Loop optimization for aggregate array computations

Yanhong A. Liu* and Scott D. Stoller*

Abstract

An aggregate array computation is a loop that computes accumulated quantities over array elements. Such computations are common in programs that use arrays, and the array elements involved in such computations often overlap, especially across iterations of loops, resulting in significant redundancy in the overall computation. This paper presents a method and algorithms that eliminate such overlapping aggregate array redundancies and shows both analytical and experimental performance improvements. The method is based on incrementalization, i.e., *updating the values of aggregate array computations from iteration to iteration rather than computing them from scratch in each iteration. This involves maintaining additional information not maintained in the original program. We reduce various analysis problems to solving inequality constraints on loop variables and array subscripts, and we apply results from work on array data dependence analysis. Incrementalizing aggregate array computations produces drastic program speedup compared to previous optimizations. Previous methods for loop optimizations of arrays do not perform incrementalization, and previous techniques for loop incrementalization do not handle arrays.*

1 Introduction

We start with an example—the local summation problem in image processing: given an n-by-n image, compute for each pixel $\langle i, j\rangle$ the sum $sum[i, j]$ of the m-by-m square with upper left corner $\langle i, j\rangle$. The straightforward program (1) takes $O(n^2m^2)$ time, while the optimized program (2) takes $O(n^2)$ time.[1]

```
for i := 0 to n−m do
  for j := 0 to n−m do
    sum[i, j] := 0;
    for k := 0 to m − 1 do                                   (1)
      for l := 0 to m − 1 do
        sum[i, j] := sum[i, j] + a[i+k, j+l]
```

```
for i := 1 to n−m do
  for j := 1 to n−m do
    b[i−1+m, j] := b[i−1+m, j−1]−a[i−1+m, j−1]                (2)
                                +a[i−1+m, j−1+m];
    sum[i, j] := sum[i−1, j]−b[i−1, j]+b[i−1+m, j]
```

Inefficiency in the straightforward program (1) is caused by aggregate array computations (in the inner two loops) that overlap as array subscripts are updated (by the outer two loops). We call this *overlapping aggregate array redundancy*. Figure 1 illustrates this: the horizontally filled square contributes to the aggregate computation $sum[i-1, j]$, and the vertically filled square contributes to the aggregate computation $sum[i, j]$. The overlap of these two squares reflects the redundancy between the two computations. The optimization for eliminating it requires explicitly capturing aggregate array computations in a loop body and, as the loop variable is updated, updating the results of the aggregate computations incrementally rather than computing them from scratch. In the optimized program (2), $sum[i, j]$ is computed efficiently by updating $sum[i-1, j]$. Finding such incrementality is the subject of this paper, and it is beyond the scope of previous compiler optimizations.

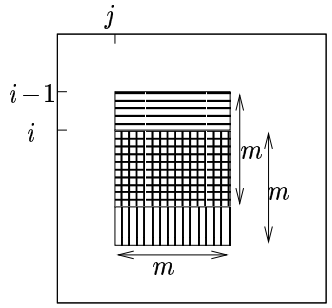

Figure 1: The overlap of the two squares shows the redundancy in the straightforward program (1) for the local summation problem.

There are many applications where programs can be written easily and clearly using arrays but with a great deal of overlapping aggregate array redundancy. These include problems in image processing, computational geometry, computer graphics, multimedia, matrix computation, list processing, graph algorithms, distributed property detection [25], serializing parallel programs [8, 17], *etc.* For example, in image processing, computing information about local neighborhoods is common [20, 32, 60, 62, 64, 65]. The local summation problem above is a simple but typical example [62, 64].

Overlapping aggregate array redundancy can cause severe performance degradation, especially with the increasingly large data sets that many applications are facing, yet methods for eliminating overlapping aggregate array redundancy have been lacking. Optimizations similar to incrementalization have been studied for various language features [7, 12, 28, 38, 39, 40, 43, 45, 44, 55, 63], but no systematic technique han-

*The authors gratefully acknowledge the support of the National Science Foundation under Grant CCR-9711253. Authors' address: Computer Science Department, Indiana University, Lindley Hall 215, Bloomington, IN 47405. Email: {liu,stoller}@cs.indiana.edu.

