Redundancy Elimination Revisited

Keith Cooper
Dept of Computer Science
Rice University
Houston, TX
keith@rice.edu

Jason Eckhardt
Dept of Computer Science
Rice University
Houston, TX
jle@rice.edu

Ken Kennedy*
Dept of Computer Science
Rice University
Houston, TX

ABSTRACT

This work proposes and evaluates improvements to previously known algorithms for redundancy elimination.

Enhanced Scalar Replacement combines two classic techniques, scalar replacement and hash-based value numbering. The former detects redundant array references within and across loop iterations, while the latter detects a large class of redundancies, but only within a single loop iteration. By integrating the two techniques, ESR detects and eliminates a wider range of expression redundancies across loop iterations.

We also extend hash-based value numbering to perform *reassociation*. Classic redundancy elimination techniques operate on an intermediate representation of the program in which operand association and order is of fixed shape. This rigidity in code shape may sometimes obscure redundancies. Our optimizer attempts to shape the code by changing associativity, exposing more redundancies. Opportunities for ESR, in particular, are increased with reassociation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—optimization, compilers

General Terms

Algorithms, Experimentation, Performance

Keywords

expression optimization, loop optimization, redundancy elimination, scalar replacement, reassociation

1. INTRODUCTION

A variety of successful techniques for redundancy elimination have been developed since the 1950's. Hash-based *value numbering* is one such family of methods, which was originated by Ershov in the mid- to late 1950's [12] and more fully described (and extended) in the early 1970's by Cocke and Schwartz [10]. It is still a widely implemented and useful technique.

The key idea behind value numbering is to assign to each expression a *value number*, such that two expressions are assigned the same number if they compute the same value. Value numbering detects opportunities in any region of code, but has limitations with respect to loops.

Another technique is *Scalar replacement* [9], which detects re-use of subscripted variables within loops and replaces such redundant accesses with references to scalar variables. A high-quality register allocator will then allocate the new scalar variables to registers. Scalar replacement utilizes data dependence analysis to discover re-use not only within a loop iteration, but also across loop iterations. From this point on, this algorithm will be referred to as "classic scalar replacement" or CSR.

1.1 Improving Redundancy Elimination in Loops

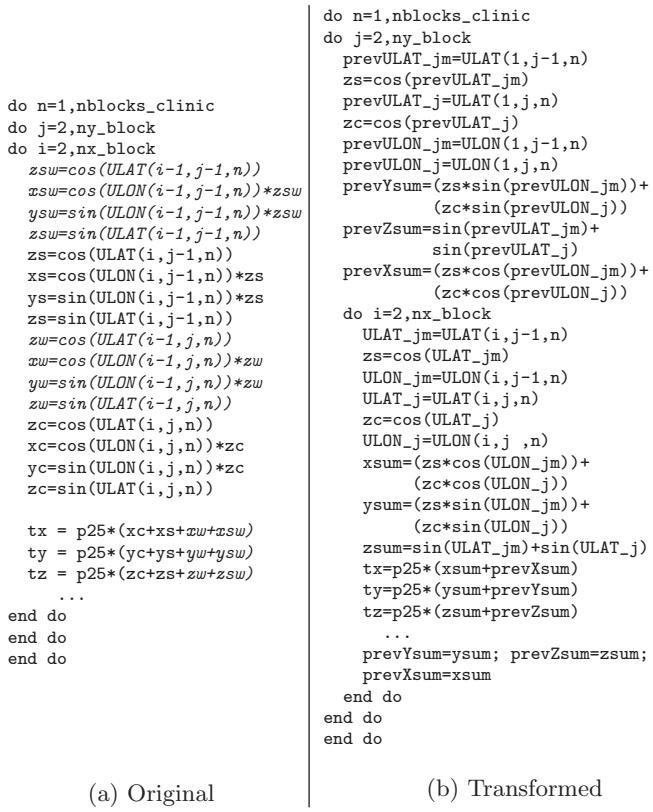
Value numbering is an effective and powerful method for eliminating redundant expressions. However, it is able to do so only within a single iteration of a loop. There are situations where it is profitable to detect redundant expressions from *different* iterations. Classic scalar replacement, on the other hand, detects redundancies across loop iterations, but only when the expression takes the form of an array reference. What is desired is to combine the two ideas into a more powerful method.

As a first example illustrating the idea of inter-iteration redundancy, consider the loop in Figure 1a from Los Alamos National Lab's Parallel Ocean Program (POP). Upon close inspection, it can be seen that the italicized expressions are redundant, having been computed one iteration earlier by the corresponding unitalicized expressions. With care, one can eliminate the redundancies by rewriting the code as shown in Figure 1b. In this case, there is a substantial reduction in memory references, floating-point operations, and even expensive intrinsic calls. The original inner loop contains 16 loads, 11 floating-point multiplies, 11 floating-point additions, 8 `cos` intrinsic calls, and 8 `sin` intrinsic calls. The transformed inner loop contains only 4 loads, 7 floating-point multiplies, 8 floating-point additions, 4 `cos` intrinsic calls, and 4 `sin` intrinsic calls.

Note that standard redundancy elimination techniques will not detect many of the redundancies in the example.

*Ken co-directed this work before his death.

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.



```
do n=1,nblocks_clinic
do j=2,ny_block
do i=2,nx_block
  zsw=cos(ULAT(i-1,j-1,n))
  xsw=cos(ULON(i-1,j-1,n))*zsw
  ysw=sin(ULON(i-1,j-1,n))*zsw
  zsw=sin(ULAT(i-1,j-1,n))
  zs=cos(ULAT(i,j-1,n))
  xs=cos(ULON(i,j-1,n))*zs
  ys=sin(ULON(i,j-1,n))*zs
  zs=sin(ULAT(i,j-1,n))
  zw=cos(ULAT(i-1,j,n))
  xw=cos(ULON(i-1,j,n))*zw
  yw=sin(ULON(i-1,j,n))*zw
  zw=sin(ULAT(i-1,j,n))
  zc=cos(ULAT(i,j,n))
  xc=cos(ULON(i,j,n))*zc
  yc=sin(ULON(i,j,n))*zc
  zc=sin(ULAT(i,j,n))

  tx = p25*(xc+xs+xw+xsw)
  ty = p25*(yc+ys+yw+ysw)
  tz = p25*(zc+zs+zw+zsw)
    ...
end do
end do
end do
```

