Eliminating Redundancies in Sum-of-Product Array Computations*

Steven J. Deitz
University of Washington
Dept of CSE; Box 352350
Seattle, WA 98195-2350 USA
deitz@cs.washington.edu

Bradford L. Chamberlain
University of Washington
Dept of CSE; Box 352350
Seattle, WA 98195-2350 USA
brad@cs.washington.edu

Lawrence Snyder
University of Washington
Dept of CSE; Box 352350
Seattle, WA 98195-2350 USA
snyder@cs.washington.edu

ABSTRACT

Array programming languages such as Fortran 90, High Performance Fortran and ZPL are well-suited to scientific computing because they free the scientist from the responsibility of managing burdensome low-level details that complicate programming in languages like C and Fortran 77. However, these burdensome details are critical to performance, thus necessitating aggressive compilation techniques for their optimization. In this paper, we present a new compiler optimization called Array Subexpression Elimination (ASE) that lets a programmer take advantage of the expressibility afforded by array languages and achieve enviable portability and performance. We design a set of micro-benchmarks that model an important class of computations known as stencils and we report on our implementation of this optimization in the context of this micro-benchmark suite. Our results include a 125% improvement on one of these benchmarks and a 50% average speedup across the suite. Also we show a speedup of 32% improvement on the ZPL port of the NAS MG Parallel Benchmark and a 29% speedup over the hand-optimized Fortran version. Further, the compilation time is only negligibly affected.

1. INTRODUCTION

Array programming languages, such as Fortran 90 [2], High Performance Fortran [16] and ZPL [20] have proven effective in letting a scientist express a computation in a clear and concise manner. It has also been shown that aggressive compilation techniques can bring the performance level of programs written in array languages to acceptable levels. This paper presents a compiler technique to improve performance further. We demonstrate that this approach lets a scientist achieve even more performance and portability without sacrificing expressibility.

*The first author is supported by a DOE High Performance Fellowship. The second author is supported by a scholarship from the USENIX Association.

Specifically, this paper makes the following contributions:

- We introduce a new compiler optimization called *Array Subexpression Elimination* (ASE) that is critical to the performance of an important class of computations known as stencils. This optimization applies to a limited but important class of codes: Perfectly nested loops containing only straight-line code consisting of sum-of-product expressions.[1]
- We present a new compiler abstraction called a *Neighborhood Tablet* that serves as a simple framework for implementing ASE and lets the compiler deal, in a novel way, with the large number of subexpressions that exist by associativity.
- We develop a set of benchmarks called the *Stencil Micro-Benchmarks*: a collection of "Real World" stencil kernels. This benchmark suite fills a void because though there have been numerous papers written on optimizing stencil codes, there is no standard set of stencil kernels to compare approaches with. We compare the effects of ASE and Scalar Replacement on this benchmark suite.

Array Subexpression Elimination (ASE) is similar, though orthogonal, to *Loop Common Expression Elimination* first introduced by Ernst [12]. Both of these optimizations seek to extend *Common Subexpression Elimination* [3] (CSE) beyond the boundaries of loops by finding expressions that are redundantly computed in different iterations of the same loop. CSE fails in this case not only because the subexpressions are not common, but also because associativity makes the number of such expressions potentially large and unwieldy. Ernst's method relies on loop unrolling to eliminate redundant computation only in expressions involving array references that differ in a single dimension. Our method, on the other hand, does not rely on loop unrolling and applies to expressions involving multi-dimensional arrays. The actual redundant expressions found by each of these optimizations are necessarily distinct.

[1]Though we refer only to sum-of-product expressions, throughout this paper, our methods and implementation are sufficiently robust to deal with any expressions involving only sums and products, including negatives and divisions. Also our methods can apply to any operations forming a commutative ring, e.g. logical "and" and "or".

ASE subsumes a basic form of the well-studied optimization known as *Scalar Replacement* [8, 9]. In Scalar Replacement, array references are replaced by scalar variables. This is either a source-to-source transform in scalar languages or part of the scalarization process when array languages are compiled. The goal behind this transform is to achieve a better register allocation for array references in the face of standard register allocation strategies such as coloring [10].

These optimizations are critical to the performance of an important class of computations known as stencil codes. Stencil codes are commonly used in solving partial differential equations, geometric modeling and image processing. For the purpose of this paper, and in the tradition of others [7, 6, 19], we define a stencil code to be a stylized matrix computation in which a group of neighboring data elements are combined in the form of a sum of products to produce a new value. We extend this definition to include arrays or iteration spaces that are strided. Further we consider multiple source and destination arrays provided they occur in the same loop.

As a basic example of how ASE applies to a stencil, consider the canonical 2D 9-point isotropic stencil code. An isotropic stencil is a stencil in which the weights are symmetric about the center element. Figure 1 illustrates two approaches one can take to compute this stencil. The naive approach involves traversing the array and, for each array element, computing the full weighted sum of the element and its eight neighbors. This naive approach is depicted in Figure 1a. Scalar code for this computation under the naive approach is show in Figure 2a. The problem with this approach lies in the following observation: for any given weighted sum in the array traversal except the first one in each row, two partial sums from the previous iteration are redundantly computed.

To overcome this problem, a programmer could write the stencil taking the optimized approach. The optimized approach conceptually involves twice traversing each row of the array. On the first sweep over each row, we store the sum of the elements in the row above and below the current row being iterated over. On the second sweep, we compute the full weighted sum using the stored partial sums where appropriate. These two conceptual sweeps are depicted in Figures 1b and 1c. In the actual implementation, the two sweeps over the rows of the array can be combined in a single sweep and the stored sums can be stored in scalar variables. Optimized scalar code is shown in Figure 2b.

It is easy to verify that the optimization is responsible for eliminating approximately two additions and four array references per array element. Since we store the sums in scalar variables, which are likely allocated to registers, only five memory references must be made in each iteration of the loop. In the naive case, we would make nine memory references: one for each array reference in the stencil. The number of array references can be further decreased by applying Scalar Replacement to the middle row. Another two memory references per array element can be eliminated assuming, of course, that there are enough registers to meet the demands of the optimizations. On the computation front, we save the two additions by accessing three saved sums and computing one saved sum per array element.

Benchmark	Points	Use
DBIGBIHARM	1×25	Biharmonic operator
DISO3X3	1×9	Partial derivatives
DISO5X5	1×25	Partial derivatives
DLILBIHARM	1×13	Biharmonic operator
DRESID(3D)	1×21	NAS MG Benchmark
DROW3X3	1×9	Partial derivatives
DRPRJ3(3D)	1×19	NAS MG Benchmark
IBIGLAPLACE	1×97	Gradient edge detection
ILINEDET	6×8	Line detection
IMORPH	1×21	Mathematical morphology
INEVATIA	$6 \times 23.5^*$	Gradient edge detection
INOISE1	1×9	Noise cleaning
INOISE2	1×25	Noise cleaning
INOISE3	1×49	Noise cleaning
IPREWITT	2×6	Edge detection
IROBINSON	4×6	Gradient edge detection
ISOBEL	2×6	Edge detection
IWIDELINEDET	8×9	Wide line detection
IYOKOI	4×3	Connectivity number
IZEROCROSS	$10 \times 21.1^*$	Edge detection

Table 1: Classification of the Stencil Micro-Benchmarks showing the rank, type, size and use of the stencils. The type is given by the first letter of the benchmark name: "D" for floating point and "I" for integer codes. The number of points, or neighboring array references, in a stencil is in terms of the number of statements times the number of points per statement's stencil. *Since the number of points is an average over the statements, this number is not necessarily whole.

