Space-Sensitive Partial Redundancy Elimination

Oliver Rüthing, Jens Knoop, and Bernhard Steffen

Universität Dortmund, Baroper Str. 301, D-44221 Dortmund, Germany
{ruething,knoop,steffen}@ls5.cs.uni-dortmund.de

1 Motivation

Partial redundancy elimination (*PRE*) is one of the most important and widespread optimizations in compilers (cf. [3]). Intuitively, it aims at avoiding unnecessary recomputations of values at run-time. Technically, this is achieved by storing the value of computations for later reuse in temporaries. Traditionally, PRE-techniques focuses on reducing the number of computations performed at run-time to a minimum while keeping the lifetimes of introduced temporaries as small as possible in order to avoid unnecessary register pressure. These techniques, however, are not space-sensitive: They can cause code replication without providing any control about this, which limits their adequacy for application areas where code size is crucial as e.g. embedded systems (cf. Figure 1 for illustration).

In this article (cf. [4]), we add a third dimension to partial redundancy elimination by considering *code size* as a further optimization goal in addition to the more classical consideration of computation costs and register pressure. This results in a family of *sparse* code motion algorithms coming as modular extensions of the algorithms for *busy* and *lazy code motion* (cf. [1, 2]). Each of them optimally captures a predefined choice of priority between these three optimization goals, e.g. code size can be minimized while (1) guaranteeing at least the performance of the argument program, or (2) even computational optimality. Each of them can further be refined to simultaneously reduce the lifetimes of temporaries to a minimum. These algorithms are well-suited for size-critical application areas like smart cards and embedded systems, as they provide a handle to control the code replication problem of classical code motion techniques. We believe that our systematic, priority-based treatment of trade-offs between optimization goals may substantially decrease development costs of size-critical applications: users may "play" with the priorities until the algorithm automatically delivers a satisfactory solution.

References

1. J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'92*) (*San Francisco, California*), volume *27*,7 of *ACM SIGPLAN Notices*, pages 224 – 234, 1992.

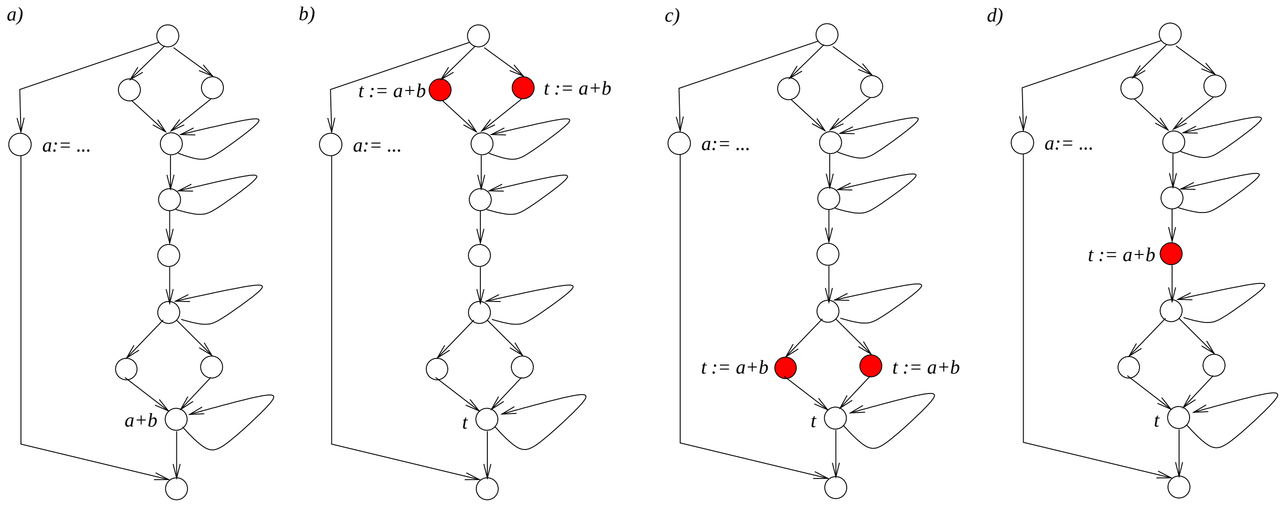

Fig. 1. (a) A program with loop invariant computation of $a + b$. (b) Busy and (c) lazy code motion lead in this example both to code replication. (d) Computationally optimal program transformation particularly free of code replication.

2. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.
3. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96 – 103, 1979.
4. O. Rüthing, J. Knoop, and B. Steffen. Sparse code motion. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 2000*) (*Boston, Massachusetts*), pages 170 – 183. ACM, New York, 2000.