(a) Original

```
do n=1,nblocks_clinic
do j=2,ny_block
  prevULAT_jm=ULAT(1,j-1,n)
  zs=cos(prevULAT_jm)
  prevULAT_j=ULAT(1,j,n)
  zc=cos(prevULAT_j)
  prevULON_jm=ULON(1,j-1,n)
  prevULON_j=ULON(1,j,n)
  prevYsum=(zs*sin(prevULON_jm))+
           (zc*sin(prevULON_j))
  prevZsum=sin(prevULAT_jm)+
           sin(prevULAT_j)
  prevXsum=(zs*cos(prevULON_jm))+
           (zc*cos(prevULON_j))
  do i=2,nx_block
    ULAT_jm=ULAT(i,j-1,n)
    zs=cos(ULAT_jm)
    ULON_jm=ULON(i,j-1,n)
    ULAT_j=ULAT(i,j,n)
    zc=cos(ULAT_j)
    ULON_j=ULON(i,j ,n)
    xsum=(zs*cos(ULON_jm))+
         (zc*cos(ULON_j))
    ysum=(zs*sin(ULON_jm))+
         (zc*sin(ULON_j))
    zsum=sin(ULAT_jm)+sin(ULAT_j)
    tx=p25*(xsum+prevXsum)
    ty=p25*(ysum+prevYsum)
    tz=p25*(zsum+prevZsum)
      ...
    prevYsum=ysum; prevZsum=zsum;
    prevXsum=xsum
  end do
end do
end do
```

(b) Transformed

Figure 1: An example from the Parallel Ocean Program code

Methods based on lexical identity will not detect commonality because the operand names are not lexically identical (e.g., `ULAT(i-1,j-1,n)` and `ULAT(i,j-1,n)` are not the same) and because they cannot track writes to individual array elements. Standard methods based on value identity will do no better, since those same two references will be assumed to have different values (their leftmost index expressions will have different value numbers).

Applying classic scalar replacement would eliminate the same number of loads, but nothing else. Compared to CSR, eliminating whole expressions will often require fewer temporaries because entire expressions are stored rather than each individual array reference that might make up the expression. This implies that more redundancies can be eliminated before exhausting the supply of physical registers.

Examining the POP example, one might argue that the programmer was naive for writing the loop in the original form, considering that it does so much redundant computation. But there are good reasons to write such codes in that "naive" form. The transformed version, while it may run faster, is much less clear than the original. Initial values are fed in from outside the loop, multiple values from different iterations are live simultaneously and shifted through scalar copies, etc. The transformed version is also likely to be more difficult and error-prone to write manually. The programmer should be able to express the computation in a natural, readable, and maintainable form, and leave it to an optimizing compiler to produce a potentially less natural, but higher-performing version. A goal of this work is to be able to automatically detect and transform codes like the POP example. These kinds of loops are important and occur in a number of other scientific codes as well as signal and image processing.

1.2 Improving Redundancy Elimination via Reassociation

It has been known since the early days of compiling that opportunities for detecting redundancies or compile-time constants may be obscured when the optimizer examines only the default representation of the program provided by the parser or other pass (e.g., [13, 6]). *Reassociation* is a method whereby expressions are rearranged to expose more opportunities for redundancy elimination. Through examination of many application codes, we found that a number of opportunities for inter-iteration redundancies are *only* detectable if expressions are reassociated. This work extends value numbering with reassociation. While it is presented in the context of ESR's inter-iteration value numberer, it can also be incorporated into the classic value numbering algorithms.

1.3 Overview

Section 2 reviews the basic techniques and analyses on which this work is built. The technical contributions are presented in sections 3 and 4. Section 5 presents an empirical evaluation of the techniques. Section 6 surveys the literature closely related to this work. Finally, section 8 wraps up and gives some final perspectives on the work.

2. BACKGROUND

We assume the reader is familiar with standard data dependence analysis (see chapter 2 of [2]).

2.1 Classic Scalar Replacement

Scalar replacement utilizes data dependence information to discover re-use opportunities, and to determine how to rewrite the code. It operates by using the input- and true-dependences to pinpoint re-use of values in array elements. The primary data structure created is a list of *name partitions*. Each partition groups array references together that refer to the same value. The *generator* of a group is the reference which produces the value that is reused by all other members of the group. Intuitively, the generator is the "oldest" load, or the "most recent" store of a group of references to the same location. By assigning the generator's result to a scalar temporary, all other references in the same group can be replaced by a reference to the scalar (multiple scalar temporaries are needed to carry values across multiple iterations). A brief sketch of the classic scalar replacement algorithm is given below (following chapter 8 of [2]):

1. *Partition*: Prune dependence graph, so that only edges representing value re-use remain. Group together references that are related by true or input dependences.
2. *Select*: Select a set of name partitions using register pressure moderation.
3. *Replace*: For each selected partition, replace all references (other than the generator) with scalar temporaries.

Often scalar replacement is able to find more than enough opportunities so that rewriting them all would quickly exhaust the supply of physical registers of the machine. The

Select phase performs a register pressure moderation algorithm which picks only a subset of the groups, subject to available registers. It associates benefits and costs with each group to determine the most replacement "bang for the buck."

2.2 Value Numbering

A typical value numberer operates on a linear intermediate form. Each operation is traversed in program order, assigning value numbers to each operation so that redundant expressions receive the same value number. Numbering in program order ensures that all operands of an operation have been assigned value numbers before the operation itself is considered. Below is a brief sketch of a baseline hash-based value numberer, as described in [8].

```
For each statement "x <-- y op z" in block B
  expr <-- <NameToVN[y] op NameToVN[z]>
  if expr is found in hash table with VN v
    NameToVN[x] <-- v
    if NameToVN[VNToName[v]] = v
      Replace rhs of statement with VNToName[v]
  else
    v <-- next available value number
    NameToVN[x] <-- v
    Add expr to the hash table with VN v
    VNToName[v] <-- x
```

Value numbering is quite flexible and can easily take into account algebraic identities, exploit commutativity of operations, and perform constant propagation and folding. Furthermore, the rewrite/replacement policies are flexible and can be performed *online* (incrementally, as redundancies are discovered) or *offline* (as a postpass, after all redundancies are known).