The rest of this paper is organized as follows. In the next section, we motivate this optimization by introducing the Stencil Micro-Benchmark Suite and discussing in detail the optimization of three stencils in this suite. In Section 3, we define the Neighborhood Tablet and, in Section 4, discuss how it enables ASE. In Section 5, we present experimental results. Lastly, in Section 6, we discuss related work and conclude.

2. MOTIVATION

To further motivate the need for ASE, we introduce a benchmark suite of "real world" stencil kernels. This suite is available from the Authors. Stencil codes continue to play an important role in scientific computations as well as the fields of image processing and geometric modeling. We have collected twenty kernels that span the gamut of shapes, sizes and uses. These kernels are listed in Table 1. The 2D stencil kernels are run over a 1000 by 1000 array of floating point or integer values; the 3D stencil kernels over a 100 by 100 array.

The stencils were taken from three sources. All the integer stencils can be found in a book on image processing [15]. The floating point stencils come from the domain of partial differential equations [1] and the NAS MG Parallel Benchmark [4, 5].

Without optimizing these kernels for reuse, the number of

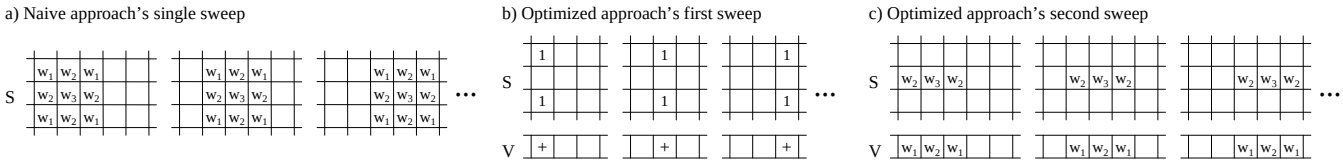

Figure 1: An illustration of two ways to compute the 2D 9-point isotropic stencil: (a) Three consecutive iterations of the computation in the innermost loop. Note the significant overlap of array references and computation. (b) Three consecutive iterations of the pre-computation (the first of two sweeps). (c) Three consecutive iterations of the optimized computation (the second sweep) using the values computed in the first sweep.

```
for i := 1 to n do
  for j := 1 to n do
    D[i,j] := w1*(S[i-1,j+1]+S[i+1,j+1]+S[i-1,j-1]+S[i+1,j-1]) +
              w2*(S[i-1,j]+S[i+1,j]+S[i,j-1]+S[i,j+1]) +
              w3*S[i,j]
```

(a)

```
for i := 1 to n do
  V1 := S[i-1,0] + S[i+1,0]
  V2 := S[i-1,1] + S[i+1,1]
  for j := 1 to n do
    V3 := S[i-1,j+1] + S[i+1,j+1]
    D[i,j] := w1*(V1+V3) +
              w2*(V2+S[i,j-1]+S[i,j+1]) +
              w3*S[i,j]
    V1 := V2
    V2 := V3
```

(b)

Figure 2: Unoptimized and optimized code to compute the 2D 9-point isotropic stencil

additions necessary is one less than the total number of points involved in the stencil. The number of multiplications is at least equal to the number of distinct weights (less any that can be eliminated by strength reduction). In our experiments, we always factor the multiplications as much as possible for both the optimized and unoptimized case. In Section 5, we report on how many additions and multiplications we are able to eliminate with this optimization.

Before delving into the details of our optimization, it is worth discussing three more examples. Figure 3 illustrates these three examples: INOISE1, INOISE2, and IYOKOI. The INOISE1 stencil is similar to the 2D 9-point isotropic stencil but contains only two weights. In addition, the value of these weights is important; in the previous example, it was enough to know the weights were equal. In the naive approach, we make one sweep over the array and for each element, compute the weighted sum given by the stencil. In this case, the computation involves doubling the element currently iterated over and adding to that product the sum of the elements in the eight neighboring positions.

The optimized approach, illustrated in Figure 3a, involves three conceptual sweeps over the rows of the array. In the first sweep, we store the sum of the elements in the row above and below the row being iterated over. This is as in the first sweep from the introductory example. In the second sweep, we store the sum of the element in the row being iterated over and the stored sum from the first sweep. Finally, in the third conceptual sweep, we compute the weighted sum using the stored sums where appropriate. The second sweep allows us to eliminate one more addition per array element. Thus, instead of eight additions in the naive case and six additions without the second sweep, we compute the weighted sum with only five additions per array element.

The INOISE2 stencil example differs from the previous two examples in that we are able to eliminate a multiplication. This type of stencil is common in the image processing domain. It is referred to as *separable* because we can separate the weights into a set of vertical weights and a set of horizontal weights. In this case, those sets are identical.

The optimized approach involves two conceptual sweeps over the array. In the first sweep, we store the weighted sum of five elements of a given column of the array: the element in the row we are iterating over, the two elements above and below this row, and the two elements above and below those. We use the vertical weights to compute this weighted sum. In the second sweep, we use the horizontal weights to compute the weighted sum of the stored sums. The optimization impact includes the elimination of sixteen additions, one multiplication and twenty array references.

Our next example involves a *multi-statement stencil*. A multi-statement stencil is a stencil computation in which multiple stencils are computed in the same loop and over the same index space. We distinguish this from a *multistencil* which is a conceptualization of multiple instances of a stencil over a single dimension [7, 6]. The IYOKOI stencil is a multi-statement stencil consisting of exactly four stencils, each a ninety degree rotation of the other. If each of these four stencils were computed separately, then we could not save any additions between two subsequent iterations. By optimizing between the stencils, we can eliminate two additions. In the first two conceptual sweeps of the array, we calculate two sums: one between the element in the row being iterated over and the element above it and the other between the element in the row being iterated over and the element below it. In the final four conceptual sweeps, we compute the weighted sums given by the four stencils and store the results in the destination arrays.

