A Retargetable Instruction Reorganizer

Jack W. Davidson

Computer Science Technical Report 85-17
September 1, 1985

A Retargetable Instruction Reorganizer

Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

ABSTRACT

Extant peephole optimizers can perform many optimizations that are handled by higher-level optimizers. This paper describes a retargetable instruction reorganizer that performs targeting and evaluation order determination by applying a well known algorithm for optimal code generation for expressions to object code. The reorganizer fits well into the framework of an existing peephole optimizer. By rearranging computations and instructions, it minimizes the number of registers and temporaries required to compile expressions. For some machines, this can reduce both the size and execution speed of programs. By generalizing its operation, the reorganizer can also be used to reorder instructions to avoid delays in pipelined machines. For one pipelined machine, it has provided a 5 to 10 percent improvement in the execution speed of benchmark programs.

September 1, 1985

Department of Computer Science

The University of Virginia

Charlottesville, VA 22901

A Retargetable Instruction Reorganizer

1. Introduction

Normally execution ordering, the determination of the "optimal" sequence of instructions to evaluate an arithmetic expression, is performed during code generation [Aho77b, Seth70, Stoc73]. In most cases, "optimal" means the code sequence with the fewest instructions and/or the minimum number of storage references. In an ideal situation, it has been proved that these algorithms generate optimal code in time linear to the size of the expression. Unfortunately most present-day machines are far removed from the ideal assumed by these algorithms. Nevertheless, these algorithms can be applied in practice with good results. Johnson provides an excellent discussion of the application of this theory in a real compiler [John78]. Here the compiler attempts to produce a worst case estimate of the resources (e.g. registers or temporaries) required to emit code for each subtree of the expression. These estimates are often referred to as Sethi-Ullman numbers. Based on these estimates, an evaluation order is produced. As Johnson notes, the notion of determining the resources required to generate code for a subtree breaks down for almost all practical machines. Consequently, it is often necessary to use heuristics to provide estimates about the resources required to evaluate an expression. Thus, while the algorithm itself is machine-independent, its efficacy depends on the machine-dependent portion used to produce the estimates.

This paper describes a retargetable instruction reorganizer, called Order, that implements two optimizations, targeting [Leve79, Wulf75] and evaluation order determination. Targeting makes use of the commutativity of many arithmetic and logical operators to avoid unnecessary loads and stores†. Order has been integrated into PO, an existing peephole optimizer [Davi80, Davi84a]. In a manner similar to this optimizer, Order operates on object code instead of the conventional expression tree. Through this and by applying the straightforward evaluation order algorithm of Sethi and Ullman [Seth70], it is able to handle machine features that complicate conventional implementations of evaluation order algorithms. Consequently, for a given expression tree Order generates the optimal evaluation order.

By rearranging computations and instructions, Order can reduce the number of instructions and registers required to compile expressions. For some machines, this can reduce both the size and execution speed of programs. By generalizing its operation, it can also be used to reorder instructions to avoid delays in pipelined machines. For one such machine, the execution speed of a set of benchmark programs was reduced substantially.

†Many evaluation order algorithms handle targeting, but it is actually a separate optimization. The technique presented here treats it separately from evaluation order optimization.

2. Motivation

Emerging peephole optimizers have subsumed many of the code generation tasks conventionally performed at a higher level [Davi84a, Davi84b, Fras82, Gieg83]. A major motivation for performing as many optimizations as possible on object code is that this often provides some relief from phase ordering problems caused by interdependence of the various phases of a compiler. For example, in most compilers common subexpressions are eliminated in early phases that operate on a machine-independent representation of the program. They do so to simplify their implementation and to remain machine-independent. During the code generation phase when the intermediate code is translated to machine code, it is possible that new common subexpressions will be introduced. By performing common subexpression elimination on object code, all common subexpressions are detected and the interdependence between an early and later phase of compilation is removed. A similar situation exists when register assignment is performed early in the compilation process as opposed to later. Removing such interdependencies usually results in the generation of better code.

