RETROSPECTIVE:

Complete Removal of Redundant Expressions

Rastislav Bodík
EECS, University of California, Berkeley
and
IBM T.J. Watson Research Center
bodik@cs.berkeley.edu

Rajiv Gupta
Department of Computer Science
University of Arizona
Tuscon, AZ
gupta@cs.arizona.edu

Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA
soffa@cs.pitt.edu

Background

Invented in 1979, Partial Redundancy Elimination (PRE) received renewed interest in the 90's, thanks to multiple industrial efforts crafting optimizing compilers for upcoming VLIW processors, and also thanks to new efficient PRE algorithms, most notably the Lazy Code Motion [*24*].

Although PRE was ready for industrial implementation, some of its potential was left unexploited, due to the limited transformational power of code motion underlying the standard PRE. We became interested in developing complete PRE when our experiments revealed that standard PRE failed to eliminate about 70% of loop invariant expressions—a disappointing result given that PRE was being implemented as a generalization of loop-invariant code motion.

At the time, it was already known that redundant expressions could be removed completely, in fact with a simple algorithm that restructured the control flow graph via path duplication [*30*]. In practice, however, the resulting code growth limited the use of restructuring to controlled ad hoc transformations like do-until conversion, which in some cases improved the standard PRE, but were inadequate in general.

The question that initiated our research thus was how to integrate code motion with restructuring in a PRE algorithm that would resort to restructuring only when necessary, i.e., when the (preferred) code motion fails. The result, we hoped, would subsume ad hoc transformations by performing custom restructuring needed to enable code motion.

Besides restructuring, a promising technique for improving standard PRE was control flow speculation. First applied in instruction scheduling, this profile-guided technique broke the very guarantee that made standard PRE incomplete: it hoisted instructions along hot program paths into predecessor basic blocks without restriction, thus enabling more code motion, but potentially de-optimizing the program by executing some instructions more often than in the original program. Two algorithms for speculative PRE were known [5, *20*], but a few questions were left open, particularly how to perform optimal speculative PRE, and what kind of profile needs to be used for the purpose.

The questions that we wanted to answer thus were:

1. *Completeness.* Is it possible to eliminate all redundant expressions without the excessive code bloat of code duplication? Is the benefit of completeness worth the additional implementation complexity?

2. *Profiling.* Which trade-offs in PRE should be profile-guided? Clearly, profiling should be able to prioritize optimization of hot paths at the expense of cold paths, but what are suitable abstraction for guiding the trade-offs and for integrating the transformations?

3. *Profile format.* To guide PRE (a path-sensitive optimization), a path profile seems necessary. Could the cheaper edge profile be used instead? When are the two profiles equally accurate?

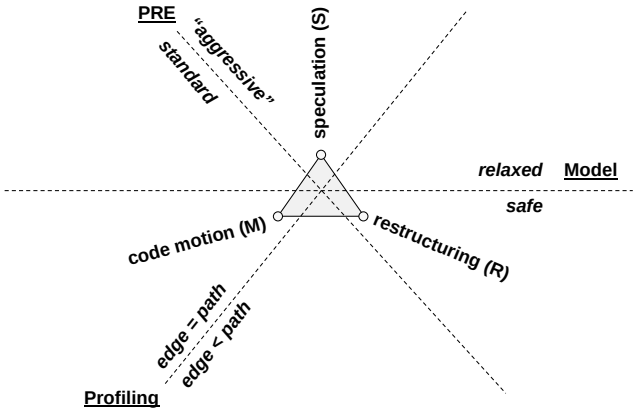
20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

Evolution of the idea

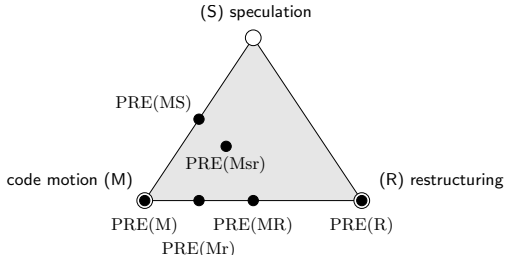
At the outset, we felt confident only about solving the first question. Guided by the intuition developed in our work on dead-expression removal [2], we were able to define the CMP region that decomposed the program into parts optimizable via code motion and parts optimizable via restructuring. As we came to realize only later, the CMP region is a refinement of the Use Region from [2]: instead of computing a single dataflow problem of its precursor, it is based on two problems, one forward and backward, each on a richer lattice, which is what allows it to reduce restructuring. The CMP region was also instrumental in answering the other two questions, but not until the follow-up paper [4], which presented the family of estimators. To provide a big-picture perspective, we summarize here the connections among the relevant results, leveraging the insights gained after these papers were published.

