traditional CM PRE to become complete.

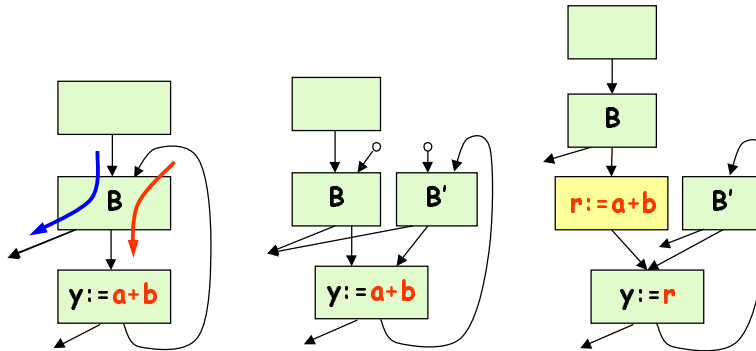
Transition:

To identify what should be duplicated, we define a special program region.

CMP: code motion preventing region

8

👉 duplicate only nodes blocking code motion



Intro

The region contains all nodes that prevent desired code motion. So, we call the region a CMP, for code motion preventing region.

Notes

The obstacles exist precisely where any reuse path meets with a path that is expression-free. We cannot hoist from the reuse path because the free path would immediately be impaired. The CMP here is only single block **B** here but in general it can be arbitrary multi-entry multi-exit subgraph. The entries and exits of the CMP are always edges.

Once we have the CMP region, we duplicate it. Exit edges are duplicated and entries are attached so that each distinct path uses one copy. There are always only two copies, one for reuse paths, one for free paths. After such restructuring, paths become separated and hoisting from the loop is possible into this edge.

Conclusion

The program after hoisting and insertion, no redundancy remains, PRE is complete.

Transition

The task now is how to algorithmically identify the CMP, for a given expression.

Identifying CMP regions

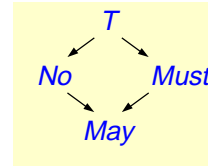
9

AVAIL - availability

incoming paths

ANTIC - anticipability

outgoing paths



$AVAIL[n, a+b] = \begin{matrix} \text{Must} & & \text{all} \\ \text{No} & \text{available along} & \text{no} \\ \text{May} & & \text{some} \end{matrix}$ paths

CMP[a+b] = set of nodes n s.t.

$AVAIL[n, a+b] = \begin{matrix} \text{May} \\ \text{impaired path} \dots\dots \\ \text{"reuse" path} \dots\dots \end{matrix} \wedge \begin{matrix} \text{ANTIC}[n, a+b] = \begin{matrix} \text{May} \\ \text{No} \\ \text{Must} \end{matrix} \dots\dots \dots\dots \dots\dots \end{matrix}$

Intro: We identify CMPs by solving the two data flow problems that are also used in traditional PRE.

Notes:

availability: a forward problem, determines if computed along all incoming paths. Anticipability: backward, if will be computed along each outgoing path. The first is about the past, the other about the future.

Traditionally, boolean: true only if computed along all paths. We need to refine into three values. We say that a+b is Must-available at node n if it is computed on all incoming paths. No means no paths and May means strictly some paths. So, May implies there is a No paths and a Must paths. Anticipability is defined similarly.

Once we have solutions to these problems, the CMP for an expression is the set of nodes where both solutions are May. To see why: May implies both Must and No. Must means that the expression is computed before and after n on some paths, this is a path with reuse. No means there is a path free of the expression before and after n, this path would be impaired.