[1] For simplicity, initializations of *sum* and *b* for the array margins are omitted here. The full program is in Section 6.

dles aggregate computations on arrays. At the same time, many optimizations have been studied for arrays [1, 2, 3, 5, 24, 26, 31, 34, 42, 49, 54, 58], but none of them achieves incrementalization.

This paper presents a method and algorithms for incrementalizing aggregate array computations. The method is composed of algorithms for four major problems: (1) recognizing an aggregate array computation and how its parameters are updated, (2) transforming an aggregate array computation into an incremental computation with respect to an update, by exploiting array data dependence analysis and algebraic properties of the primitive operators, (3) determining additional values not maintained in the original program that need to be maintained for the incrementalization, using a method called cache-and-prune, and (4) forming a new loop using incrementalized array computations, with any additional information needed appropriately initialized. Both analytical and experimental results show drastic speedups that are not achievable by previous compiler optimizations.

Methods of explicit incrementalization [40], cache-and-prune [39], and use of auxiliary information [38] were first formulated for a functional language. They have been adopted for loop incrementalization of imperative programs with no arrays, generalizing traditional strength reduction [37]. This paper extends that work to handle imperative programs that use arrays. It presents a broad generalization of strength reduction from arithmetics to aggregates in common high-level languages, such as FORTRAN, rather than to aggregates in special very-high-level languages, such as SETL [22, 23, 44, 45]. Changes in hardware design have reduced the importance of strength reduction on arithmetic operations, but the ability to incrementalize aggregate computations remains essential.

Compared to work on parallelizing compilers, our method demonstrates a powerful alternative that is both orthogonal and correlated. It is orthogonal, since it speeds up computations running on a single processor, whether that processor is running alone or in parallel with others. It is correlated, since our optimization either allows subsequent parallelization to achieve greater speedup, or achieves the same or greater speedup than parallelization would while using fewer processors. In the latter case, resource requirements and communication costs are substantially reduced. Additionally, for this powerful optimization, we make use of techniques and tools for array dependence analysis [18, 19, 41, 42, 50, 51, 52] and source-to-source transformation [5, 34, 42, 49, 54] that were developed for parallelizing compilers.

Comparing the straightforward program (1) with the optimized program (24) on page 8, one can see that performing the optimizations by hand is tedious and error-prone. A central goal of programming language and compiler research is to allow programmers to write clear, straightforward programs and still have those programs execute efficiently. That is exactly the goal of this work.

This paper is organized as follows. Section 2 gives the programming language. Sections 3 describes how to identify and incrementalize aggregate array computations and form incrementalized loops. Section 4 describes how to maintain additional information to facilitate incrementalization. Section 5 presents the overall algorithm and discusses relevant issues. Section 6 gives examples with performance figures. Section 7 discusses related work.

2 Language

This paper considers an imperative language whose data types include multi-dimensional arrays. The language has variables that can be array references ($a[j_1, ..., j_m]$). To reduce clutter, we use indentation to indicate syntactic scopes and omit **begin** and **end**. We use the following program as a running example.

Example 2.1 Given an n_1-by-n_2 array a, the following code computes, for each i in the n_1-dimension, the sum of the m-by-n_2 rectangle starting at position i. It takes $O(n_1 n_2 m)$ time.

$$\begin{array}{l}\textbf{for } i := 0 \textbf{ to } n_1 - m \textbf{ do}\\ \quad s[i] := 0;\\ \quad \textbf{for } k := 0 \textbf{ to } m - 1 \textbf{ do}\\ \quad\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do}\\ \quad\quad\quad s[i] := s[i] + a[i + k, l]\end{array} \tag{3}$$

Our primary goal is to reduce the running time. Of course, maintaining additional values takes extra space. Our secondary goal is to reduce the space consumption. We use $a[.i.]$ to denote a reference of array a that contains i in a subscript. We use $t[x := e]$ to denote t with each occurrence of x replaced with e.