Conclusions, in retrospect

The CMP-based algorithms in the paper equip the standard PRE based on code motion (M) with the power of two additional transformations: control-flow-graph restructuring (R) and control-flow speculation (S). One can view the new PRE algorithms as more aggressive, shown by the two PRE planes in the figure below. Regarding the style of the optimization, both code motion and restructuring operate under the safe PRE model, in which no program path is allowed to execute more expressions after optimization. In contrast, speculation requires relaxing this model, by allowing paths to be impaired with new expressions, typically under profile-guidance. Regarding the necessary profiling accuracy, when code motion is combined with speculation (which is in fact a form of code motion), an edge profile is as accurate as the path profile. In contrast, when restructuring is profile-guided, path-profile is more accurate than edge profile [4].



The three transformations can be combined by means of the CMP region to produce several algorithms, as shown in the figure below. **PRE(M)**, which achieves the same transformation as [*24, 26*], uses only code motion, and finds the best optimization when execution frequencies are not known. **PRE(MS)** extends code motion with speculation, but uses the speculation carefully, to maximize the optimization for a given profile. (**PRE(R)**, the algorithm in [*30*], can also be classified in the triangular space of transformations. It removes all redundant expressions by restructuring, and not moving any code.) **PRE(MR)** also removes all redundant expressions, but duplicates basic blocks only when code motion fails (it does not minimize code duplication, though; as was later shown by Melski [6], that problem is NP-hard). **PRE(Mr)** and **PRE(Msr)** use a profile to trade off some optimization benefit for code-growth reduction.



Results of interest to compiler engineers

After experiments with PRE for scalar replacement in the follow-up paper [4], we drew the following conclusions that may be of interest to those who want to implement PRE.

- *How powerful is the standard (code-motion-only) PRE?* In our experiments with integer benchmarks, PRE(M) eliminated 50% expressions more (dynamically) than common-subexpression elimination—the most powerful path-insensitive transformation. In contrast, complete PRE (PRE(MR) or PRE(R)) eliminated roughly 100% more expressions (see Figure 6.20 in [1]), suggesting that extensions to standard PRE should be implemented.
- *How close is speculative PRE to complete PRE?* Somewhat surprisingly, PRE(MS) removes nearly all redundant expressions. It is also easier to implement (because no restructuring is needed), and hence provides a very attractive alternative to the complete PRE(MR).
- *What if speculation cannot be used?* There are situations when PRE cannot use speculation, e.g., due to side effects of redundant expressions, such as exceptions. In such situations, restructuring must be used. For simple loops, do-until conversion will do; for more complicated control flow, the PRE(MR) algorithm will perform the necessary restructuring.
- *Is path profile needed to guide restructuring?* When performing restructuring, one may want to determine the optimization benefit of code duplication. In theory, path profiles are needed for accurate computation of the benefit. In practice, the edge-profile-based estimators in [4] incur very small error.

The future of redundancy elimination

The trend of programming with components may move the focus of program optimization research towards high-level transformations, such as selection of data structures or rewriting a client code from one API to another. Since API calls can be modeled as more complex expressions, PRE may play a role in such transformations. A relevant open problem: can speculation be used to simplify the complex interprocedural code-motion PRE?

Profile-based demand-driven dataflow analysis

Section 5 of the paper presents an analysis that computes the frequency of dataflow facts (like [*27*]), but conservatively self-terminates as soon as sufficient (frequency-wise) facts have been collected. Building on the inter-procedural analysis in [*10*], where early termination by preventing exploration of some program paths facilitated scalability, the algorithm in the paper seeks to provide a guarantee that unexplored paths did not contain important optimization opportunities. In retrospect, the alternative approach of restricting the analysis to hot program paths, as was done in in [*4, 19, 20*], may lead to an equally practical solution. It should be noted that although we did not experimentally confirm the usefulness of Section 5, we later successfully used its idea in a simplified fashion in [3].

REFERENCES

(Italicized citations refer to the paper. Note that [*24, 4*] also appear in this Selection.)

[1] R. Bodik. *Path-Sensitive Value-Flow Optimizations*. PhD thesis, University of Pittsburgh, November 1999.

[2] R. Bodik and R. Gupta. Partial dead code elimination using slicing transformations. In *ACM SIGPLAN PLDI'97*, pages 159–170, June 1997.

[3] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *ACM SIGPLAN PLDI'00*, pages 321–333, June 2000.

[4] R. Bodik, R. Gupta, and M.L. Soffa. Load-reuse analysis: Design and evaluation. In *ACM SIGPLAN PLDI'99*, pages 64–76, May 1999.

[5] R.Nigel Horspool and H.C. Ho. Partial redundancy elimination based on a cost-benefit analysis. In *Proceedings of 8th Israeli Conference on Computer Systems and Software Engineering*, 1997.

[6] D. Melski. *Interprocedural Path Profiling and the Interprocedural Express-lane Transformation*. PhD thesis, University of Wisconsin–Madison, 2002.