Summary:

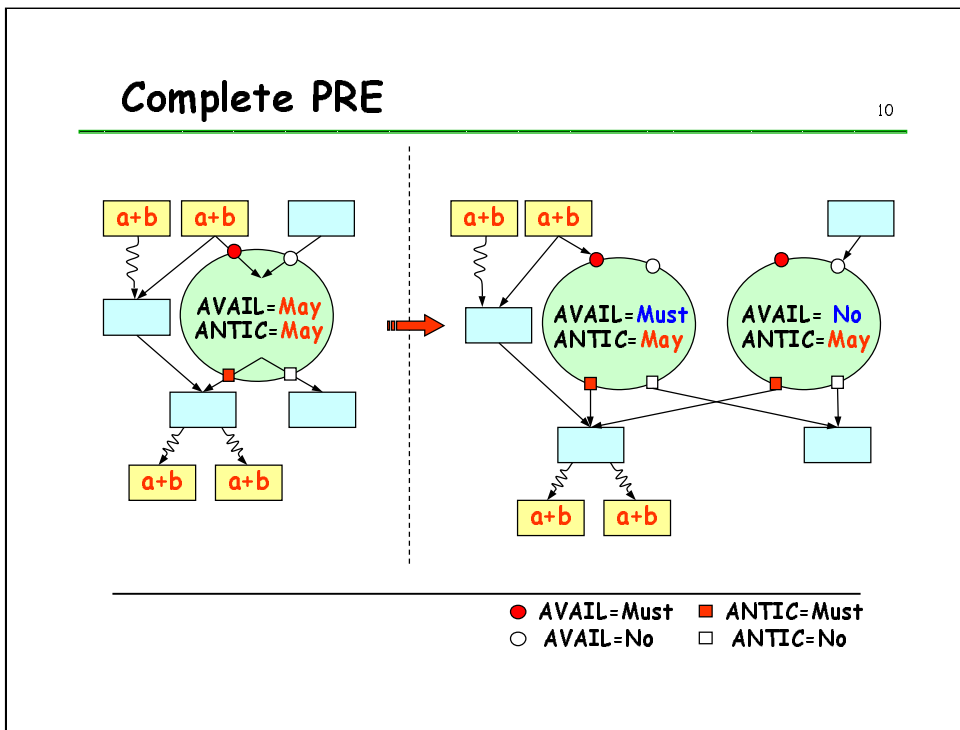
So, we are using the same problems as other PRE algorithms, only the lattice is refined.

Transition:

once we have the solution, can restructure

Complete PRE

10



Intro:

The CMP is where both problems have May solution.

Transition:

This is really where the two kinds of paths meet. CMP localizes adverse effects of control flow on code motion, because where motion is possible, (like here), the nodes are not in CMP.

Two copies are created, one is used by the entry edges where value is always available, the other where it's never available. This way, availability in the region changes to Must, and No, and the May May condition disappears. Code motion is possible into this edge.

Notice that this block is not split. It would be with pure R approach. We are using here CM as much as possible.

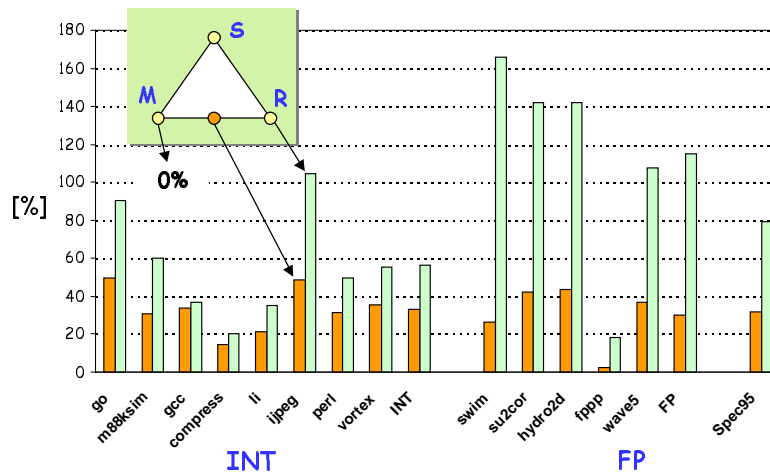
Transition:

This experiments show the benefit of integration.

Code growth

11

motion+restructuring vs. restructuring-only



Intro:

This is for spec95.

Notes:

Overall, integration reduces the code growth about 2.5 times when compared to pure restructuring. Less for integer benchmarks because complicated CF causes CM fail more often.

Conclusion:

So, we need about 30% of code growth to get complete PRE. This may be more or less depending what expressions we are eliminating. If we add loads of literals, as is the case with the experiments in the paper, it will jump to 60%, the ratio between pure R remaining the same.