3 Incrementalizing aggregate array computations

We first show how to identify aggregate array computations and determine how the parameters they depend on are updated. We then show how to incrementalize aggregate array computations with respect to given updates, by exploiting properties of the functions involved. Finally, we describe how to transform a loop with aggregate array computations in the loop body into a new loop with incrementalized aggregate array computations in the loop body.

3.1 Identifying candidates

Candidates for optimizations are in nested loops, where inner loops compute accumulated quantities over array elements and outer loops update the subscripts used by the array references.

Definition 3.1 *An* aggregate array computation (AAC) *is a loop that computes an accumulated quantity over array elements. The canonical form of an AAC is*

$$\textbf{for } i := e_1 \textbf{ to } e_2 \textbf{ do } \quad v := f(v, g(a[.i.], \ldots)) \tag{4}$$

where e_1 and e_2 are expressions, f is a function of two arguments, g is a function of one or more arguments, and throughout the loop, v is a variable (which may be an array reference) that refers to a fixed memory location and is updated only with the result of f, and the values of variables (array reference or not) on which g depends remain unchanged. An initial assignment to v may be included in an AAC.

The existence of four items in the loop body identifies such a computation. First, an *accumulating variable*—v—a variable that holds the accumulated quantity. This variable may itself be an array reference whose subscripts depend only on variables defined outside the loop. Second, a *contributing array reference*—$a[.i.]$—an array reference whose subscripts depend on the loop variable. Third, a *contributing function*—g—a function that computes using the contributing array references but not the accumulating variable. Fourth, an *accumulating function*—f—a function that updates the accumulating variable using the result of the contributing function.

More generally, an AAC may contain multiple **for** clauses, multiple assignments, and multiple array references. We use the above form only to avoid clutter in the exposition of the algorithms. Extending the algorithms to allow these additional possibilities is straightforward, and these extensions are assumed in the examples.

The essential feature of an AAC A is that a set of computations are performed by the contributing function, and their results are accumulated by the accumulating function. We characterize this set by the *contributing set* $S(A)$, which describes the ranges of the subscripts of the contributing array references. For an AAC of the form (4), the contributing set is $S(A) = \{\langle a[.i.]\rangle \,|\, e_1 \le i \le e_2\}$. More generally, for an AAC A with contributing array references $a_1, \ldots, a_k$,

$$S(A) = \{\langle a_1, \ldots, a_k\rangle \,|\, R\}, \tag{5}$$

where R is the conjunction of the constraints defined by the ranges of all the loop variables of A.

Example 3.1 For the program (3), the loop on k and the loop on l each forms an AAC; we denote them as A_k and A_l, respectively. For both of them, $s[i]$ is the accumulating variable, $a[i+k, l]$ is the contributing array reference, $g(u) = u$, and $f(v, u) = (v + u)$. The contributing sets are

$$\begin{array}{ll} S(A_k) & = \{\langle a[i+k,l]\rangle \,|\, 0 \le k < m \ \wedge \ 0 \le l < n_2\} \\ S(A_l) & = \{\langle a[i+k,l]\rangle \,|\, 0 \le l < n_2\}. \end{array}$$

Definition 3.2 *A* parameter *of an AAC A is a variable used in A but defined outside A. A* subscript update operation (SUO) *for A is a redefinition of a parameter that appears in A only in the subscripts of the contributing array references. A SUO for a parameter w is denoted $\oplus_w$; in contexts where it is irrelevant to the discussion which parameter is being considered, we simply write $\oplus$.*

The heart of our approach is incrementalization of an AAC with respect to a SUO. We consider only updates to parameters that are loop variables of loops enclosing the AAC. Since we omitted specification of step size from **for** loops, it is implied that the update operation for each parameter is the operation "increment by 1". It is straightforward to deal with updates in a more general way.

Example 3.2 For A_k in program (3), variables i, m, n_2 are its parameters, and the update $\oplus_i$ is a SUO. For A_l in program (3), i, k, n_2 are its parameters, and the updates $\oplus_i$ and $\oplus_k$ are SUOs.