A second advantage to operating on object code is that many optimizations require some knowledge of the target machine even when they are applied to a machine-independent representation. Targeting and evaluation order determination are two optimizations in this class. In the case of evaluation order determination, the algorithm is applied to expression trees, yet knowledge of the target machine is necessary to estimate the resources required to generate code for the expression trees. Typically such optimizations are performed at a higher level by isolating the machine-dependent information in separate modules. The amount of target machine information that these modules can contain is necessarily limited. This lack of information can limit the power of the optimization. When this type of optimization is performed on object code, however, the complete knowledge of the target machine allows the full power of the optimization to be realized.

Accordingly, Order was designed with the following goals in mind.

1. It should fit nicely into the PO model of compilation. In particular, it should operate on register transfers in a manner similar to the other phases of PO.
2. It should use the theory provided by the Sethi-Ullman algorithm [Seth70].
3. It should be easily retargeted for a new machine. Retargeting should require no more than a days effort.
4. It should provide good improvements in performance at little or no cost.

3. PO

The PO optimizer is made up of three distinct phases called Cacher, Combiner, and Assigner. Each phase operates on register transfer lists ('RTLs') which describe an instruction's effect. Any RTL is machine specific, but the form of the RTL is machine independent. Briefly, Cacher performs common subexpression elimination, allocates registers,

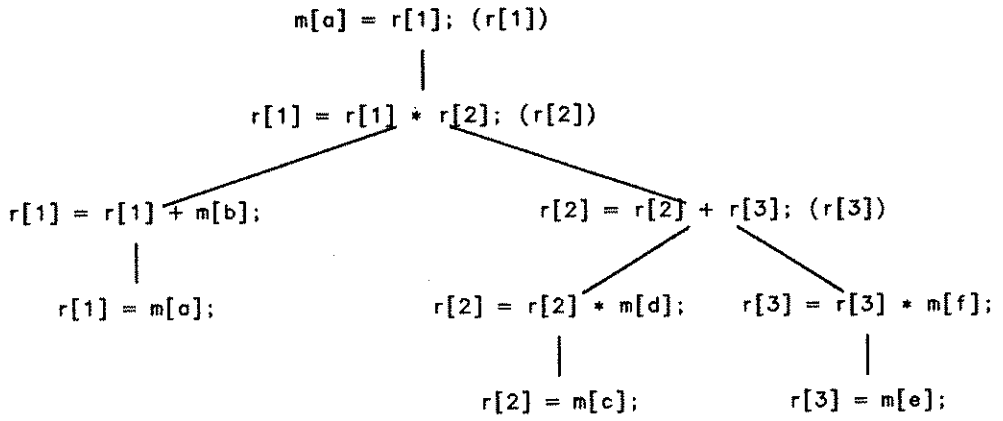
does a limited type of flow analysis, and identifies dead variables. Combiner advances over Cacher's output seeking adjacent instructions that can be replaced with singletons. Once optimization is complete, Assigner does register assignment and translates the RTLs to assembly language. Other documents offer a more complete treatment of Cacher and Assigner [Davi84b], and PO [Davi84a].

4. Order

Order is integrated into the Combiner phase of PO. It performs targeting while Combiner operates, and evaluation order determination after Combiner has completed its optimization.

While Combiner is usually described as operating on a linear ordering of RTLs, it actually processes a tree that was created by the flow analysis of Cacher. These trees are referred to as R-trees because the nodes are the records that contain the RTLs and the dead-variable list. Below is an example of a R-tree for a simple register machine for the expression:

```
a = (a + b) * (c * d + e * f)
```



The registers (e.g. r[2]) in parentheses denote the dead variables.

The R-tree for an expression is very similar to the expression trees used to perform conventional evaluation order determination. Indeed, the simple algorithm involving a bottom-to-top tree walk for finding the optimal evaluation order can be easily applied to R-trees. There is one important difference. This algorithm requires that each node be labeled with the cost to compute it. Usually the cost associated with a subtree is either the number of registers required to evaluate it, or the number of instructions it contains. By processing R-trees it is now possible to determine the exact cost of evaluating a subtree. The labeling algorithm can handle machine idiosyncrasies such as requiring multiplication and division to be performed in even/odd register pairs, exotic addressing modes, and different register classes. There is no need for heuristics or estimation functions.