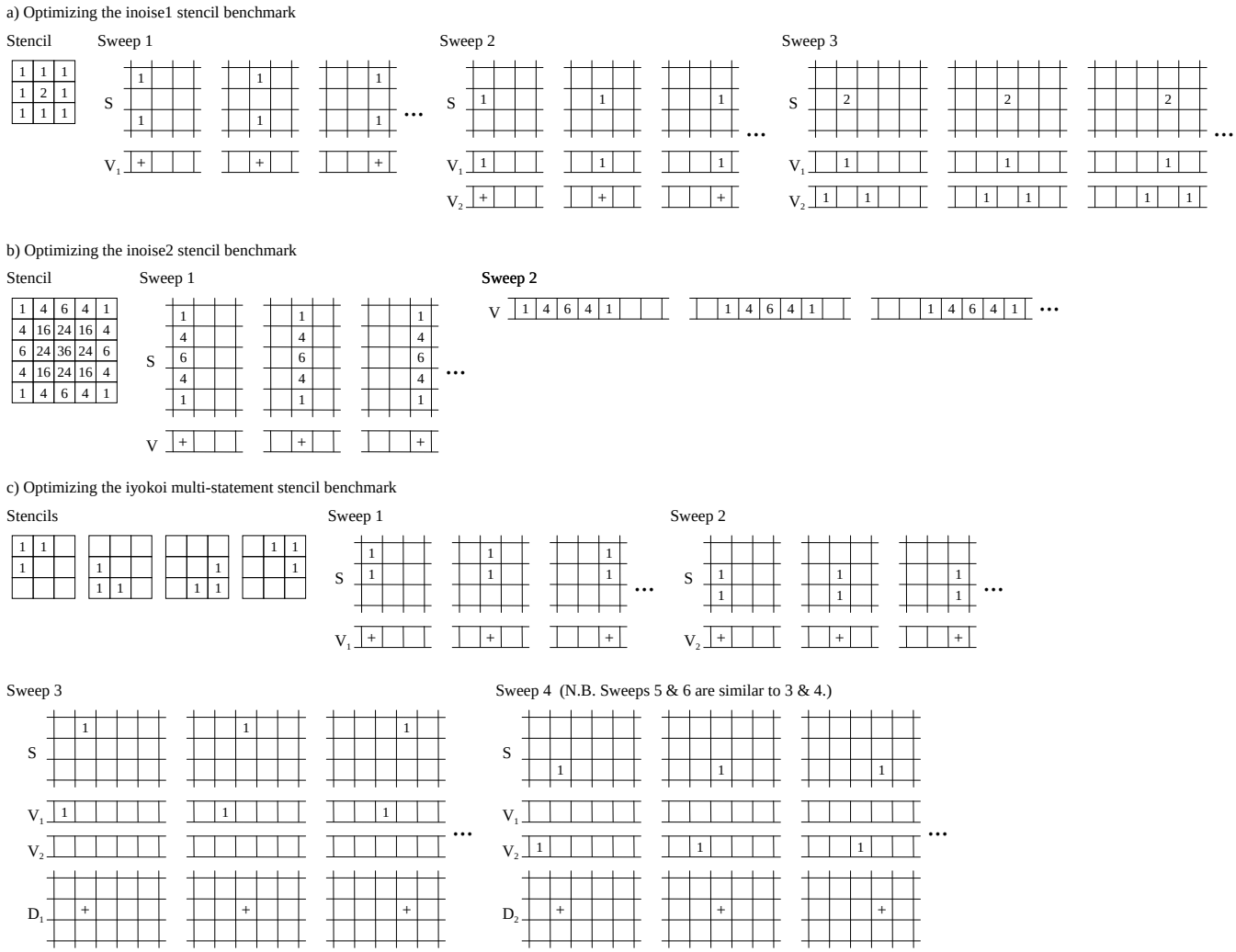

Figure 3: Motivating examples

3. TABLET CONSTRUCTION

In this section, we describe an array statement normal form and show how to translate Fortran 90 array statements into it. From this form, we show how to construct a neighborhood tablet which will form the basis of our optimization.

3.1 Normalized array statement sequences

Based on the definition of *normalized array statements* first introduced by Lewis *et al.*[17], we define a *normalized array statement sequence* to be a sequence of element-wise array operations that has the following properties: (i) the same array (or aliasing arrays) may not be both read and written, (ii) the statements contain only arrays of a common rank, (iii) the extent of the array statements' computation is defined by an index set, called a *region*, and all array references are specified as constant offsets from this index set, and (iv) the array statements are fused into a single loop. A normalized array statement sequence has the following form.

$$
\begin{aligned}
[\mathrm{R}]\ & f_1(\mathrm{A}_1@\mathrm{d}_1,\mathrm{A}_2@\mathrm{d}_2,\ldots,\mathrm{A}_{i_1}@\mathrm{d}_{i_1}) \\
& f_2(\mathrm{A}_{i_1+1}@\mathrm{d}_{i_1+1},\mathrm{A}_{i_1+2}@\mathrm{d}_{i_1+2},\ldots,\mathrm{A}_{i_2}@\mathrm{d}_{i_2}) \\
& \ldots \\
& f_m(\mathrm{A}_{i_{m-1}+1}@\mathrm{d}_{i_{m-1}+1},\mathrm{A}_{i_{m-1}+2}@\mathrm{d}_{i_{m-1}+2},\ldots,\mathrm{A}_{i_m}@\mathrm{d}_{i_m})
\end{aligned}
$$

The indices of array A_j involved in the computation are those of the region $\mathtt{R}=[1..\mathrm{n}_1,1..\mathrm{n}_2,\ldots,1..\mathrm{n}_r]$ offset by the integer r-tuple $\mathtt{d}_j=(\mathrm{d}_{j_1},\mathrm{d}_{j_2},\ldots,\mathrm{d}_{j_r})$. As an example, consider the following F90 array statements.

```
D0(1:m,1:n) = S1(0:m-1,2:n+1)+S1(2:m+1,1:n)+S2(1:m,2:n+1)
D1(1:m,1:n) = S1(2:m+1,0:n-1)+S1(0:m-1,2:n+1)+
              S1(0:m-1,1:n)+S1(2:m+1,1:n)+S2(1:m,1:n)
```

These statements form a single normalized array statement sequence that is written as follows.

```
[1..m,1..n] D0 = S1@(-1,1)+S1@(1,0)+S2@(0,1)
            D1 = S1@(-1,0)+S1@(-1,1)+S1@(1,-1)+
                 S1@(1,0)+S2@(0,0)
```

Though only normalized array statement sequences are used for the construction of the Neighborhood Tablet, unnormalized sequences do not prevent independent sequences from being optimized. This particular statement representation is appropriate for reasons discussed in the literature [17].

3.2 Generalized Tablet

The Neighborhood Tablet stores all the information we need to perform optimizations such as ASE and Scalar Replacement. The basic idea behind the tablet is to store temporal information about a pair of references to the same array as spatial information. For example, we know that the array reference given by `S1@(1,0)` refers to the same array location as `S1@(1,-1)` in the next iteration of the loop. Thus in the tablet, these two array references would map next to one another.

To construct a Neighborhood Tablet from a normalized array statement sequence we must fix the dimension traversed by the innermost loop as well as the direction of the traversal. We do not wish to store array references and expressions in scalars if they are not going to be reused in a small number of subsequent iterations so the only dimension that is important is the innermost one. Note that instead of fixing the dimension and direction, we can construct all combinations of possible Neighborhood Tablets based on all the possible combinations of innermost dimensions and traversal directions. There are $2r$ such combinations where r is the rank of the array. Typically, though, there are other more important constraints that dictate the traversal configuration such as the need to be cache friendly. Further, in the common case, we have found that if redundant computation is present, the stencil is symmetric.

The Neighborhood Tablet is a 2D array of nodes. In each node of the tablet, we store three entities: an array reference, the statement that the array reference occurs within, and a weight given by the function *wfunc* that takes the array reference as its argument. We refer to these entities with the following functions respectively, each of which take a node as an argument: *arr*, *stm*, and *wgt*. We define the functions *row* and *col* to return the row and column of a node in the tablet. These functions will be especially useful in the next section.

Array references that access the same location in memory at some point in the innermost loop are stored in the same row of the tablet. Given two array references that are next to one another in the same row, the one on the left will access the memory location of the one on the right in a subsequent iteration of the loop, assuming these array references are in the same statement.

To describe the construction of a general tablet, we provide formulas for determining the width and height of the tablet as well as for determining where each array reference is mapped. The tablet is constructed by mapping each valid array reference to a node in the tablet and storing it, the statement it occurred in, and the weight given by *wfunc*. An array reference is valid if the function *vfunc* determines so. The functions *wfunc* and *vfunc* depend on the optimization, whether ASE or Scalar Replacement, and will be defined shortly.