An AAC A and a SUO $\oplus_w$ together form a problem of incrementalization. We use $A^{\oplus_w}$ to denote A with parameter w symbolically updated by $\oplus_w$. For example, if A is of the form (4), w is the loop variable of a loop enclosing A, and the update operation is "increment by 1", then $A^{\oplus_w}$ is

$$\begin{array}{l} \textbf{for } i := e_1^{\oplus_w} \textbf{ to } e_2^{\oplus_w} \textbf{ do} \\ \quad v^{\oplus_w} := f(v, g(a[.i.], \ldots))^{\oplus_w} \end{array} \tag{6}$$

where for any t, $t^{\oplus_w}$ abbreviates $t[w := w + 1]$.

3.2 Incrementalization

Incrementalization aims to perform an AAC A incrementally as its parameters are updated by a SUO $\oplus$. The basic idea is to replace with corresponding retrievals, wherever possible, subcomputations of $A^{\oplus}$ that are also performed in A and whose values can be retrieved from the saved results of A. To consider the effect of a SUO on an AAC, we consider (i) the ranges of the subscripts of the array references on which the contributing function is computed and (ii) the algebraic properties of the accumulating function. These two aspects correspond to the following two steps.

The first step computes the differences between the contributing sets of A and $A^{\oplus}$. These differences are denoted

$$\begin{array}{l} decS(A, \oplus) = S(A) - S(A^{\oplus}) \\ incS(A, \oplus) = S(A^{\oplus}) - S(A). \end{array} \tag{7}$$

Example 3.3 For the program (3), consider incrementalization of A_k with respect to $\oplus_i$. $A_k^{\oplus_i}$ is

$$\begin{array}{l} s[i+1] := 0; \\ \textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\ \quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\ \quad\quad s[i+1] := s[i+1] + a[i+1+k, l] \end{array} \tag{8}$$

and its contributing set is

$$S(A_k^{\oplus_i}) = \{\langle a[i+1+k, l]\rangle \,|\, 0 \le k < m \wedge 0 \le l < n_2\}.$$

To compute the difference of two sets represented in the form (5), we formulate the difference as a single set of constraints and then use the methods and tools developed by Pugh *et al.* in the Omega project [50, 51, 52] to simplify the constraints.

Algorithm 3.1 (Difference of contributing sets)

Input: Two contributing sets $S_1 = \{\langle a_{11}, \ldots, a_{1k}\rangle | R_1\}$ and $S_2 = \{\langle a_{21}, \ldots, a_{2k}\rangle | R_2\}$

Output: The set difference $S_1 - S_2$.

1. Let $\bar{u}$ be a tuple of all the constrained variables in S_2. Let $\bar{u}'$ be an equal-length tuple of fresh variables. Note that $S_2 = \{\langle a_{21}[\bar{u} := \bar{u}'], \ldots, a_{2k}[\bar{u} := \bar{u}']\rangle \,|\, R_2[\bar{u} := \bar{u}']\}$.
2. Let $S = \{\langle a_{11}, \ldots, a_{1k}\rangle \,|\, R_1 \wedge \neg(\exists \bar{u}' : (a_{11} = a_{21}[\bar{u} := \bar{u}'] \wedge \cdots \wedge a_{1k} = a_{2k}[\bar{u} := \bar{u}'] \wedge R_2[\bar{u} := \bar{u}'])\}$.
3. Simplify the constraints in S using Omega [50, 51, 52].

Example 3.4 For incrementalization of A_k with respect to $\oplus_i$ in the running example, the set differences are computed as follows:

$$
\begin{aligned}
&decS(A_k, \oplus_i) = S(A_k) - S(A_k^{\oplus_i}) \\
&= \{\langle a[i+k, l]\rangle \mid (0 \le k < m \wedge 0 \le l < n_2) \wedge \neg(\exists \langle k', l'\rangle : \\
&\qquad i + k = i + 1 + k' \wedge l = l' \wedge 0 \le k' < m \wedge 0 \le l' < n_2)\} \\
&= \{\langle a[i+k, l]\rangle \mid k = 0 \wedge 0 \le l < n_2\} \\
&= \{\langle a[i, l]\rangle \mid 0 \le l < n_2\}
\end{aligned}
$$