Order implements both targeting and evaluation order determination by computing the cost of evaluating an expression's R-tree. For targeting the cost is computed by performing a trial optimization to determine whether a child instruction can be combined with its parent. The cost of a node is the number of children that can combine with it. Determining whether one instruction can combine with another is a relatively expensive operation. Fortunately to achieve targeting this cost need only be computed for subtrees where the root contains a commutative operation. For evaluation order determination, the cost is the number of registers required to evaluate the subtrees. In a later section, it is shown that by changing the criterion for optimality that it is possible to have Order find code sequences that are good for pipelined machines.

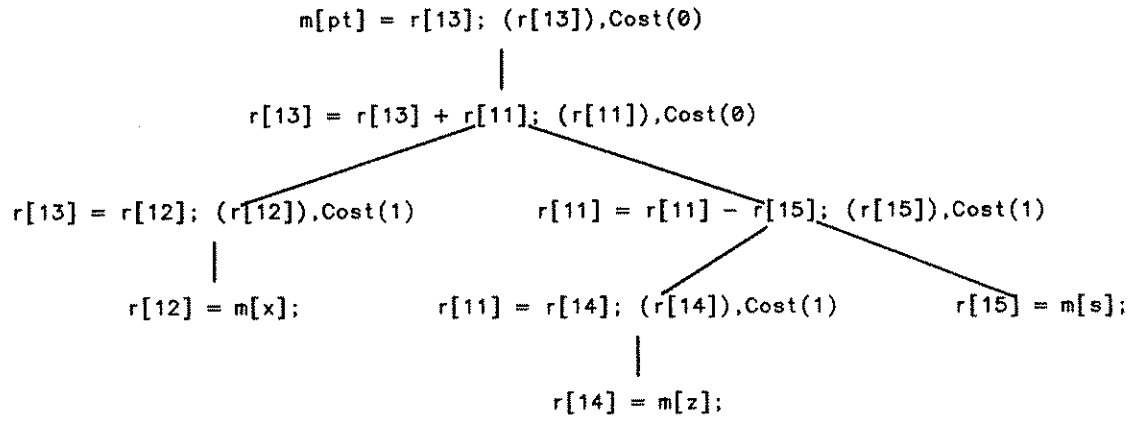
4.1 Targeting

Targeting or renaming [Scha76] is an optimization that is useful on one-address and two-address general register architectures. It makes use of the commutativity of arithmetic and logical operators to avoid unnecessary loads and stores. It is also useful on three-address machines, like the VAX-11/780, where cheaper two-address instructions variants are also available. For a true three-address machine, like many of the CDC Cyber architectures, targeting is less useful.

Consider the following simple expression:

```
pt = x + (z - s)
```

The R-tree for this expression on the PDP-11 with the cost of each node computed is:



The code generator assumes an infinite supply of registers. Assigner maps references to nonexistent pseudo-registers (like r[13]) onto the machine's real registers. This is a common technique for simplifying code generation [Chai81]. In addition, all values were formed in new registers so that each value was available during common subexpression elimination [Davi84b]. Any unnecessary register moves will be removed by Combiner/Assigner.

For the above example, when no targeting is performed, Combiner/Assigner produces the following code:

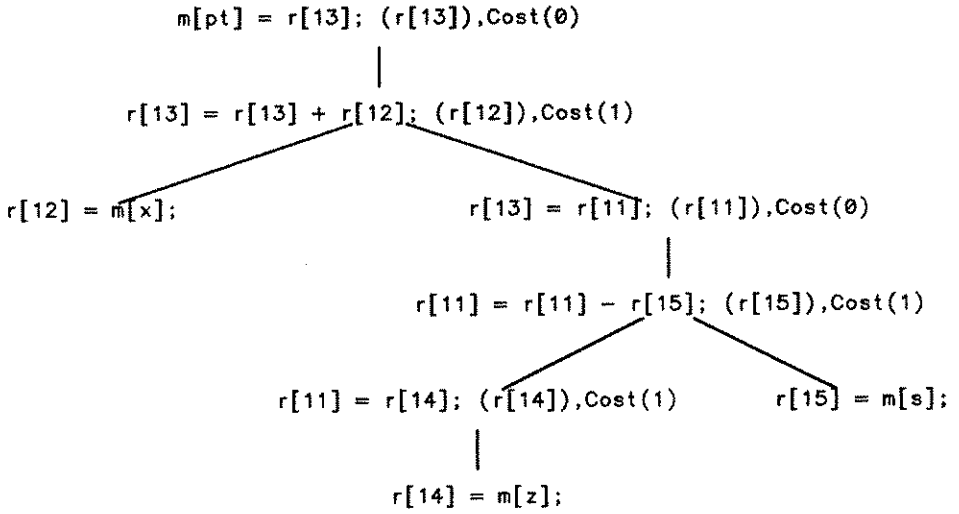
```
mov     x,r2
mov     z,r3
sub     s,r3
add     r3,r2
mov     r2,pt
```

This sequence has five instructions and uses two registers. The optimal sequence has four instructions and uses one register.