3. ENHANCED SCALAR REPLACEMENT

This section describes an algorithm that combines the benefits of both CSR and value numbering, with the goal of eliminating the kinds of inter-iteration redundancies examined in section 1.1. Our general approach builds directly on the well-developed and understood classic scalar replacement algorithm, and directly incorporates value numbering into the process. The idea is to leverage the data structures and infrastructure (including register pressure moderation) of CSR and to tightly integrate value numbering so that phase ordering problems are not introduced and all redundancy opportunities are considered together. Due to space limitations, we explain the technique in prose as opposed to low-level pseudocode.

3.1 Preliminaries

Assumptions.

It is assumed that the optimizer's intermediate representation is in the form of a medium- to high-level abstract syntax tree[1]: 1) a function is represented as a hierarchical list of statements, where side-effects are only allowed at statement boundaries (simplifies value numbering); 2) array references correspond closely to the source-level, so that index expressions have not been expanded to low-level address expressions; 3) any array/vector syntax (e.g., Fortran 90 triplet notation) has been scalarized into explicit looping constructs.

It is also assumed that the optimizer has a high-quality data dependence analyzer, which is critical to the success of CSR, ESR, or any other dependence-based transformation. This implies that no assumptions are made regarding the form or complexity of array subscripts. Any subscripted variable is an ESR candidate as long as the dependence analyzer can disambiguate them. No special assumptions are made regarding the naming scheme of variables, such as that required by static single assignment form.

Definitions.

In order to detect inter-iteration redundancies, we need to incorporate into value numbering the notion that an expression may have been computed on some previous iteration. The term *expression threshold* (denoted $\Delta(e_1, e_2)$) is coined to represent a quantity analogous to dependence distance or threshold, but applied to whole expressions. Intuitively, if two expressions have the same value d iterations apart (over the whole iteration space), then the distance between them (expression threshold) is d. More formally, for two expression trees e_1 and e_2 with the same operator (or algebraically equivalent), and operands $c_1^1 \ldots c_1^n$ and $c_2^1 \ldots c_2^n$, respectively, $\Delta(e_1, e_2)$ is defined as

$$\begin{cases} thresh(e_1, e_2), & \text{if } n = 0 \wedge (thresh(e_1, e_2) \neq \perp) \\ \forall i, \min \Delta(c_1^i, c_2^i), & \text{if } n > 0 \wedge (\Delta(c_1^1, c_2^1) \equiv \\ & \quad \Delta(c_1^2, c_2^2) \equiv \ldots \equiv \Delta(c_1^n, c_2^n)) \\ \perp, & \text{otherwise.} \end{cases}$$

Function *thresh*(r_1, r_2) is defined to compute the distance between a pair of leaves (references), or to return undefined (denoted by $\perp$) if none exists.

$$thresh(r_1, r_2) = \begin{cases} \tau, & \text{if } (r_1 \ \delta^* \ r_2) \wedge consistent(\tau) \\ -\tau, & \text{if } (r_2 \ \delta^* \ r_1) \wedge consistent(\tau) \\ \infty, & \text{if } invar(r_1) \wedge invar(r_2) \\ \perp, & \text{otherwise} \end{cases}$$

Where *invar(e)* is a predicate that returns *true* if expression e is loop invariant, or *false* otherwise. If both leaves are array references with a consistent dependence between them, then the function will determine the direction of the dependence edge and return the threshold τ (or its negative). If both references are loop invariant, then the "threshold" is defined to be ∞. The equivalence of expression thresholds ($\equiv$) is defined to be *true* if both operands are the same constant, if one operand is constant and the other ∞, or if both are ∞, and $false$ otherwise.

3.2 Algorithm

Our algorithm fits into the overall CSR framework by creating a new primary data structure analogous to name partitions, which we call *expression groups*. These groups will contain entire redundant expressions, not just array references. We use the term "groups" instead of partitions because redundant expressions in one group can overlap with those of other groups (i.e., when they share some operands). We insert a new *Groups* phase between the *Partitions* and *Select* phases which computes the expression groups. It does this by performing inter-iteration value numbering. Once all

[1]A linear IR could be used, but as will be seen in section 4, operating over expression trees exposes more context which can be capitalized upon for manipulating expressions in certain ways.

expression groups are known, we prune away certain uninteresting groups and then calculate the number of registers required to rewrite/replace each one. The rest of scalar replacement operates essentially as before, but using expression groups instead of name partitions.

3.2.1 Determining Expression Groups (inter-iteration VN)

The classic descriptions of value numbering assume a linear intermediate form (see section 2.2) and they perform replacements online (that is, incrementally as redundancies are discovered). Because ESR will be operating on an AST, and the question of high register pressure needs to be addressed, and inter-iteration redundant subexpressions need to be detected, a few adaptations of the standard techniques must be made.

1. *Adapt to AST*: Recall that the basic algorithm traverses each operation (a simple three-address expression) in program order assigning numbers to each. This is done so that the inputs to each operation will already have value numbers assigned to them before the operation is processed. The equivalent idea on an arbitrarily complex expression tree is to traverse the tree in postorder (i.e., bottom up), so that each child (operand) of an operator will have been visited before the operator.

2. *Offline replacement*: A key part of CSR is moderation of register pressure. CSR defers selection of references to rewrite until all opportunities are known. To fit into the CSR framework, the value numberer must operate offline, only collecting and recording redundancies. Once all redundancies have been discovered, the algorithm will determine register pressure and select some subset of the expressions to rewrite.

3. *Inter-iteration redundancies*: Finally, the value numberer in ESR must incorporate data dependence by numbering array references appropriately and utilizing $\Delta(e_1, e_2)$ in determining redundant expressions. This is the key to value numbering in the presence of array references and to detecting inter-iteration opportunities.

Traditional value numbering has the limitation that unless two references to the same array have index expressions with the same value number, then the array references themselves will be assigned different value numbers. In the context of a loop, this results in only being able to detect when array references have the same value within a single iteration of the loop. Value numbering in a loop with induction variable `i` could thus recognize that expressions `sqrt(a(i))` and `sqrt(a(i))` are redundant, but not that expressions `sqrt(a(i+3))` and `sqrt(a(i))` are. So a key issue in detecting redundancies across loop iterations is to somehow assign meaningful value numbers to array references, even when their index expressions do not have the same value number.