$$
\begin{aligned}
&incS(A_k, \oplus_i) = S(A_k^{\oplus_i}) - S(A_k) \\
&= \{\langle a[i+1+k, l]\rangle \mid (0 \le k < m \wedge 0 \le l < n_2) \wedge \neg(\exists \langle k', l'\rangle : \\
&\qquad i + 1 + k = i + k' \wedge l = l' \wedge 0 \le k' < m \wedge 0 \le l' < n_2)\} \\
&= \{\langle a[i+1+k, l]\rangle \mid k = m - 1 \wedge 0 \le l < n_2\} \\
&= \{\langle a[i+m, l]\rangle \mid 0 \le l < n_2\}
\end{aligned}
$$

The second step in incrementalization uses the properties of the accumulating function to determine how a new AAC can be performed efficiently by updating the result of the old AAC. The goal is to update the result of A by removing the contributions from $decS(A, \oplus)$ and inserting the contributions from $incS(A, \oplus)$ in an appropriate order.

We order the elements of a constributing set $S(A)$ by the order they are used in the loops of A. The elements of $decS(A, \oplus)$ and $incS(A, \oplus)$ are ordered in the same way as those in $S(A)$ and $S(A^{\oplus})$, respectively. Let $first(S)$ and $last(S)$ denote the first and last element, respectively of S; for example, $last(decS(A_k, \oplus_i)) = \langle a[i, n_2 - 1]\rangle$ and $first(incS(A_k, \oplus_i)) = \langle a[i+m, 0]\rangle$. We say that a subset S' of S is *at the end* of S if the elements in S' are after the elements in $S - S'$; for example, $incS(A_k, \oplus_i)$ is at the end of $A_k^{\oplus_i}$, but $decS(A_k, \oplus_i)$ is not at the end of A_k.

We remove contributions from $decS(A, \oplus)$ only if it is not empty. To remove contributions, the accumulating function f must have an inverse f^{-1} with respect to its second argument, satisfying $f^{-1}(f(v, c), c) = v$. If $decS(A, \oplus)$ is not at the end of A or $incS(A, \oplus)$ is not at the end of $A^{\oplus}$, then we must also require that f is associative and commutative. If these two requirements are satisfied, $A^{\oplus}$ of the form (6) can be transformed into an incrementalized version of the form

$$
\begin{array}{l}
v^{\oplus} := v; \\
\textbf{for } i := last(decS(A, \oplus)) \textbf{ downto } first(decS(A, \oplus)) \textbf{ do} \\
\quad v^{\oplus} := f^{-1}(v^{\oplus}, g(a[.i.], \ldots)); \\
\textbf{for } i := first(incS(A, \oplus)) \textbf{ to } last(incS(A, \oplus)) \textbf{ do} \\
\quad v^{\oplus} := f(v^{\oplus}, g(a[.i.], \ldots))
\end{array} \qquad (9)
$$

where v contains the result of the previous execution of the AAC, and i is a re-use of the loop variable of the outermost loop in A. If f is not associative or not commutative, in which case $decS(A, \oplus)$ must be at the end of A, then the contributions from the elements of $decS(A, \oplus)$ must be removed from v in the opposite order from which they were added; this is why **downto** is used in (9).

The structure of the code in (9) is schematic; the exact loop structure needed to iterate over $decS(A, \oplus)$ and $incS(A, \oplus)$ depends on the form of the simplified constraints in them, which depends on the ranges of the loops in A and on subscripts in the contributing array references. If the loop bounds and array subscripts are affine functions of the loop variables of A, then the constraints can be simplified into a set of inequalities giving upper and lower bounds on these variables; using Omega's code generation facility, these inequalities are easily converted into loops that iterate over $decS(A, \oplus)$ and $incS(A, \oplus)$. When the size of the set $decS(A, \oplus)$ or $incS(A, \oplus)$ is zero, the corresponding **for** loop can be omitted; when the size is a small constant, the corresponding **for** loop can be unrolled.