Order achieves targeting in the following way. It locates instructions that perform commutative operations that have two descendants. This identifies subtrees that are eligible for targeting. It then produces a revised subtree where the instruction copying the value from the left subtree is inserted in the right subtree rewriting it so that it copies the value produced by the original right subtree. It then rewrites the commutative instruction so that the source operand is taken from the left subtree. The costs of the original and revised subtrees are computed and compared. If the original subtree has a higher cost it is retained, otherwise it is replaced by the revised subtree.†

This operation is similar to the "flipping" operation performed by the Sethi-Ullman algorithm. Order must perform targeting early so that Combiner's ability to choose the best addressing modes and instructions can be applied to the transformed code.

In the above example, the instruction that performs the addition is a candidate for targeting. The revised subtree has a cost of one so Order transforms the R-tree to:



Combiner/Assigner reduces this to:

```
mov     z,r1
sub     s,r1
add     x,r1
mov     r1,pt
```

which is the optimal four instruction sequence that uses one register.

†A higher cost implies fewer instructions in the resulting code.

For a three-address machine like the VAX-11, the code generator always emits three-address code to avoid destroying any values. For these machines, Order rewrites the commutative instruction so that the operand with the higher cost is first. Later phases of optimization will transform the instruction to the cheaper two-address variant if possible.

4.2 Evaluation Order Determination

After Combiner has processed the R-trees, Order computes the number of registers needed to evaluate each node. By computing the number of registers at this point in the compilation process, Order is able to determine the exact number of registers required for each subtree.

The algorithm for computing the number of registers required to evaluate a node of a R-tree is:

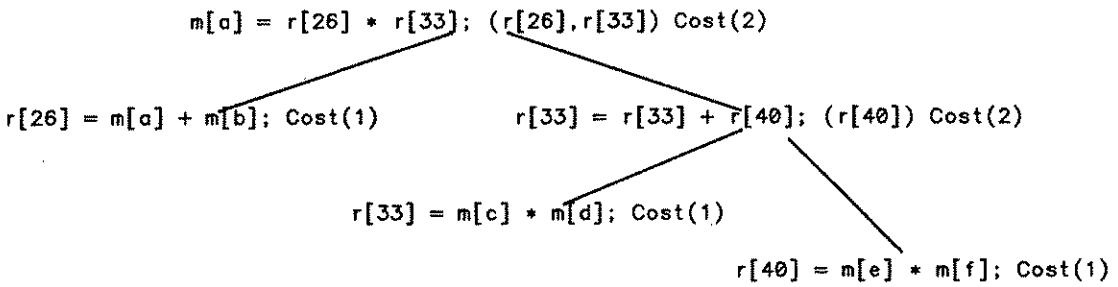
```
struct node {
   char *rtl;                                   /* register transfer list */
   char *deads;                                 /* dead variable list */
   struct node *descendents[MAXDESCEND];        /* pointers to descendents */
   int cost;                                    /* cost to compute node */
};
int cost(n)
struct node *n;
{
int rusage(), i, max();

   n->cost = rusage(n);
   for (i = 0; n->descendents[i] != NULL; i++)     /* evaluate subtrees if any */
      n->cost = max(n->cost, cost(n->descendents[i]));
   return(n->cost);
}
```

`rusage` computes, for a given RTL and dead-variable list, the exact number of registers used by the node. It is necessary to process the dead-variable list as well as the RTL because some instructions change registers not explicitly appearing in the RTL. The code generator places such registers on the dead-variable list.