Let k be the dimension of the innermost loop and suppose without loss of generality that it is traversed in the upward direction. We define two functions $span(k)$ and $\overline{span}(k)$ for a given normalized array statement sequence as follows:

$$span(k) = \left[\max_{j=1..i_m} (d_{j_k}) - \min_{j=1..i_m} (d_{j_k}) + 1 \right]$$

and

$$\overline{span}(k) = \left[\sum_{s=1, s \neq k}^{r} \max_{j=1..i_m} (d_{j_s}) - \min_{j=1..i_m} (d_{j_s}) + 1 \right].$$

These functions compute the span of the offsets in the kth dimension and all but the kth dimension respectively. If there is only one statement and one distinct array in the sequence, then the width of the Neighborhood Tablet is given by $span(k)$ and the height by $\overline{span}(k)$. To account for the possibility of more than one distinct array, we multiply the height by the number of distinct arrays which we call u. Since two array references to two distinct arrays will never refer to the same memory location, we restrict these to separate rows.

We also must allow for multiple statements in the same loop. Different array references to the same array can occur in different statements and can refer to the same memory location even in the same iteration. For this reason, we keep these array references in the same row. So we expand the width of the Neighborhood Tablet by the number of statements yielding $m \times span(k)$. In addition, the value returned by $span(k)$ is rounded up so that it is divisible by the stride or step that the innermost dimension is traversed by. For our example sequence in Section 3.1, there are six rows and six columns since there are two statements, two distinct arrays, and the offsets in each dimension span from -1 to 1.

The functions that map valid array references to nodes in the tablet are as expected. We put all array references in the same statement together in a block of columns and all array references of the same array together in a block of rows. We uniquely number the arrays from one through u and the statements from one through m. Then the row and column of a given array reference is found by the following two functions respectively:

$$f_r = \sum_{t=1, t \neq k}^{r} d_t + (a-1)\overline{span}(k)$$

and

$$f_c = d_k + (s-1)span(k).$$

The Neighborhood Tablet for our example sequence from Section 3.1 is illustrated in Figure 4.

Because the Neighborhood Tablet contains a large amount of spatial information, e.g. how many iterations until two array references in the same row refer to the same memory location, a large portion of the array is possibly empty. We have found that a sparse implementation of the Neighborhood Tablet as well as the subtablets (which are built during the optimizations) are necessary to limit the time spent optimizing code.

		S1@(-1, 1)		S1@(-1, 0)	S1@(-1, 1)
	S1@(1, 0)		S1@(1,-1)	S1@(1, 0)	
		S2@(0, 1)		S2@(0, 0)	

Figure 4: An example Neighborhood Tablet.

4. ASE

In this section we discuss how the Neighborhood Tablet is used as a tool to automate array subexpression elimination. We introduce the concept of a subtablet which corresponds to a redundant array subexpression if the redundancy conditions are satisfied. We show how to generate optimized code once a set of subtablets is found. Finally, we offer a heuristic to find a desirable set of subtablets.

First, though, we define the weight and valid functions for constructing ASE's Neighborhood Tablet. The valid function is defined as follows. An array reference can be included in the neighborhood tablet if there is a weight (possibly the unit weight) such that the statement can be rewritten as the sum of a computation and the product of the array reference and its weight. Further, the weight must be invariant to the innermost loop and the array must not be assigned in the loop. The weight function returns the weight as just described in the definition of *vfunc*.

4.1 Subtablets and redundancy conditions

The naive approach to writing a stencil computation involves computing the full weighted sum of any given subtablet. However, if the redundancy conditions are satisfied, this is potentially suboptimal. We define a subtablet to be a set of nodes in the Neighborhood Tablet. Then if a subtablet satisfies the redundancy conditions, its columns represent redundantly computed sums. The optimized way of computing the portion of the weighted sum given by the subtablet is to sum up only one of the columns per array element and reuse sums for the other columns. The redundancy conditions are listed in Figure 5 using the functions from the previous section. In addition, $str(T)$ is the stride or step by which the innermost dimension is traversed.

The first redundancy condition ensures that the weighted sum of products can be factored in the way the subtablet requires it to be. The first part lets us multiply the weight by the stored sums. The second part lets us multiply the weight to compute the stored sums. The third part lets us do both. For the 2D isotropic stencil described in the introduction, it is the first part of this condition that applies. For the INOISE2 stencil, it is the third part. It can be argued that only the third part is necessary because if either of the first two parts are true, then so is the third. However, we write it this way so as to explain later how we find subtablets of different types. Also, equality is possible to determine between runtime constants at compile time, but a ration between runtime constants is indeterminable.

Redundancy Conditions. *A subtablet s with width w and height h of Neighborhood Tablet T consisting of nodes $s_{1,1}, \ldots, s_{w,1}, s_{1,2}, \ldots, s_{w,h}$ is said to satisfy the redundancy conditions if each of the following hold:*

(1) (i) $\forall_{x:1\le x\le w}\forall_{y:1\le y\le h} wgt(s_{x,1}) = wgt(s_{x,y})$ or
(ii) $\forall_{x:1\le x\le w}\forall_{y:1\le y\le h} wgt(s_{1,y}) = wgt(s_{x,y})$ or
(iii) $\forall_{x:1\le x\le w}\forall_{y:1\le y\le h} \dfrac{wgt(s_{1,1})}{wgt(s_{1,y})} = \dfrac{wgt(s_{x,1})}{wgt(s_{x,y})}$

(2) $\forall_{x:1\le x\le w}\forall_{y:1\le y\le h} row(s_{1,y}) = row(s_{x,y})$

(3) $\forall_{x:1\le x\le w}\forall_{y:1\le y\le h}$
$col(s_{1,1}) - col(s_{x,1}) = col(s_{1,y}) - col(s_{x,y})$

(4) $\forall_{x:1\le x\le w}\forall_{y:1\le y\le h} stm(s_{x,1}) = stm(s_{x,y})$

(5) $\forall_{x:1\le x\le w}\forall_{y:1\le y\le h} col(s_{1,y}) = col(s_{x,y}) \bmod str(T)$.

Figure 5: Redundancy Conditions for ASE

The second and third conditions ensure that the redundantly accessed memory is redundant. Condition two forces the rows of the subtablet to align to the rows of the Neighborhood Tablet. The row in the Neighborhood Tablet by definition consists of redundantly accessed memory locations. Condition three ensures that these redundantly accessed memory locations occur on the same iterations in every row. We can only eliminate the redundant sum if the sum happens in the same iteration.

Conditions four and five are relevant only to certain types of stencils. Condition four pertains only to multi-statement stencils. It is necessary to ensure that the sum is computed in the same statement. Condition five pertains only if we are traversing over the array with a stride greater than one. Here we do not want to mistakenly identify a sum as redundant if it is skipped in a subsequent iteration because the stride is too large.

4.2 Heuristic approach

4.2.1 Growing subtablets

Because larger subtablets correspond to larger amounts of redundancy, we take the greedy approach of searching over only maximal subtablets. A subtablet is maximal if it can be made neither wider nor higher. Depending on the exact formula we use to determine the benefit of a subtablet, it is possible that the best solution does not contain any maximal subtablets. Since the number of subtablets is exponential (even the number of maximal subtablets is exponential) and we let the benefit of a given subtablet be configurable, we are forced to use a heuristic so that the optimization runs in a reasonable amount of time.

