Optimization of Straight–Line Code Revisited

Thomas Noll and Stefan Rieger
{noll,rieger}@cs.rwth-aachen.de

Software and Verification Group
RWTH-Aachen University
D-52056 Aachen, Germany

Abstract

We study the effect of an optimizing algorithm for straight–line code which first constructs a directed acyclic graph representing the given program and then generates code from it. We show that this algorithm produces optimal code with respect to the classical transformations known as Constant Folding, Common Subexpression Elimination, and Dead Code Elimination. In contrast to the former, the latter are also applicable to iterative code containing loops. We can show that the graph–based algorithm essentially corresponds to a combination of the three classical optimizations in conjunction with Copy Propagation. Thus, apart from its theoretical importance, this result is relevant for practical compiler design as it allows to exploit the optimization potential of the graph–based algorithm for non–linear code as well.

Overview

Literature on optimizing compilers describes a wide variety of code transformations which aim at improving the efficiency of the generated code with respect to different parameters. Most of them concentrate on specific aspects such as the elimination of redundant computations or the minimization of register usage. There are, however, also combined methods which integrate several optimization steps in one procedure.

In this talk we compare certain classical optimizing transformations for straight–line code with a combined procedure which first constructs a directed acyclic graph (DAG) representing the given program and then generates optimized code from it. The basic version of the latter has been introduced by Aho, Sethi and Ullman (1970) [3], and the authors claim that it produces optimal results regarding the length of the generated code. However the DAG procedure cannot directly be applied to iterative code containing loops, which on the other hand is possible for most of the classical transformations.

Optimality here is meant w.r.t. strong equivalence of programs, hus transformations are not allowed to modify the syntactic representation of computation results. Optimality in general is only decidable for a small subclass of operations, such as the arithmetics on the integers without division [4].

Analyzed Algorithms

We first present a slightly modified version of the DAG algorithm, denoted by T_{DAG}, which in addition supports constant folding. We then show that it integrates the following three classical transformations:

Constant Folding (T_{CF}), which corresponds to a partial evaluation of the program with respect to a given interpretation of its constant and operation symbols;

Common Subexpression Elimination (T_{CS}), which aims to decrease the execution time of the program by avoiding multiple evaluations of the same expression; and

Dead Code Elimination (T_{DC}), which removes computations that do not contribute to the actual result of the program.

It will turn out that these transformations are not sufficient to completely encompass the optimizing effect of the DAG algorithm. Rather a fourth transformation, *Copy Propagation* (T_{CP}), has to be added, which propagates variables from variable–copy assignments. This does not have an optimizing effect on its own but generally enables other transformations such as Common Subexpression Elimination and Dead Code Elimination. In fact we will show that the DAG procedure can essentially[1] be characterized as a combination of Copy Propagation and the first three transformations.

Determining the Application Order

To find the right combination of the transformations we have to analyze the dependences of the algorithm. Some of them *enable* other optimizations, i.e. a transformation T is called T'-enabling if a T'-optimal[2] program exists such that the application of T on it produces additional optimization potential w.r.t. T'.

For the classical tranformations and Copy Propagation the relations shown in the following table hold:

[1] "Essentially" here means that some additional minor modifications are required to make the two resulting programs syntactically equal.

[2] Optimality here means that an application of T' has no effect.

	T_{DC}	T_{CS}	T_{CF}	T_{CP}
T_{DC}		-	-	-
T_{CS}	-		-	$\rightarrow$
T_{CF}	$\rightarrow$	$\rightarrow$		-
T_{CP}	$\rightarrow$	$\rightarrow$	-	

Here an arrow means that the transformation labeling the row enables the transformation indexing the column, whereas a dash indicates the absence of an enabling effect.

We will show that, under the above restriction, the DAG algorithm corresponds to a repeated application of Common Subexpression Elimination and Copy Propagation in alternation which arises due to the mutual dependence of the two algorithms, preceded by Constant Folding and followed by Dead Code Elimination:

$$T_{DAG} \approx T_{DC} \circ (T_{CP} \circ T_{CS})^* \circ T_{CF}.$$

The number of applications of $T_{CP} \circ T_{CS}$ is bounded by the number of instructions in the program.

Apart from its theoretical importance, this result is also relevant for practical compiler design as it allows to exploit the optimization potential of the DAG–based algorithm for non–linear code as well.

Here we could only give a short summary of our work. Further details, including a full description of the algorithms and the proofs of our results can be found in our technical report [1]. For experimenting with concrete examples a web-interface has been made available at [2].

References

[1] Thomas Noll and Stefan Rieger: *Optimization of Straight–Line Code Revisited*, Aachener Informatik Bericht 2005–21, RWTH Aachen University, 2005, `aib.informatik.rwth-aachen.de/2005/2005-21.pdf`.

[2] Stefan Rieger: *SLC-Optimizer - Web Interface*, `slc.srieger.com`.

[3] Alfred V. Aho and Ravi Sethi and J. D. Ullman: *A formal approach to code optimization*, Proceedings of a symposium on Compiler optimization, pp. 86–100, Urbana-Champaign, Illinois, 1970.

[4] Oscar H. Ibarra and Brian S. Leininger: *The complexity of the equivalence problem for straight-line programs*, STOC '80: Proceedings of the twelfth annual ACM symposium on theory of computing, pp. 273–280, Los Angeles, California, United States, 1980.