To minimize register usage, Order outputs the subtrees whose register usage is the highest first. Consider the expression

```
a = (a + b) * (c * d + e * f)
```

The R-tree produced by Combiner for the VAX-11/780 is:



The straightforward postorder walk of the R-tree yields the following code:

```
addl3   b,a,r2
mull3   d,c,r3
mull3   f,e,r4
addl2   r4,r3
mull3   r3,r2,a
```

which uses three registers. By processing the subtrees with the highest register usage first, the following code is obtained:

```
mull3   d,c,r2
mull3   f,e,r3
addl2   r3,r2
addl3   b,a,r3
mull3   r2,r3,a
```

which is the optimal code that uses two registers.

Conventional compilers perform evaluation order early because they must do so to generate intermediate code. Without knowledge of the machine, evaluation order determination must rely on heuristics to determine the cost of evaluating the subtrees. The problem is further compounded as code generation and optimization may make substantial transformations to the code. By performing evaluation order determination on object code after code generation and optimization these problems are avoided.

4.3 Common Subexpressions

Cacher, the first phase of PO, can be directed to eliminate common subexpressions from the object code. It is well known that optimal code generation in the presence of common subexpressions is a difficult problem [Aho77a,Brun76]. Aho, Johnson, and Ullman [Aho77a] present several approaches to this problem and analyze their worst-case behavior. The approach used by Order was chosen not so much for the quality of the code generated (although it seems to do reasonably well), but because it was easy to integrate into Order.

The approach, discussed by Waite [Wait76] and mentioned by Aho, Johnson and Ullman [Aho77a], is to convert the DAG representing the computation to a forest of R-trees that can then be processed in the proper order by the previous algorithm. Because information about sharing has been lost, it is not possible to guarantee optimal code.

5. Instruction Scheduling

By generalizing its operation, Order can also be used to reorganize code to avoid delays in pipelined machines, a technique sometimes called instruction scheduling. Rymarczyk [Ryma82] describes and classifies possible sources of pipeline delays. For a pipelined machine, Order evaluates the subtrees of a R-tree node to determine which would delay the evaluation of the node the longest. It outputs this subtree first, which has the effect of moving the code for that subtree away from the instruction it delays.

By only rearranging the order of evaluation of the descendents of a node, Order remains simple, yet it may miss opportunities for avoiding delays. A more thorough technique for

optimizing pipelined code is described by Hennessy and Gross [Henn83]. We are investigating incorporating a modified version of their algorithm in Order. Nonetheless, Order's simple algorithm is quite effective. For the Prime 9950, a general-register machine with a five-stage pipeline, Order reduced the execution speed of a C [Kern78] benchmark program that calls a character string to integer conversion routine 30000 times from 7.9 seconds to 7.4 seconds — a 6.7 percent improvement in execution speed. For a matrix multiplication program, the improvement was 10 percent.

6. Implementation

Order is written in C and is integrated into the Combiner phase of PO. PO runs under most implementations of UNIX. The machine-independent portion of Order is approximately 300 lines of code. It uses a negligible amount of processing time. Order is retargeted by supplying a new cost function. For conventional machines, this requires specifying the number of registers available and writing patterns that identify the registers. For pipelined machines, the cost function is more difficult to write. It must identify and classify potential sources of conflicts between instructions. The pipeline delay cost function for the Prime 9950 is 35 lines of code.

7. Performance

Order can significantly reduce the size of the object code. For a machine like the PDP-11, with only 5 allocable registers, reductions come from two sources: fewer register spills and fewer register-to-register moves. Order evaluation determination reduces the number of spills, while targeting eliminates unnecessary register-to-register moves. For one program on the PDP-11, Order reduced the number of register spills from 5 to 1. When used to compile a large body of C and Y [Hans81], test programs it reduced the number of lines of code produced by two to six percent.

Even for machines with a large register set, where register spills are rare, Order can still provide savings. Consider the standard calling convention on the VAX-11 [Digi81]. Registers R0 and R1 are considered scratch registers. A called procedure may use R2 through R11 provided it saves and restores them. These stores and loads can be a significant portion of the call/return overhead [Lund75]. By producing the optimal evaluation order, most expressions can be handled with just two registers — R0 and R1. This means that no registers must be saved. For one highly recursive program run on the VAX, this allowed Order to reduced the execution speed by 5.3 percent. When run on a large set of C and Y test programs, Order reduced the code size by two to three percent.