A solution in the context of CSR presents itself if we recall that the key to CSR is the use of name partitions— that is, having information regarding which array references represent re-use. In other words, references in the same partition all access the same value, just on different iterations. This simple but critical observation gives us a way to value number array references. We assign the same value number to all references in the same partition. In practice, we get this effect by treating each array reference as having a new name derived from its CSR name partition. Each reference in partition 0, for example, is treated as if its name is "`?p0`".

With a method to number array references in hand, value numbering now applies to larger expressions involving those references. But there is still the issue of expression distance being taken into account. That is, value numbering should not consider the addition subexpressions

```
do i
  ... = (c(i)+b(i+1))-(b(i+2)+c(i+2))
enddo
```

redundant since the expression distance is undefined between them. This is done by modifying the hash table lookup to utilize Δ. On the first occurrence of an expression (it isn't found in the hash table), the expression is inserted into the hash table. If a subsequent expression occurrence e_1 hits in the hash table, then an arbitrary existing expression e_{ref} from the matching bucket is used as a reference point (every occurrence is inserted into the hash table, since we remove redundancies offline). We then check if expression difference $\Delta(e_{ref}, e_1)$ is defined, in which case an inter-iteration redundancy has been detected. Otherwise there is no match. Note that if expression distance is not checked, then the code snippet above would appear (incorrectly) to have a redundant addition subexpression.

Looking at the recursive definition of Δ, it might appear that the many repeated expression difference checks required would be prohibitively costly. But the actual calculation does not have to be structured in the same way the mathematical definition is (which is written for conciseness and clarity of explanation). What we do is to incrementally carry the AST leaves up the tree during the postorder traversal, so that when we compare two expression trees, all leaves necessary for the computation are immediately available. The reader will also notice that it isn't strictly necessary to carry *all* the leaves up the tree, but one from each child will do. However, carrying all the leaves makes checking for algebraic inter-iteration equivalences easier. That is, since ESR is based on value-identity, we'd like to detect that expressions such as `z(i)+z(i)` and `2*z(i+3)` are redundant.

Finally, scalar assignments require slightly more bookkeeping in inter-iteration value numbering than in the intra-iteration case. Not only must we set the value number of the scalars, but we must maintain a list of leaves associated with them. This is so that we can test expression distance between expressions that involve one or more scalars. For instance, it could be that two expressions `x/a(i+3)` and `y/a(i+2)` are redundant, but that depends on the value numbers of `x` and `y` as well as the distance between the expressions that defined `x` and `y`. In this way, we track values through assignments as in normal value numbering, but we also carry leaves through assignments so that the expression distance computation can be done efficiently.

3.2.2 Pruning Expression Groups

The value numberer iterates through every statement of the AST, numbering and inserting *every* expression into the hash table. Thus, once all expression groups are known, there are a number of uninteresting groups that need to be pruned from the hash table— that is, groups that will not be

replaced and rewritten. For example, an expression that is nested in a larger expression is automatically rewritten when the larger containing one is. Other uninteresting groups include those that only contain one occurrence, those consisting of an integer constant, etc. Once the *Groups* phase has computed all expression groups, it prunes away all those uninteresting types in preparation for *Replace*.

3.2.3 Determining Number of Scalar Temporaries

In CSR, determining the number of scalar temporaries needed to rewrite a partition is straightforward. Given partition p_i with generator r_i, the number of temporaries NT needed to rewrite p_i is

$$NT(p_i) = 1 + \max_{e \in (r_i \delta^* r_j)} \tau(e).$$

In other words, it is one more than the threshold of the dependence edge with source at the generator whose threshold is largest. This works because each partition is independent—whether or not some partition p_k is scalar replaced does not affect the number of registers that any other partition p_m will require to be replaced.

If we assume for the moment that all expression groups are independent, then an analogous formula works for the ESR scenario:

$$NT'(eg_i) = 1 + \max_{e_j \in exprs(eg_i)} \Delta(e_i, e_j),$$

where e_i is the generating expression of group eg_i with expressions $exprs(eg_i)$. However, the situation is generally more complicated in the ESR scenario since we can have overlap (nested subexpressions) among the different groups. Consider:

```
   DO K=2,N
s1   P = (A(I,J,K+1)-B(K+1))*C(K+1)*D(K+1)
s2   M = (A(I,J,K-1)-B(K-1))*C(K-1)*D(K-1)
s3   X(K) = C(K)*D(K)
s4   F(K) = P-M
   ENDDO
```

Here there is not only a "large" RSE (group 1) in s1 and s2, but a smaller nested RSE `C(...)*D(...)` (group 2) in s1, s2, and s3. If we only consider the first RSE in isolation, it is easy to see we need 3 temporaries ($\Delta = 2$). Likewise, considering only the second RSE, we also need 3 temporaries. But if we wish to eliminate both RSEs, then the number of temporaries is not the sum of them in isolation, but something different.

In fact, only five (3 and 2) temporaries are required to eliminate both RSEs, rather than six (3 and 3) if we just summed the number of temporaries for each in isolation. The reason is that the containing RSE will only occur once in the transformed code, so we have fewer occurrences of the contained RSE (and Δ of the remaining occurrences is smaller).

The challenge is in computing this algorithmically. For the overlap case above, the algorithm has to determine that in the rewritten code, if both groups are eliminated, that there won't really be three occurrences of the nested expression, but just two. More generally, if more than two groups overlap, then we need to calculate the number of temporaries needed for every combination of overlapping groups that could be eliminated.

3.2.4 Register Pressure Moderation

CSR models register pressure moderation as a *0-1 Knapsack Problem*. This problem can be solved to optimality via a pseudo-polynomial time dynamic programming algorithm. A key feature of the 0-1 KP problem is that each "object" (i.e., a name partition in CSR) has a fixed weight and profit assigned ahead of time. That is, objects are independent and choosing one object for inclusion into the knapsack has no effect on the weight or profit of any of the other objects. Name partitions satisfy this property since choosing one doesn't affect the weight or profit of the others.

Things are more complicated for expression groups in ESR since they may overlap. If some groups do overlap, then the decision to replace a redundant expression can actually determine how many registers are needed to eliminate one of the overlapping ones (and even the number of operations the other one will have). In the knapsack analogy, this is the same as saying that both the weight and profit of an object might change depending on whether or not some other object is chosen to be in the knapsack.