Our algorithm to find a set of subtablets starts by finding a candidate set of maximal subtablets. We then choose from this candidate set the subtablet with the greatest benefit, as defined by a configurable benefit function, and recurse on whatever is left in the Neighborhood Tablet minus those nodes in the chosen subtablet. In addition, we insert a new row in the neighborhood tablet corresponding to the subtablet just removed. We call this step reinsertion since we reinsert some pseudo array references that relate to the subtablet just removed. This step lets us optimize stencils like

the INOISE1 stencil illustrated in Figure 3a. The algorithm for identifying the candidate set is shown in Figure 6.

```
INPUT T :  a Neighborhood Tablet

OUTPUT Candidates :  a set of subtablets

1   Candidates = φ
2   foreach part i of redundancy condition 1
3      Candidates_i = φ
4      foreach 2 × 2 subtablet s in T
5         if s ∉ s' forall s' ∈ Candidates_i
6            widen s
7            heighten s
8            Candidates_i = Candidates_i ∪ {s}
9      Candidates = Candidates ∪ Candidates_i
10  return Candidates
```

Figure 6: Algorithm to find the candidate set of subtablets for ASE

In this algorithm, we start with all possible subtablets of height two and width two. We first expand these subtablets horizontally and then vertically using functions in lines 6 and 7 that are easy to reason about. We add each expanded subtablet to the candidate set. To save computation time, we avoid expanding subtablets that are contained in a maximal subtablet already in the candidate set.

This algorithm is appropriate because it is fast and yet still finds a large set of interesting subtablets. It effectively prunes out the two by two subtablets that would lead to the same maximal subtablet. In addition, the use of each of the three parts of redundancy condition 1 separately help us avoid an overwhelming number of the separable subtablets in case the number of multiplications is a big concern. Those subtablets that satisfy the first part of the first redundancy condition do not affect the number of multiplications. Those that satisfy the second part can possibly increase the number of multiplications and those that satisfy the third part can increase or decrease the number of multiplications.

4.2.2 Benefit function

To decide which of the candidate subtablets is chosen to optimize the code, we use a benefit function that returns higher numbers for potentially better subtablets. The benefit of a subtablet is given by the following formula

$$
\begin{aligned}
&B1 \times (\text{number of adds eliminated}) + \\
&B2 \times (\text{number of multiplications eliminated}) + \\
&B3 \times (\text{number of array references eliminated}) + \\
&B4 \times (\text{number of scalar variables used})
\end{aligned}
$$

The number of adds eliminated is easy to calculate. In the naive weighted sum approach we compute $wh-1$ additions. In the optimized approach we compute $h-1$ additions to presum the columns and then $w-1$ additions to add the column sums which leads to a total of $h-1+w-1$ additions. So the total number of additions eliminated is given by $(w-1)(h-1)$. The number of array references eliminated is similarly calculated to be $(w-1)h$.

We eliminate or add multiplications based on the weights. In the naive case, the number of multiplications is given by the number of distinct weights not counting one or its negative. The number in the optimized case is given by the sum of the distinct number of multiplications in each pass.

The number of variables used is important because it corresponds to the number of registers needed to eliminate the array references. It is easy to determine from a given subtablet and is equal to $span(k)$ where k is the innermost dimension, *span* is as previously defined, and we consider only nodes in one row of the subtablet.

The variables $B1$ through $B4$ configure tradeoffs between multiplications, additions, array references and additional variables. We use separate configurations for floating point and integer codes since the number of floating point registers often differs from the number of integer registers and the latency of floating point operations is typically higher than integer operations. Note that $B4$ should be negative while the others are positive. Since these numbers are machine dependent, we do not give values for them here. In future work, we plan to look at a method for determining these values automatically possibly with a test suite of stencils.

4.2.3 Benefit thresholds

The number of subtablets can adversely affect the performance. By using more variables and requiring more registers, it could be the case that ASE is detrimental. In our implementation of the optimization in the ZPL compiler, we produce C code. For this reason, we do not have access to the register allocater. So how do we know if adding another small subtablet will cause the creation of some spill code that will slow the program down? We experimented with a number of heuristics to cope with this problem and found a simple approach to suffice.

Counting the number of variables inserted into the code and not adding any more subtablets after a given limit has been reached proved ineffectual. No single limit seemed to suffice. A spill might be appropriate if the number of eliminated additions is large enough. Modifying this scheme by making the limit flexible (e.g. increasing the necessary benefit to add the subtablet once some limit of variables is reached) also failed. It is difficult to ascertain the number of registers needed to implement the rest of the loop efficiently. This is machine/compiler dependent.

We found that, though not perfect, simply requiring the benefit of every subtablet to be above a threshold was adequate, although not perfect: In certain cases we could occasionally do better if we were to have a lower threshold or none at all. This assured that the optimized code was never slower than the naive code; this case occurred if we added extra scalar variables without eliminating a large enough number of additions and array references.

4.2.4 Example subtablets

As an example, let us continue to optimize the Fortran 90 statements from Section 3.1. There are two candidate subtablets in the Neighborhood Tablet from Figure 4. These are illustrated in Figures 7a and 7b.

S1@(-1, 1)	S1@(-1, 0)	S1@(-1, 1)
S1@(1, 0)	S1@(1,-1)	S1@(1, 0)

(a)

S1@(-1, 1)	S1@(-1, 0)
S1@(1, 0)	S1@(1,-1)
S2@(0, 1)	S2@(0, 0)

(b)

Figure 7: Two candidate subtablets

The subtablet in Figure 7a has a height of two and a width of three. The number of array references eliminated is four; the number of additions eliminated is two; and the number of scalar variables used to generate the code is three. There are no multiplications in this example. For the other candidate subtablet, we would eliminate two additions and three array references, but use only two variables. Depending on the values of B1 through B4, the benefit function would let the compiler decide which to use.

Assume for this example that the subtablet in Figure 7a is used to optimize the statements. Then we insert a new row into the Neighborhood Tablet based on this subtablet and remove the nodes contained in the subtablet. The Neighborhood Tablet, after reinsertion, is depicted in Figure 8.

		S2@(0, 1)		S2@(0, 0)	
	SUBTAB1		SUBTAB1	SUBTAB1	

Figure 8: The example Neighborhood Tablet after Reinsertion.

From this new Neighborhood Tablet, we are able to find one candidate subtablet consisting of the two remaining array references and the first two scalar variables used by our first subtablet and depicted by the two leftmost `SUBTAB1` nodes in the bottom row of the Tablet. Thus, we eliminate one more addition and array reference. Had we chosen the subtablet of height three rather than width three initially we would not be able to eliminate this extra addition. We would have only eliminated two. For this reason, we give a higher absolute value to $B3$ than $B4$ so as to favor wider subtablets.

4.3 Scalarization and code generation

After finding the subtablets, we must optimize the actual code. For array languages, this means scalarizing the code. In the case of the ZPL compiler, we produce optimized C code. For scalar languages, it means completing a source-level transform. For the purpose of simplifying this exposition, we show a Pascal-like syntax where indenting corresponds to *begin-end* blocks rather than the actual C code produced by our compiler.

There are three basic approaches to generating this optimized code: *scalar*, *unroll* and *vector*. The unoptimized version of our running example is below.

```
for i := 1 to n do
  for j := 1 to n do
    D0[i,j] := S1[i-1,j+1]+S1[i+1,j]+S2[i,j+1]
    D1[i,j] := S1[i-1,j]+S1[i-1,j+1]+S1[i+1,j-1]+S1[i+1,j]+S2
```

We show the *scalar*, *unroll*, and *vector* versions of the code in Figure 9. The *scalar* approach involves computing the partial sums as we traverse the array. In the case of our example, we have two subtablets corresponding to two partial sums. We store the partial sums in scalar variables that we shift. Since the span of each of these subtablets measures two, we need only two scalar variables for each subtablet. The use of the first subtablet in the second (reinsertion), is seen below in that the scalar variables for the second subtablet, `V2_1` and `V2_2`, are computed from the scalar variables for the first subtablet.