1. Duplicate only profitable CMP's

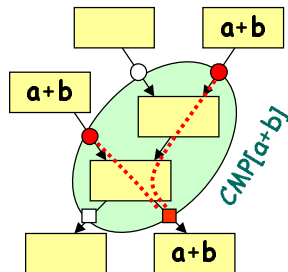
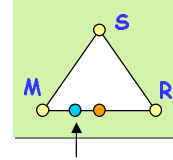
12

profitable

$\text{benefit} > c.\text{size}(\text{CMP})$

benefit

dynamic amount of eliminated expressions



Computing the benefit

edge profile

probability of $a+b$ being available

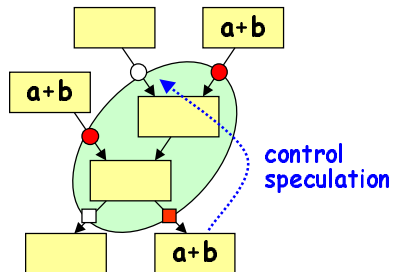
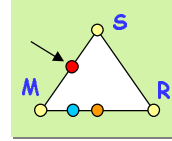
path profile

sum of **Must-Must** paths freq's

2. Motion + speculation

13

- Relaxed model,
- hoist into No-AVAIL entries,
- ✗ harmful without profile info.

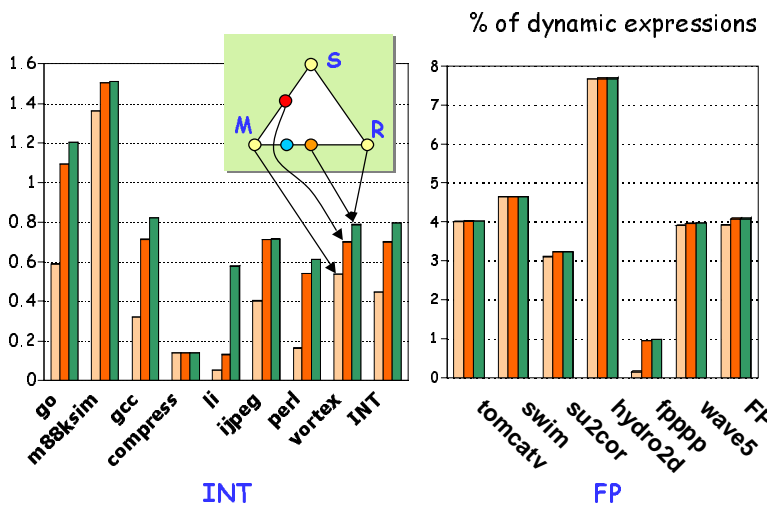


Computing the benefit
edge profile

$$\begin{array}{l} +\text{freq}(\blacksquare) \text{ removal} \\ -\text{freq}(\circ) \text{ insertion} \\ \hline +\text{freq}(\blacksquare) - \text{freq}(\circ) \end{array}$$

Partial redundancy removed

14



Computing benefit on large scale

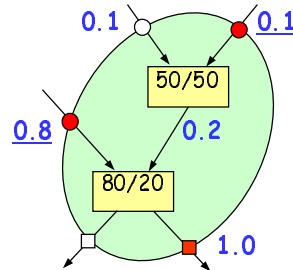
15

Goals:

- focus on hot paths, but
- drop cold nodes only within allowed imprecision.

Approximate frequency analysis

- extension of [Ramalingam '96]
- propagate contribution weight
- *demand* interval analysis



Intro

goal: compute freq df info but approximate in order o reduce analysis cost

Notes

approximate by dropping queries at cold nodes, assuming most conservative assumption (UNDEF in case of correlation)

query propagation wave: traditional vs. approximate freq analysis

need to know contribution weight of each node

seems simple but to do precisely in programs with loops a new kind of demand analysis is needed, based on interval analysis