This new twist on the knapsack problem is critical, because it becomes strongly NP-hard. In fact, it is a generalized version of the *Quadratic Knapsack Problem* discussed in [17]. In the QKP case, only one of the object parameters change, while in our case, both do. Even the simpler QKP is strongly NP-hard, so our problem is too. This implies that, unlike 0-1 KP, there is no hope of even a pseudo-polynomial dynamic programming algorithm. We must rely strictly on a heuristic. We have developed a simple iterative heuristic to solve the problem in reasonable time that is derived from a QKP heuristic given in [17]. There is still more work needed to understand how well the algorithm works in practice on a large number of inputs.

Note that if all the expression groups happen to be independent, as they often are, then our problem reduces to 0-1 KP and we solve using the standard RPM algorithm.

4. REASSOCIATION

The ESR algorithm presented in section 3 achieves the goal of detecting a broad range of inter-iteration redundancies. However, it still carries one of the limitations of classical redundancy elimination techniques— it operates over a fixed code shape for expressions (other than allowing for simple commutativity in binary operators). Thus minor differences in expression code shape can foil the value numberer. For example, in the expressions (using scalars for simplicity) `x=a*(b*(c*d))` and `y=a*(b*(c*e))`, not a single redundancy is recognized because the innermost value number has a mismatch. That is, assuming `d` and `e` have different value numbers, then expressions `c*d` and `c*e` have different value numbers. Since value numbering performs a postorder traversal of the expression tree, it has the effect of propagating the mismatched value number up the tree. The effect is the same for value numbering applied to linear representations such as three-address code.

On the other hand, if the expressions are rearranged using associativity to be `x=((a*b)*c)*d` and `y=((a*b)*c)*e`, then the subexpression `((a*b)*c)` in the second statement will be discovered to be redundant. [2]

[2]For floating-point expressions, reassociation is not necessarily safe. Like commercial compilers that perform optimizations using the associative property (e.g., parallelization of

Often, expressions are not explicitly coded with parentheses as above. Depending on the specification of the source language being compiled, expressions are usually consistently associated (internally) to the right or to the left. So even though redundancies may be present, they can be hidden due to the shape of the code.

Despite this problem having been mentioned numerous times in the literature over decades, no one has yet proposed a solution in the context of value numbering. In order to increase the robustness of value numbering so that it is impervious to (or at least less hindered by) such changes in representation, it is desirable to automatically arrange expressions in such a way that latent redundancies are exposed. This section incorporates the idea of reassociation into value numbering (and hence ESR), so that the properties of associativity and commutativity are used to produce equivalent expressions with more redundancies.

4.1 Issues and Design of a Reassociation Algorithm

To get an idea of some of the challenging issues involved in designing a reassociation algorithm (and our high-level solutions), this section discusses some more thought-provoking examples. For this discussion, it is assumed that all chains of binary operators in the input AST have been "flattened" into equivalent operators of arbitrary arity (N-ary operators). Prefix notation is used for conciseness, and to help illustrate the shape of the corresponding ASTs.

Canonical orders and affinity.

Consider the following expression (again using scalars for simplicity):

```
/* Example "Diabolic". */
(+ (* X (+ A C I H B H)))
   (* Y (+ A B I D C E F G A C I B))
   (* Z (+ F D E G D E F G))
```

Inspecting the `+` terms carefully, it can be seen that there is a subexpression `(+ A B C I)` that occurs three times— once in the first line and twice in the second line. A second subexpression `(+ D E F G)` also occurs three times, with the first occurrence in the second line, and two more occurrences in the third line. While this is reasonably easy for a human to eyeball, the value numberer has no chance of discovering this unless the expressions are shaped so that the subexpressions become apparent. That is, the algorithm must reshape the operators so that redundancies are explicit in the AST:

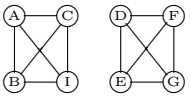
```
(+ (* X (+ (+ A B C I) H H)))
   (* Y (+ (+ A B C I) (+ A B C I) (+ D E F G)))
   (* Z (+ (+ D E F G) (+ D E F G)))
```

In this way, the redundancies can now be easily discovered by the value numberer.

The difficulty is in determining how to automatically group the operands in a satisfactory way. There must be guiding principles to be able to reorder operands for maximal (or at least more) redundancy. Some researchers have suggested the possibility of using some canonical ordering or ranking scheme for operands, such as lexical order of variable names [15]. That scheme clearly fails for our example, and even for simpler cases such as `(+ R S T S R)` where we would fail to recognize subexpression `(+ R S)`. Another ranking might be by frequency of occurrence of each variable, which would get some cases. This also fails for Diabolic since all interesting operands occur with the same frequency, so that it provides no basis for ranking. In fact, there does not seem to be any single canonical ordering that will do. This is especially true in the Diabolic example where subexpressions occur within and across terms, so that we cannot just consider reassociating one term at a time in isolation, but instead all must be considered together.

The fundamental problem is that expressions like Diabolic require a ranking that expresses affinity. That is, we wish to group together the names that have the most mutual attraction across the entire expression. To that end, we develop the following notion of affinity. For each unordered pair of of names (X,Y), the affinity A associated with that particular pair is the number of times that X and Y occur together in the same term. For example, in expression (+ R R S S S), the pair (R,S) appears twice. To be useful, the affinities for all unordered pairs of names are needed and can be represented by a symmetric $n \times n$ table for n names.

Alternatively, the affinities can be naturally viewed as an undirected *affinity graph*. That is, let each node represent a name. Then for every pair of names which have affinity greater than one, let there be an undirected edge between the two nodes representing those names. Each edge can be thought of as being labeled or weighted by the affinity. Coming back to expression Diabolic, the affinities would be computed as follows (affinities > 1). The pairs (A,B), (A,C), (A,I), (B,C), (B,I), (C,I), (D,E), (D,F), (D,G), (E,F), (E,G), (F,G) all have an affinity of three. The corresponding affinity graph is:



Once viewed graphically, we can observe that redundant subexpressions appear as maximal cliques in the graph. Thus, at a high level, the problem of grouping names/operands into redundant subexpressions reduces to the problem of determining all (or "enough") of the maximal cliques of the affinity graph. Note that nested subexpressions are also naturally handled with the method. Any sub-clique with higher affinity than the containing clique represents a nested subexpression. Finally, given the list of cliques, the algorithm mechanically rewrites the ASTs to reflect the cliques.