The scalar variables are shifted at the bottom of the innermost loop so that the sums are kept current. The clever reader will notice that the shifting of the first subtablet is unnecessary since its earlier references are not used; they have been subsumed by the second subtablet. In our implementation, we leave the elimination of this statement to the C compiler and dead assignment elimination.

Though there is only one necessary shift of scalar variables in our example, often there are more. In the *unroll* approach, we avoid shifting the scalars by unrolling the loop. Each instance of the computation in the unrolled loop refers to different scalars as the meanings of these scalars shift. In addition to the code in Figure 9, a loop similar to the *scalar* version must be used on the last few iterations since the size of `n` is not necessarily a multiple of the unrolling factor, the maximum of the bounding widths of the subtablets. In this case, it is two.

The *vector* approach to generating code is closest to the way we thought about optimizing this stencil with the two conceptual sweeps over the array. In the first sweep, we do any partial computations which in our example involves the two precomputed sums. In the second sweep, we use the vector of precomputed sums to compute the full weighted sum.

If we are generating the vector version of the code, we must change the benefit function. By finding subtablets, we would no longer eliminate array references. Instead, we would increase the number of array references. We also would ignore the number of variables and count instead one vector per subtablet.

4.4 Scalar Replacement

As stated earlier, Scalar Replacement is an optimization in which array references are replaced by scalar variables to achieve a potentially better allocation of registers and eliminate redundant array references. To modify the Neighborhood Tablet so that it is suited to Scalar Replacement,

```
for i := 1 to n do
V1_1 := S1[i-1,1]+S1[i+1,0]
V2_1 := V1_1+S2[i,1]
for j := 1 to n do
  V1_2 := S1[i-1,j+1]+S1[i+1,j]
  V2_2 := V1_2+S2[i,j+1]
  D0[i,j] := V2_2
  D1[i,j] := V1_2+V2_1
  V1_1 := V1_2
  V2_1 := V2_2
```

(a)

```
for i := 1 to n do
V1_1 := S1[i-1,1]+S1[i+1,0]
V2_1 := V1_1+S2[i,1]
for j := 1 to n by 2 do
  V1_2 := S1[i-1,j+1]+S1[i+1,j]
  V2_2 := V1_2+S2[i,j+1]
  D0[i,j] := V2_2
  D1[i,j] := V1_2+V2_1
  V1_1 := S1[i-1,j+2]+S1[i+1,j+1]
  V2_1 := V1_2+S2[i,j+2]
  D0[i,j] := V2_1
  D1[i,j] := V1_1+V2_2
```

(b)

```
for i := 1 to n do
  for j := 0 to n+1 do
    V1[j] := S1[i-1,j]+S1[i+1,j-1]
    V2[j] := V1[j]+S2[i,j]
  for j := 1 to n do
    D0[i,j] := V2[j+1]
    D1[i,j] := V1[j+1]+V2[j]
```

(c)

Figure 9: Three versions of ASE optimized code for our running example: (a) scalar, (b) unroll, and (c) vector.

we need simply change the weight and valid functions. For Scalar Replacement, these functions become trivial. There is no weight associated with any array reference, so this function is ignored. The valid function always returns true; all array references are valid for Scalar Replacement.

The subtablets are defined to be any set of tablet nodes such that each node is in the same row of the tablet. This is the second redundancy condition for ASE. For space reasons, we leave out of this discussion how to find a good set of such subtablets. It is a trivial exercise for the interested reader to come up with a reasonable heuristic.

One caveat of implementing this optimization pertains to those array references for which the array is assigned a value in the loop. Recall that we consider such array references invalid for ASE. For Scalar Replacement to be correct, we must be sure to assign values not only to this array, but also to any scalars which we generate to replace this array's references.

We have found that it is wise to combine Scalar Replacement and ASE when generating scalar or unrolled code for ASE. We use the more restrictive weight and valid functions of ASE for this purpose and compare the subtablets under the same benefit function described earlier. The subtablets for Scalar Replacement gain no points for eliminating additions and multiplications, but do gain and lose points for using variables and eliminating array references respectively. We continue with a second pass over Scalar Replacement to possibly optimize any array references not valid for ASE.

4.5 A Further Extension

Consider the classic Jacobi stencil as given below.

```
D(1:m,1:n) = (S(0:m-1,1:n)+S(1:m,0:n-1)
              S(2:m+1,1:n)+S(1:m,2:n+1))/4
```

At first glance, there is no computation to eliminate in the inner loop. But we can eliminate one addition if we compute two rows simultaneously. The framework we have developed is easy to extend to do just this. The resulting Neighborhood Tablet is shown in Figure 10. The key idea is to treat the stencil as two. Except for a small change to the code generation procedure to allow the outer loop to increment by two, no other modifications need to be made.

	S@(-1, 0)				
S@(0,-1)		S@(0, 1)		S@(-1, 0)	
	S@(1, 0)		S@(0,-1)		S@(0, 1)
				S@(1, 0)	

Figure 10: The Neighborhood Tablet for simultaneously computing two rows of the Jacobi stencil.

5. EXPERIMENTAL EVALUATION

In this section we evaluate our implementation of the Neighborhood Tablet in the ZPL Compiler. We focus on the NAS MG Parallel Benchmark and the Stencil Micro-Benchmark Suite. We compare ASE against scalar replacement as it pertains to the stencil benchmarks

5.1 NAS MG Benchmark

A performance study of the NAS MG Benchmark across a number of high-level array languages and parallel platforms is already reported in the literature [11]. The optimizations discussed in this paper proved crucial to the performance/expressibility ratio of the ZPL version of this benchmark. If the compiler does not support this optimization, it is necessary to hand-code it to achieve comparable performance. Hand-coding is an error prone task and results in code that is more difficult to maintain and less portable.

The NAS MG Parallel Benchmark involves a number of stencils where the weights are known at compile-time. It is thus a good match for ASE. In four important routines, stencils account for much of the time. Two of these stencils, those from the residual and projection routines, are also in the Stencil Micro-Benchmarks as RESID and RPRJ3 respectively.

Figure 11 shows the results of applying ASE to the NAS MG Benchmark. The machine used for this experiment is a 500MHz Pentium P-III with 0.938 GB of RAM. For the Fortran codes, we used the GNU g77 compiler version 0.5.25 with the -O3 flag. For the C compiler serving as the back-

end of the ZPL compiler, we used the GNU gcc compiler version 2.95.2 with the -O3 flag.

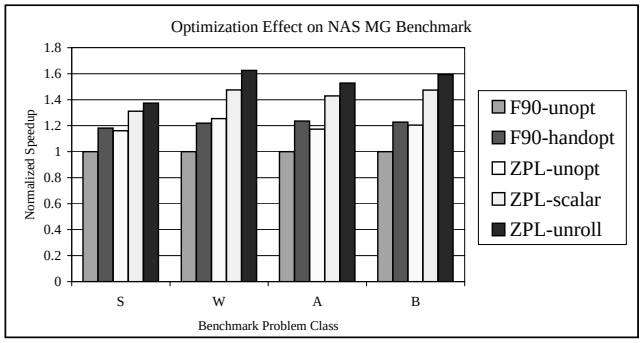

Figure 11: Optimization impact on the NAS MG Benchmark on a 500 MHz Pentium P-III

The Fortran version is the official version of the benchmark and is written in a mix of Fortran 90 and Fortran 77. The stencils are written in Fortran 77 so that they can be hand-optimized. Figure 11 shows the speedup resulting from this optimization over all but the largest problem class size. This largest size could not run on a single processor.