Conclusion

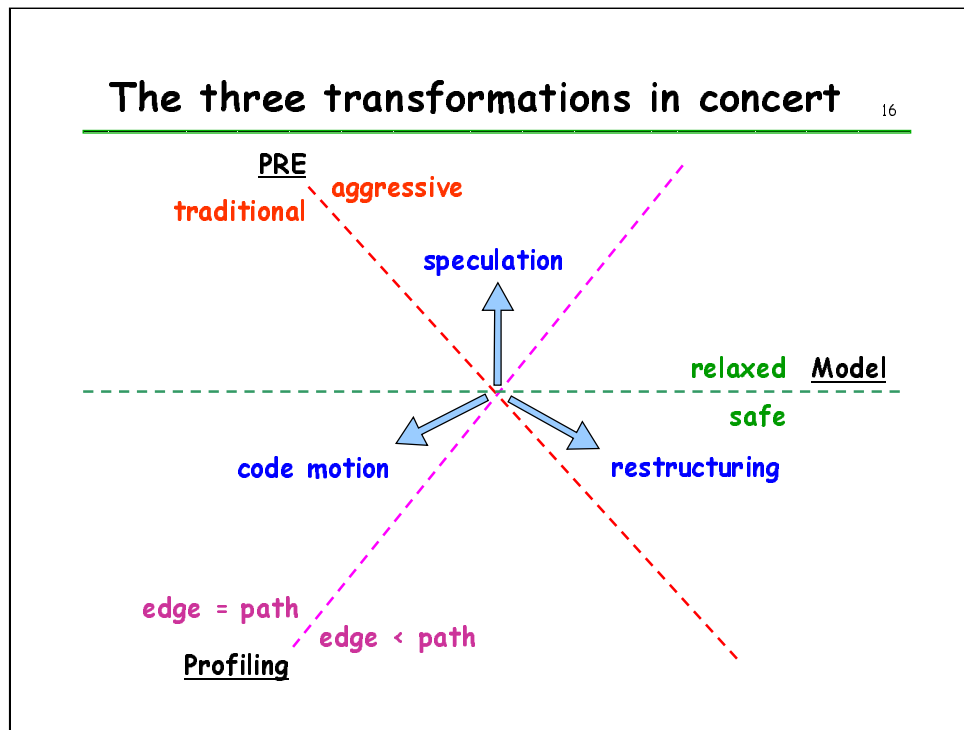
region based on hot paths may differ from hot region based on opportunities

Transition

to see such analysis is important, look at graphs (need to know benefit before applying the transformation)

The three transformations in concert

16



Intro:

ready to summarize

Notes:

PRE, at least the MR style, can be carried out with 3 transformations, each with distinct properties

Conclusion:

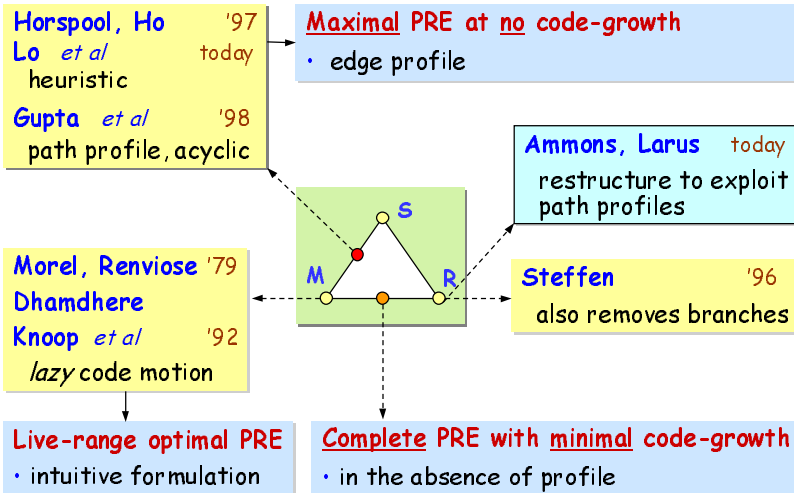
Transition:

Misc:

you have to draw the line somewhere

Related work & summary

17



CMP: a versatile PRE abstraction

18

localizes effects of control flow

only examine:

- CMP paths (restructuring)
- CMP entries/exits (speculation)

small effective size

- small number of entries, executed paths
- combinatorial algorithms become practical