Despite the NP-completeness of maximal clique finding, there are efficient heuristics that work well on the types of graphs we encounter in practice. We currently use an $O(n^2)$ sequential greedy heuristic [5] (n is the number of nodes in the graph) which appears to find most of the interesting cliques we see in practice.

Value numbering with reassociation.

Thus far, we've only discussed reordering operands of expressions where we have a variable name to work with. If those were the only kind of expressions to reassociate, then a simple way to incorporate reassociation into value numbering is to run the clique finder as a pre-pass. However, in general, expressions will have operands that themselves are arbitrary expressions. Suppose we have expression `(+ (cos R) (cos R) S S S)` (where `cos` is a pure function call) and we wish to reassociate the `+` operands. We have available

sum reductions, tree height reduction), we provide a compiler option to disable reassociation when it is known to be unsafe.

the name `S`, but what is to be used for the operand `(cos R)`, which is itself a subtree (and the subtree can be arbitrarily complex)? One approach is to sidestep the issue by ignoring those operands, and only attempt to reassociate operands that are variables. But that misses opportunities as the example shows. A better approach is to assign some surrogate name to each subtree, so that redundant subtrees have the same name. The surrogate names can then be used for building the affinity graph. A key observation is that we can do exactly that by using the value numbers of the operands as surrogate names. This enables us to reorder arbitrarily complex operands of an expression.

Using value numbers as names leads to an interesting circularity. In order to do reassociation, we need value numbers. On the other hand, to do value numbering, we need reassociation. This implies that it is impossible to perform reassociation as a separate pre-pass. Instead, reassociation must be tightly intertwined with value numbering. Our approach is to, in some sense, iterate between reassociating and value numbering on portions of the AST.

In order to enable this, we change the ESR value numberer from a simple postorder traversal to a worklist approach akin to the list scheduling used in many instruction schedulers. Recall that the idea of the postorder traversal is to ensure all operands of an AST node have been numbered before the node is numbered. The same effect is achieved in the worklist approach where a "ready" list of nodes is maintained. This list is initialized with all nodes that have no children. All other nodes are iteratively added to the ready list to be numbered only after all children of each node have been processed.

To integrate reassociation into the worklist algorithm, we do not immediately put associative operators on the ready list when they become ready, but instead add them to a "deferred" list (one list for each opcode). At the point when a node becomes ready, all of its children have been numbered, so the children of *that* node could be reassociated at that moment. But recall that we wish to consider all terms of the expressions with the same operator simultaneously (e.g., Diabolic). For that reason, we do not immediately reassociate that node, but defer it and continue the numbering process as long as nodes are available on the ready list. Eventually, a point will be reached where either the numbering is finished (every node was numbered), or the process has blocked because the only remaining nodes are on the deferred list(s). It is at this blocking point that we perform reassociation, using the clique finding technique described previously. Once all deferred expressions have been reshaped, then they are added to the ready list. Note that expressions will never be deferred more than once. Finally, the worklist algorithm continues the value numbering process with the newly reassociated nodes. This will in turn enable ancestors of the reassociated nodes that were not available before because of the blockage, and some of those too may be reassociated. In this way, numbering and reassociation continues iteratively until it eventually terminates. The process is efficient– ignoring the cost of reassociating deferred nodes, the worklist algorithm runs in time linear in the number of nodes. The time is dominated by reassociation, if reassociation occurs (i.e., clique detection and alignment, see the following sections).

Inter-iteration reassociation.

The previous discussion has been using only scalar variables in expressions to simplify the explanation. That is sufficient for detecting redundancies in acyclic regions (or a single loop iteration). However, in this work what we are really after is inter-iteration redundancy detection, so that distance between expressions ($\Delta(e_1, e_2)$) must now be incorporated. The clique finder described so far groups expressions based only on their name. This means potentially that either a clique may not actually represent an inter-iteration RSE at all, or that expressions must be "aligned" in a certain way (to be explained below) to be redundant.

Take the simple case when the operands to be considered are just array references. Recall that baseline ESR represents an array reference by a name derived from the partition it was placed in during the *Partitions* phase. For the expression `(* A(I+1) A(I+3) B(I+0) B(I+3) B(I+2))`, the name partitions created by scalar replacement would be p0={A(I+1), A(I+3)} and p1={B(I+0), B(I+3), B(I+2)}. The clique finder effectively treats the expression as `(* ?p0 ?p0 ?p1 ?p1 ?p1)`. A clique representing `(* ?p0 ?p1)` would be found as expected. The problem is that this clique could represent at least two different ways of rewriting the expression, depending on what order we happen to choose the operands to rewrite. One rewrite,

```
(* (* A(I+1) B(I+0))
   (* A(I+3) B(I+2)) B(I+3))
```

is an RSE, while another rewrite

```
(* (* A(I+1) B(I+3))
   (* A(I+3) B(I+2)) B(I+0))
```

is not because the expression distance isn't defined between the two subexpressions in it.

The problem is that in the inter-iteration case, the clique represents potential RSEs. That is, we need to verify whether the clique can be written as an RSE, and then make sure we rewrite it properly. This is done by *aligning* the operands represented by each clique node so that the maximum number of references are at a constant distance from each other. That is, think of each each clique node having an associated column or vector of operands sorted by distance from the group's generator. Note that the operands can be arbitrarily complex, even though for clarity of explanation we are using simple array references. For the example, the columns would be (dashes show an empty entry):

```
Group     ?p0 [distance]    ?p1 [distance]
        A(I+3) [0]        B(I+3) [0]
         ----  [1]        B(I+2) [1]
        A(I+1) [2]         ----  [2]
                          B(I+0) [3]
```

It is clear that if we are to create redundant expressions from these operands, that we should choose `A(I+3)` and `B(I+2)` for one subexpression and `A(I+1)` and `B(I+0)` for the other. In this case, no other choice will create a redundant expression.