We normalize to the Fortran version of the benchmark without the hand-coded optimization. We removed the optimization from the official version and replaced it with a naive implementation of the stencil. The hand-coded optimization in the Fortran version achieves roughly a twenty percent speedup on all the problem sizes. The hand-coded optimization is similar to the vector version of the code produced by the ZPL compiler. The authors of the NAS MG Benchmark decided not to optimize for scalar machines because that would devastate the performance on vector processors. So to achieve portability, the benchmark was optimized for a vector processor with the hope that this optimization would then also benefit scalar processors.

The ZPL unoptimized version takes slightly more time than the Fortran optimized version on average. The ZPL compiler produces highly optimized C code that is further optimized by the underlying C compiler. When the ZPL compiler optimizes the code using either the *unroll* or *scalar* version of code generation, performance improves by up to 32%.

On a parallel supercomputer, we see less of an impact. We compare the ZPL optimized and unoptimized versions on a 256 node CRAY T3E where each node is an Alpha with 0.256 GB RAM running at 450 MHz. The back-end C compiler is the CRAY cc compiler and we used the -O3 flag on both our unoptimized and optimized codes, as always. Figure 12 shows the results on both the B and C classes of the benchmark. The class A results are similar to the class B results and are omitted.

As more processors are thrown at the problem, without varying the problem size, the optimization becomes less significant. The speedup of the optimized version is less than the speedup of the unoptimized version. It is easy to parallelize unnecessary work. Shown in the graph, however, is the speedup of both versions normalized to the fastest running time of either version on the smallest number of processors for which we have timings. In all cases, the optimized version is faster. The percentage faster remains relatively constant on the largest problem size, decreasing from 19.5% to 16.3% between 16 processors and 216 processors.

5.2 Stencil Micro-Benchmarks

The stencil micro-benchmarks give us a better look at the optimization as it directly affects stencils. Our optimization as implemented in the ZPL compiler is able to eliminate a large number of additions, multiplications and array references from the naive stencil codes. These numbers are reported in Table 5.2 along with the number of scalar variables used to do so. We prepend the letter "D" or "I" to the benchmark to signify that it is either a floating point code or an integer code respectively. Also reported are the number of array references eliminated solely by Scalar Replacement, abbreviated in the table by "SR", and the number of scalar variables used to do so. We implemented the scalar replacement optimization by turning off the stencil part of ASE, i.e., by only finding subtablets of height one. Since the stencil kernels are simple loops without conditional control flow in the inner-loop, a more complicated form of scalar replacement is unnecessary. There are no missed opportunities. As noted in the literature [9], it is easy to overoptimize with Scalar Replacement. To ensure that we are comparing against a good version of scalar replacement, we compare against an adjusted form. The adjusted form is Scalar Replacement run with a limit on the number of replacements. We chose the limit so as to achieve the fastest running time.

The benefit bar discussed in Section 4.2.3 keeps us from over-optimizing these stencils. In particular, we do not attempt to optimize the IYOKOI benchmark at all. The benefit gained is too small and performance degrades because of the more complicated C code that the ZPL compiler would produce.

Figure 13 shows the effect of the optimization across all of the Stencil Micro-Benchmarks as well as class S of the NAS MG benchmark on a 400 MHz Pentium III. For this experiment, the C compiler serving as the back-end to the ZPL compiler is the GNU gcc compiler version 2.91.66 with the -O3 flag.

Scalar Replacement with unrolling achieves on average a 27% improvement over the unoptimized code, whereas ASE with unrolling achieves a 50% average improvement. The advantage of ASE comes in part from the elimination of computation which Scalar Replacement does not do. It is also a result of the introduction of less scalar variables. For the integer benchmarks, ASE improves over Scalar Replacement by 14% whereas for the floating point benchmarks, ASE improves over Scalar Replacement by 25%. This difference is a result of increased register pressure in the floating point codes. Since far more scalar variables are used for Scalar Replacement than ASE (the advantage of using one variable to store the sum of multiple array references is lost), the resulting C code is more complex.

A major benefit of the approach taken by ASE is seen in the optimization of the INOISE2 stencil which is separable. For this stencil, ASE achieves a 125% speedup. The effect of reinsertion and optimizing between stencils in multiple array statements also has a positive impact on the total

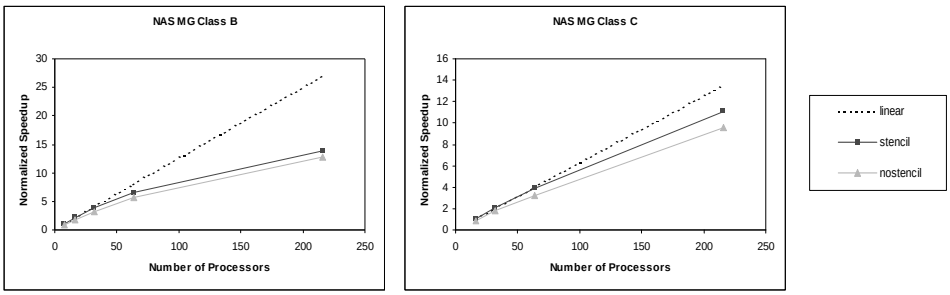

Figure 12: Optimization impact on the NAS MG Benchmark on a CRAY T3E

					SR ΔARR		SR VARS	
Benchmark	**ΔADD**	**ΔMUL**	**ΔARR**	**VARS**	MAX	ADJ	MAX	ADJ
DBIGBIHARM	-8	+1	-15	11	-23	-20	23	20
DISO3X3	-2	0	-4	3	-9	-9	9	9
DISO5X5	-9	0	-12	5	-25	-25	25	25
DLILBIHARM	-2	0	-4	3	-11	-3	11	3
DRESID	-10	+1	-13	6	-20	-17	24	21
DROW3X3	-4	0	-6	3	-9	-9	9	9
DRPRJ3	-6	0	-8	3	-15	-6	15	6
IBIGLAPLACE	-52	+8	-93	58	-97	-97	97	97
ILINEDET	-20	0	-37	13	-48	-48	9	9
IMORPH	-13	0	-20	10	-21	-21	21	21
INEVATIA	-66	+3	-100	30	-136	-136	25	25
INOISE1	-2	0	-4	3	-9	-9	9	9
INOISE2	-16	-1	-20	5	-25	-25	25	25
INOISE3	-25	+1	-36	16	-49	-49	49	49
IPREWITT	-4	0	-7	6	-12	-12	9	9
IROBINSON	-6	0	-17	8	-24	-24	9	9
ISOBEL	-4	0	-7	6	-12	-12	9	9
IWIDELINEDET	-23	+1	-41	19	-72	-66	41	37
IYOKOI	0	0	0	0	-12	-4	9	3
IZEROCROSS	-151	+4	-193	20	-211	-211	25	25

Table 2: Optimization effect on the Stencil Micro-Benchmarks of both ASE And Scalar Replacement. Columns two through four show the change in the number of additions, multiplications and array references in the produced C code after the ASE optimization. Column five shows how many scalar variables are introduced in the C code. Columns six and seven show the relevant analogous quantities for Scalar Replacement.

performance shown. Through reinsertion, we achieved a 5% speedup on the stencil kernels that were affected and, by optimizing between statements, we improved by 9% on the affected benchmarks.