Example 3.5 For the running example, incrementalize $A_k^{\oplus_i}$ in (8). Since + has an inverse −, and since + is associative and commutative, we obtain the following incrementalized AAC:

$$
\begin{array}{l}
s[i+1] := s[i]; \\
\textbf{for } l := n_2 - 1 \textbf{ downto } 0 \textbf{ do} \\
\quad s[i+1] := s[i+1] - a[i, l]; \\
\textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad s[i+1] := s[i+1] + a[i+m, l]
\end{array} \qquad (10)
$$

The transformation from (6) to (9) is worthwhile only if the total cost of (9) is not larger than the total cost of (6). The costs of f and f^{-1} and the sizes of the contributing sets together provide good estimates of the total costs.

First, consider the asymptotic time complexity. If f^{-1} is asymptotically at least as fast as f, and $|decS(A, \oplus)| + |incS(A, \oplus)|$ is asymptotically less than $|S(A^{\oplus})|$ (these quantities are all functions of the size of the input), then the transformed program is asymptotically faster than the original program. For the running example, this condition holds, since f and f^{-1} are both constant-time, $|decS(A_k, \oplus_i)| + |incS(A_k, \oplus_i)|$ is $O(n_2)$, and $|S(A_k^{\oplus_i})|$ is $O(n_2 m)$.

Asymptotic time complexity is an important but coarse metric; statements about absolute running time are also possible. If f^{-1} is at least as fast as f in an absolute sense, and if $|decS(A, \oplus)| + |incS(A, \oplus)|$ is less than $|S(A^{\oplus})|$, then the transformed program (9) is faster than the original program (6) in an absolute sense. For the running example, this condition holds when $m > 2$. This speedup is supported by our experimental results.

3.3 Forming incrementalized loops

To use incrementalized AACs, we transform the original loop. The basic idea is to unroll the first iteration of the original loop to form the initialization and, for the remaining iterations, replace AACs with their corresponding incremental versions. While incrementalized AACs are formulated to compute values of the next iteration based on values of the current iteration, we use them to compute values of the current iteration based on values of the previous iteration. This is straightforward for any **for** loop. For the particular SUO $\oplus_w$ that is "increment by 1", we just replace w by $w - 1$.

Example 3.6 For the running example, using the incrementalized AAC in (10), we obtain the following program, which takes $O(n_1 n_2)$ time and no additional

space.

$$
\begin{array}{r}
\text{init using } A_k \text{ in (3)} \\
\text{with } i = 0 \\
\\
\textbf{for clause} \\
\\
\text{inc using (10) with } i \text{ dec } 1
\end{array}
\left\lfloor
\begin{array}{l}
s[0] := 0; \\
\textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\
\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad s[0] := s[0] + a[k,l] \\
\textbf{for } i := 1 \textbf{ to } n_1 - m \textbf{ do} \\
\quad s[i] := s[i-1]; \\
\quad \textbf{for } l := n_2 - 1 \textbf{ downto } 0 \textbf{ do} \\
\quad\quad s[i] := s[i] - a[i-1,l]; \\
\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad s[i] := s[i] + a[i-1+m,l]
\end{array}
\right. \tag{11}
$$

4 Maintaining additional information

Additional information often needs to be maintained for efficient incremental computation [38, 39]. Such information often comes from *intermediate results* computed in the middle of the original computation [39]. It may also come from *auxiliary information* that is not computed at all in the original computation [38]. The central issues are how to find, use, and maintain appropriate information.

General methods have been proposed and formulated for a functional language [38, 39]. Here we apply them to AACs, using a variant of the *cache-and-prune* method [39]. We proceed in three stages: (I) transform the code for AACs to store all intermediate results and related auxiliary information not stored already, (II) incrementalize the resulting AACs from one iteration to the next based on the stored results, and (III) prune out stored values that were not useful in the incrementalization.

4.1 Stage I: Caching results of all AACs

We consider saving and using results of all AACs. This allows greater speedup than saving and using results of primitive operations.