Algorithmically, we need to do an alignment of the columns, so that the maximum number of operands on each row line up. This corresponds to shifting up or down the columns to create matches. In the above example, there is only one alignment that makes sense, which would be done by shifting the first column down one position:

```
                    B(I+3) [0]
A(I+3) [0]          B(I+2) [1]   (match)
 ----  [1]           ----  [2]
A(I+1) [2]          B(I+0) [3]   (match)
```

Another way to view the columns is as a bit vector where a "1" is placed in a position where a reference exists, and a "0" where no reference exists. Then the best alignment can be found by iteratively trying each possible alignment of the two vectors, and bitwise ANDing the vectors into a third vector. The number of bits set in each result vector is the number of rows that had a match. The best alignment is the one that maximizes this sum. Interestingly, this is exactly the well known *Match-count Problem* which arises in computational biology among other places [16]. There is a naive way to solve this which can be done in $O(n^2)$ time for two vectors of length n. Gusfield shows a more sophisticated $O(nlogn)$ approach, but it has fairly high overhead, so that large sizes of n are likely required to beat the naive approach[3]. In practice, we see small vectors, so the naive approach is acceptable. It is the approach we adopt.

Note that the match-count problem only involves two vectors. Generally, our cliques will have more than two nodes, so there are more than two vectors. This implies we need to compute the match-count for every pair of vectors. We designed an $O(rn^2)$ approach that in theory may not perform as well as an all-pairs approach, but gives reasonable results for our tested set of loops.

The idea is to only perform $r-1$ match-counts instead of r^2. First, we order the columns from left to right by decreasing number of 1's in the column. Starting at the leftmost column 1, we compare (match-count) it with column 2. We do the same for columns 2 and 3, and so on up to $r-1$ with r. Once the match-count for a pair is found, we mark which entries matched and how many positions up or down the second column was shifted for the match. Each time a subsequent pair is matched, only rows that also had matches in the previous column match in this pair (which simplifies rewriting). Finally, two entries can only match if the operands belong to the same term in the original expression.

Once all columns are aligned, we can readily choose the redundant expressions to create. Each matching row represents one new subexpression, and all the operands marked as matched in that row will be in the new subexpression. If the alignment results in only one row with matches, then there are no redundant expressions, since that corresponds to only one expression.

5. EXPERIMENTAL RESULTS

We have implemented ESR+reassociation ("ESR+") in the Open64 compiler, as part of a high-level loop nest optimizer. Table 1 shows the direct results of applying CSR, ESR, and ESR+ to loops from SPEC2000, SPEC2006, handwritten DSP, and LANL POP. Most of the selected loops are floating-point computations, with the exception of the conservative smoothing filter and Laplacian of Gaussian filter, which use integer arithmetic. Each column shows a type of operation occurring in the source code, the original number of occurrences, and the number remaining after ESR and ESR+, respectively. A blank column indicates that either the operation does not occur in the loop, or none of those operations were removed during optimization. A boldface entry indicates a case where ESR+ improved upon ESR. Since CSR only removes memory references, and all three methods remove the same number of loads, the CSR results can be read from the "load" column. Any column other than "load" which is not empty shows a case where ESR or ESR+ removed operations not detected by CSR.

It is evident from the table that CSR already optimizes these loops well, removing many redundant loads. Even so, ESR discovers other redundancies involving arithmetic operations. These include operations that are typically expensive on most hardware, such as floating-point divide and cosine intrinsics. For example, ESR eliminated 4 of 8 `cos` calls and 4 of 8 `sin` calls in `calc_tpoints`, a substantial reduction.

The table also indicates that applying reassociation in conjunction with ESR is important. In 8 of 13 loops, reassociation allowed more redundancies to be detected than ESR alone. Moreover, in most of those cases, reassociation was necessary to detect *any* arithmetic redundancies at all. While some loops only had a modest improvement, others had dramatic reductions in the overall operation count.

It is also interesting that, at least on this set of loops, the simple $O(rn^2)$ alignment heuristic performs satisfactorily (e.g., all redundancies were detected in `pintgr`, `resid`, and others). However, one loop where the alignment did not perform well is `lapl_of_gaussian`. Despite 12 of 65 redundant additions eliminated, there are at least twice that many that could be eliminated with a stronger alignment algorithm.

App/Bench	Proc/Loop	CSR	ESR+	$\frac{CSR}{ESR+}$
LANL POP	calc_tpoints	3.6	2.4	1.5
SPEC2000	mgrid:resid	473.41	289.33	1.64
	mgrid:psinv	232.02	143.27	1.62
	mgrid:interp	30.57	27.8	1.1
	mgrid:rprj3	74.49	73.24	1.02

Table 2: Runtime (seconds) and speed-up

We also measured the runtime of some of the loops to see how the static reduction in redundant operations helped execution time on a particular machine. We targeted a 3.2GHz Intel Pentium 4 CPU box with 1GB of RAM, running Red Hat Linux FC4. We tested 172.mgrid from SPEC CPU2000 and measured the four most time consuming routines in seconds. We also extracted a kernel from the Los Alamos National Lab's Parallel Ocean Model program, which was run for a 128x128x128 problem. In Table 2 the CSR column is the time for a version optimized with classic scalar replacement, while the ESR+ column was optimized with enhanced scalar replacement.

6. RELATED WORK

The work most closely related to ours is a paper by Deitz, et. al. [11]. Their work extends common subexpression elimination to the inter-iteration case, obtaining an effect similar to ours, albeit on a more restrictive class of inputs. Also

[3]Felsenstein's implementation on early 1980's hardware indicates that the FFT approach is only faster when $n > 2000$ [14]. However, on modern hardware with fast floating-point, n might be much less today.

Loop	add	cos	div	load	max	min	mul	sin	sub	x^2
applu:rhs:2658	6/4/4	-	2/1/1	10/5/5	-	-	18/16/16	-	-	8/3/3
applu:pintgr:2433	**8/8/5**	-	-	8/4/4	-	-	-	-	-	-
swim:calc1:262	9/8/8	-	-	23/6/6	-	-	11/10/10	-	-	-
swim:calc2:316	**9/9/7**	-	-	23/11/11	-	-	-	-	-	-
mgrid:resid	**23/23/11**	-	-	32/14/14	-	-	-	-	-	-
mgrid:psinv	**27/27/15**	-	-	32/14/14	-	-	-	-	-	-
mgrid:rprj3	**26/26/20**	-	-	27/18/18	-	-	-	-	-	-
apsi:dtdtz:1476	-	-	-	15/6/6	-	-	**8/6/5**	-	4/3/3	-
apsi:smth:3443	-	-	-	9/2/2	-	-	-	-	3/2/2	-
zeusmp:forces:493	-	-	-	60/23/23	-	-	36/27/27	-	12/10/10	12/8/8
csmooth_filter	-	-	-	19/3/3	**9/9/5**	**9/9/5**	-	-	3/2/2	-
lapl_of_gaussian	**64/64/52**	-	-	65/9/9	-	-	-	-	-	-
calc_tpoints	11/8/8	8/4/4	-	16/4/4	-	-	11/7/7	8/4/4	-	-

Table 1: Static operation counts (original/ESR/ESR+)

noteworthy is that they take into account associativity to enlarge the class of common subexpressions detected. While similar opportunities are exploited by both their technique and ours, the algorithms are completely distinct and ours is more general. First, they handle only expressions in the form of a sum-of-products and the operands can only be array references (and all arrays must be of the same rank and index expressions must be of a stringent form). We handle arbitrary expressions of any form and any level of nesting, so that no assumption is made regarding whether an operand is an array reference or some arbitrary expression.