6. CONCLUSIONS AND RELATED WORK

The importance of stencils to scientific computing is behind the rich history of techniques to optimize such computations. In 1991, Bromley *et al.* [7] developed and reported on a specialized stencil compiler for the Connection Machine CM-2. Three years later, Brickner *et al.* [6] extended this work for the CM-5. Their appropriately named *Convolution Compiler* is designed specifically for the Connection Machine computers, though a number of ideas described in their papers were general purpose, especially those focusing on improving inter-processor communication. The compiler used a special register allocation scheme to eliminate redundantly loading the same locations in memory and a library of hand-optimized microcode to eliminate unnecessary communication for specific stencil patterns. Only stencils specified with HPF's `CSHIFT` intrinsic were detected.

Roth *et al.* [19] do similar optimizations to the convolution compiler including the elimination of redundant loads to memory, but their strategy is more general. It involves detecting stencils not specified solely through `CSHIFT` intrinsics but also those specified with F90 array syntax. In addition to recognizing more stencils, they also perform more general optimizations that apply to more types of stencils and work on a variety of platforms. Again, they focus mostly on parallel performance and attempt to eliminate redundant communication; their focus is not on redundant computation.

Interestingly, at about the same time, work on optimizing stencil computations was being done to serialize parallel programs. Here the focus was indeed on eliminating computation. Ernst [12] discusses eliminating redundant computation in stencils much like we do, but his focus is only on one-dimensional stencils. His methods rely on loop unrolling and loop differencing to uncover the redundant computa-

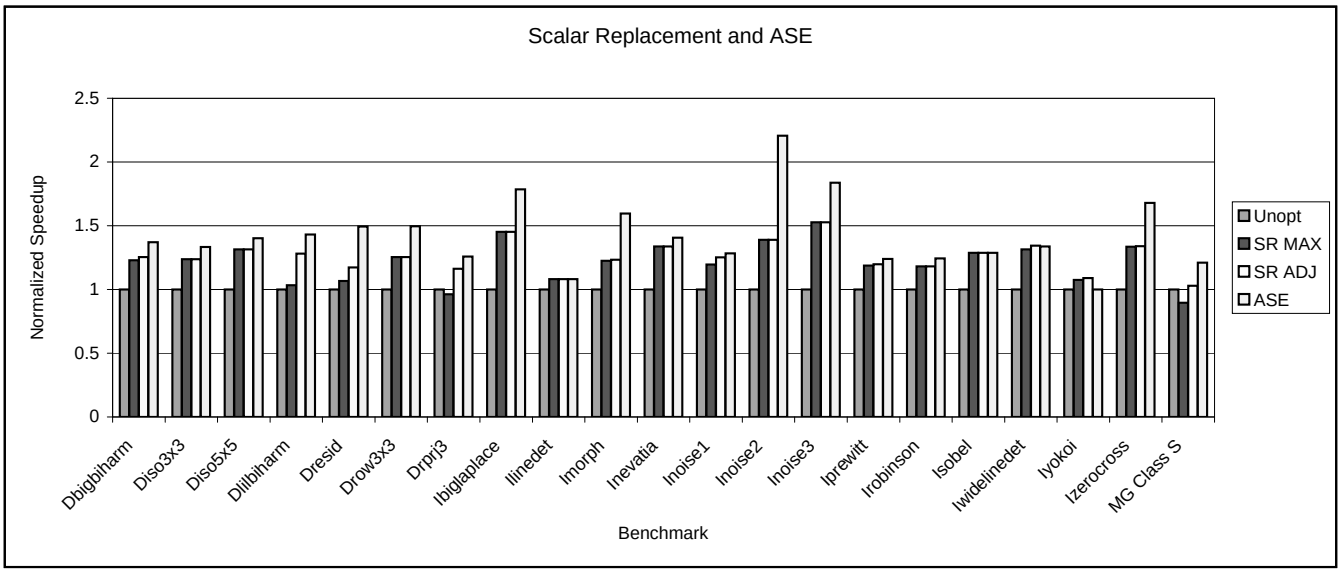

Figure 13: The effect of ASE and Scalar Replacement on the Stencil Micro-Benchmark Suite and the NAS MG Benchmark

tions. Our work focuses only on multi-dimensional stencils and finds necessarily orthogonal redundancies.

In a similar vein, Liu and Stoller [18] optimize aggregate array computations. They effectively optimize multidimensional stencils when the stencil is expressed with loops, but do not consider stencils with more than one weight.

Fisher *et al.* [13, 14] optimize two-dimensional stencils much like we do, but focus on compiling for SIMD machines using a specialized language feature called "directionals." Their work does not scale to large stencils and does not work on modern MIMD-style machines.

Scalar Replacement is a well-studied optimization technique [8]. Our subsumption of this optimization is limited when compared to some approaches [9] in that we do not detect possible applications in the presence of control flow.

In conclusion, we have developed a new technique to cope with the myriad of common subexpressions that span across loop boundaries in large stencil kernels. We have demonstrated the dire performance costs of not implementing this optimization, either by hand or automatically, and have argued that the automatic approach is more desirable.

Future directions of this work include increasing the breadth for which this algorithm applies including determining if it is possible to alter traditional common subexpression elimination algorithms to take advantage of the neighborhood tablet and achieve better performance over highly associative expressions. It is also desirable to create a better mechanism for optimizing towards a particular machine at the ZPL level of compilation. Currently, all machine specific optimizations are done by the underlying C compiler. Flags are used to switch between the vector, scalar and unroll versions of the code we produce.

7. ACKNOWLEDGMENTS

A portion of the results were obtained with a grant of HPC time from Los Alamos National Laboratory and the Arctic Region Supercomputing Center for which we are grateful. We would like to thank E. Christopher Lewis for some insightful comments during the early stages of this work.

8. REFERENCES

[1] M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions, with formulas, graphs, and mathematical tables.* Dover Publications, 1973.

[2] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook.* McGraw-Hill, 1992.

[3] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks (94). Technical report, RNR Technical Report RNR-94-007, March 1994.

[5] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical report, NAS Report NAS-95-020, December 1995.

[6] R. G. Brickner, K. Holian, B. Thiagarajan, and S. L. Johnson. A stencil compiler for the connection machine model cm-5. Technical report, Center for Research on Parallel Computation CRPC-TR94457, June 1994.

[7] M. Bromley, S. Heller, T. McNerney, and G. L. S. Jr. Fortran at ten gigaflops: The connection machine

convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[8] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, 1990.

[9] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software – Practice and Experience*, 24(1):51–77, January 1994.

[10] C. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:45–57, January 1981.

[11] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the nas mg benchmark across parallel languages and architectures. In *Proceedings of Supercomputing '00: High Performance Networking and Computing*, 2000.

[12] M. D. Ernst. Serializing parallel programs by removing redundant computation. Technical report, Microsoft Research Technical Report MSR-TR-94-15, August 1994.

[13] A. L. Fisher and P. T. Highnam. Communication and code optimization in simd programs. In *International Conference on Parallel Processing*, 1988.

[14] A. L. Fisher, J. Leon, and P. T. Highnam. Design and performance of an optimizing simd compiler. In *Frontiers of Massively Parallel Computation*, 1990.

[15] R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision*. Addison-Wesley, 1992.

[16] High Performance Fortran Forum. *High Performance Fortran Langauge Specification, Version 2.0*. January 1997.

[17] E. C. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[18] Y. Liu and S. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, 1998.

[19] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in high performance fortran. In *Proceedings of Supercomputing '97: High Performance Networking and Computing*, 1997.

[20] L. Snyder. *Programming Guide to ZPL*. MIT Press, 1999.