The above statistics on improvements underestimate Order's effectiveness. A version of Order was configured to reorder the code to mistarget operands and to maximize the register usage. When the same set of test programs were compiled, the code produced was 15 to 20 percent larger than the optimized code for both the VAX and the PDP-11. This shows that there are many opportunities for targeting and evaluation order determination, but the

naive code generators used by the C and Y compilers do a reasonably good job of selecting the correct code to emit.

8. Discussion

Conventional code generators perform evaluation order determination and targeting early, before code generation. The conventional wisdom is that this simplifies implementation and avoids machine dependence. Unfortunately, these optimizations require knowledge of the target machine to operate. The limited knowledge of the target machine during the early phases of compilation can limit the effectiveness of the optimization. Order shows that by applying these optimizations to object code, where knowledge of the machine is complete, that more thorough optimization is possible at a cost equivalent to operating on higher-level representations. By performing these optimizations later, phase ordering problems encountered by many "early" optimizers may be avoided. By generalizing its operation, Order can perform a limited form of instruction scheduling for pipelined machines. Work in progress is adapting a more thorough algorithm for these machines. This is an important area of research as many recent architectures are employing pipelining to improve performance.

9. References

[Aho77a] A. V. Aho, S. C. Johnson and J. D. Ullman, Code Generation for Expressions with Common Subexpressions, *Journal of the ACM 24*, 1 (January 1977), 146-160.

[Aho77b] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.

[Brun76] J. Bruno and R. Sethi, Code Generation for a One-Register Machine, *Journal of the ACM 23*, 3 (July 1976), 502-510.

[Chai81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, Register Allocation via Coloring, *Computer Languages 6*, 1 (1981), 47-57.

[Davi80] J. W. Davidson and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), 191-202.

[Davi84a] J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems 6*, 4 (October 1984), 7-32.

[Davi84b] J. W. Davidson and C. W. Fraser, Register Allocation and Exhaustive Peephole Optimization, *Software - Practice and Experience 14*, 9 (September 1984), 857-866.

[Digi81] *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, MA, 1981.

[Fras82] C. W. Fraser, A Generalization of Two Code Ordering Optimizations, Tech. Rep. 82-11, University of Arizona, Tucson, AZ, October 1982.

[Gieg83] R. Giegerich, A Formal Framework for the Derivation of Machine-Specific Optimizers, *Transactions on Programming Languages and Systems 5*, 3 (July 1983), 478-498.

[Hans81] D. R. Hanson, The Y Programming Language, *SIGPLAN Notices 16*, 2 (February 1981), 59-68.

[Henn83] J. Hennessy and T. Gross, Postpass Code Optimization of Pipeline Constraints, *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 422-448.

[John78] S. C. Johnson, A Portable Compiler: Theory and Practice, *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson, AZ, January 1978, 97-104.

[Kern78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Leve79] B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz and W. A. Wulf, An Overview of the Production Quality Compiler-Compiler Project, CMU-CS-79-105, Carnegie-Mellon University, Pittsburg, PA, February 1979.

[Lund75] A. Lunde, More Data of the O/W Ratios: A Note on a Paper by Flynn, *Computer Architecture News 4*, 1 (March 1975), 9-13.

[Ryma82] J. W. Rymarczyk, Coding Guidelines for Pipelined Processors, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 12-19.

[Scha76] B. R. Schatz, Algorithms for Optimizing Transformations in a General Purpose Compiler: Propagation and Renaming, RC 6232, IBM Thomas J. Watson Research Center , Yorktown Heights, NY, October 1976.

[Seth70] R. Sethi and J. D. Ullman, The Generation of Optimal Code for Arithmetic Expressions, *Journal of the ACM 17*, 6 (October 1970), 715-728.

[Stoc73] P. F. Stockhausen, Adapting Optimal Code Generation for Arithmetic Expressions to the Instruction Sets Available on Present-Day Computers, *Communications of the ACM 16*, 6 (June 1973), 353-354..

[Wait76] W. M. Waite, Optimization, in *Compiler Construction: An Advanced Course*, F. L. Bauer and J. Eickel (ed.), Springer-Verlag, New York, NY, 1976, 549-602.

[Wulf75] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, NY, 1975.