Next, our technique is based on value-numbering, and hence value-identity. This allows us to detect, e.g., algebraic equivalences across loop iterations, which their technique cannot. That is, we have the same advantages that traditional value numbering has over traditional CSE based on lexical identity. Moreover, our reassociation is more powerful since it is completely integrated with value numbering, so that reassociation applies to any level of an arbitrary expression (not just leaf expressions).

Finally, ESR is fully integrated with the standard scalar replacement framework. We thus leverage the register pressure moderation technique which is critical to avoid overapplying scalar replacement. We are able to consider ordinary scalar replacement opportunities (memory references) and enhanced opportunities (general expressions) equally and in a unified way. Deitz would have to independently apply scalar replacement as a subsequent pass.

Bodik, et. al. describe a path-sensitive dataflow analysis framework which combines value partitioning, symbolic back substitution, and dataflow analysis [3, 4]. The framework can be used to build a powerful PRE-like algorithm which can detect some values across loop iterations. The handling of loops is somewhat complicated, and requires repeatedly performing symbolic back substitution of expressions for some pre-determined window of iterations. They do not exploit associativity. The algorithm uniformly handles scalars, indirect references, and array references, potentially making it applicable to more codes than our current algorithm (which focuses on array-based codes).

An early and interesting attempt by Breuer [6] proposes a "grow factor" heuristic which incrementally builds up factors (common subexpressions) utilizing commutativity and associativity to increase the number of factors found. The technique is only applicable to a limited form for expressions and does not detect inter-iteration common subexpressions.

Briggs and Cooper discuss a global reassociation technique that aims to improve the results of code motion in a PRE-like algorithm [7]. While their algorithm does improve code motion, it doesn't necessarily lead to more common subexpressions. This technique can be seen as orthogonal to our reassociation technique.

7. FUTURE WORK

We do not currently use distributivity to increase the number of redundancies, though there are times when this is possible. The difficulty with distribution is that it may introduce *more* operations, so that any heuristic must be carefully designed and applied.

We've already seen the value of reassociation opportunities in the inter-iteration scenario. We would like to implement reassociation in a traditional acyclic value numberer, to determine experimentally how many opportunities arise in scalar and non-loop code.

We would like to extend ESR to reuse entire vectors, in the same way as Allen and Kennedy's "vector register allocation" [1] does, but for general inter-iteration expressions. Vector extensions are now becoming common in commercial CPUs, so that vector register reuse will be increasingly important.

There is much more algorithm design and experimentation that could be done in various parts of ESR+, especially alternative alignment heuristics and alternative ways of performing register pressure moderation.

Our entire heuristic approach to reassociation is based on the notion that exhaustively exploring the enormous number of ways to rearrange expressions during value numbering is not practical. But given the moderate size of some of interesting loops we see, and ever rising CPU performance, it may actually be feasible to try some sort of optimal approach for cases where the heuristics don't perform well. Even if not practical in all cases, it would set a "gold standard" by which heuristics could be measured.

Finally, it would be interesting to explore the possibility of incorporating reassociation into other redundancy elimination techniques, such as the recent PRE-based algorithms discussed in section 6.

8. CONCLUSIONS

A number of codes exhibit redundant computations that are undetectable by the classic redundancy elimination techniques. We combined classic scalar replacement with value numbering into a more powerful algorithm which detects a broad class of inter-iteration redundancies. The experimental results show that removal of these kinds of redundancies can improve runtime performance significantly, and that such computations appear in important application domains such as scientific computing and digital signal processing.

It was also seen that a number of redundancies in real applications are "hidden" in the sense that they are not directly available to a value numberer due to fixed association of operands. We describe an algorithm which incorporates into value numbering the ability to perform aggressive rearrangements (reassociation) of expressions in order to expose more redundancies. Examination of a number of applications shows that opportunities for inter-iteration redundancies are enhanced with reassociation. Before this work, no classic value numberer had this capability.

The new technique builds directly on the classic scalar replacement framework implemented in many current industrial compilers. Practitioners can thus incorporate our algorithms into their compilers with relatively modest effort.

9. ACKNOWLEDGMENTS

Luay Nakhleh discussed sequence alignment and string comparison algorithms with us. We thank the anonymous reviewers for their detailed and thoughtful suggestions.

10. REFERENCES

[1] Randy Allen and Ken Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, 1992.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[3] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–251, New York, NY, USA, 1998. ACM Press.

[4] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Load-reuse analysis: Design and evaluation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 64–76, 1999.

[5] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 4. Kluwer Academic Publishers, Boston, MA, 1999.

[6] Melvin A. Breuer. Generation of optimal code for expressions via factorization. *Communications of the ACM*, pages 333–340, June 1969.

[7] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 159–170, New York, NY, USA, 1994. ACM.

[8] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 27(6):701–724, 1997.

[9] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[10] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[11] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating redundancies in sum-of-product array computations. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 65–77, New York, NY, USA, 2001. ACM Press.

[12] Andrei P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–9, 1958.

[13] Andrei P. Ershov. *Programming Programme for the BESM Computer*. Pergamon Press, 1959.

[14] J. Felsenstein, S. Sawyer, and R. Kochin. An efficient method for matching nucleic acid sequences. *Nucleic Acids Research*, 10(1):133–139, 1982.

[15] Dennis J. Frailey. Expression optimization using unary complement operators. *SIGPLAN Notices*, 5(7):67–85, July 1970.

[16] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, January 1997.

[17] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.