After every AAC, we save in fresh variables the intermediate results that are not saved already, *e.g.*, the result of A_l in (3). Since we consider AACs that are themselves performed inside loops, we must distinguish intermediate results obtained after different iterations. To this end, for each AAC A, we introduce a fresh array variable subscripted with loop variables of all the loops enclosing A, and we add an assignment immediately after A to copy the value of the accumulating variable into the corresponding element of the fresh array. For the example A_l, we introduce a fresh array s_1 and add $s_1[i,k] = s[i]$ after A_l.

A related class of auxiliary information can be obtained to facilitate incrementalization if the accumulating function f is associative and has a zero element 0 (*i.e.*, $f(v,0) = f(0,v) = v$). In this case, we save in a fresh array values of the AAC starting from 0, rather than copying intermediate results, *i.e.*, we add an assignment before the AAC to initialize the fresh array variable to 0, accumulate values computed in AAC into the fresh variable instead of the original accumulating variable, and add an assignment after the AAC to accumulate the value of the fresh variable into the original accumulating variable.

Example 4.1 For the program (3), storing such auxiliary information yields a program that, for each value of k, accumulates separately in $s_1[i,k]$ the sum computed by A_l starting from 0, and then accumulates that sum into the accumulating variable $s[i]$:

$$
\begin{array}{l}
s[i] := 0; \\
\textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\
\quad s_1[i,k] := 0; \\
\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad s_1[i,k] := s_1[i,k] + a[i+k,l]; \\
\quad s[i] := s[i] + s_1[i,k]
\end{array}
\tag{12}
$$

Essentially, this class of auxiliary information is obtained by chopping intermediate results into independent pieces based on the associativity of g. These values are not computed at all in the original program and thus are called *auxiliary information*. It helps reduce the analysis effort in later stages, since the value of an aggregate computation is directly maintained rather than being computed as the difference of two subsequent intermediate results of the larger computation.

An optimization at this stage that helps simplify analyses in later stages and reduce the space consumed by the additional information is to avoid generation of redundant subscripts for the fresh arrays. Redundancies arise when the same value is computed in multiple iterations and therefore stored in multiple entries in that array. We detect such redundancies as follows. Let $\bar{w}$ be the subscript vector of the fresh array for an AAC A, *i.e.*, $\bar{w}$ is the tuple of the loop variables of all loops enclosing A. We define two tuples $\bar{w_1}$ and $\bar{w_2}$ to be *equivalent for* A (denoted $\equiv_A$) if they lead to the same contributing set, hence to the same auxiliary information, *i.e.*, $\bar{w}_1 \equiv_A \bar{w}_2$ iff $S(A)[\bar{w} := w_1] = S(A)[\bar{w} := \bar{w}_2]$.

Example 4.2 For A_l in (12), we have

$$
\begin{array}{lll}
\langle i_1, k_1 \rangle \equiv_{A_l} \langle i_2, k_2 \rangle & \text{iff} & (\{\langle a[i_1+k_1, l] \rangle \mid 0 \le l < n_2\} = \\
& & \quad \{\langle a[i_2+k_2, l] \rangle \mid 0 \le l < n_2\}) \\
& \text{iff} & (i_1 + k_1 = i_2 + k_2).
\end{array}
\tag{13}
$$

We exploit this equivalence by observing that the simplified expression for $\equiv_A$ is always of the form

$$
\begin{array}{lll}
\bar{w}_1 \equiv_A \bar{w}_2 & \text{iff} & (e_1[\bar{w} := \bar{w}_1] = e_1[\bar{w} := \bar{w}_2]) \wedge \cdots \wedge \\
& & (e_h[\bar{w} := \bar{w}_1] = e_h[\bar{w} := \bar{w}_2])
\end{array}
\tag{14}
$$

for some expressions $e_1, \ldots, e_h$. This implies that the values of $e_1, \ldots, e_h$ together distinguish the equivalence classes of $\equiv_A$, so we can take the fresh array to be h-dimensional and use $e_1, \ldots, e_h$ as its subscripts.

Example 4.3 For A_l in (12), the equivalence $\equiv_{A_l}$ in (13) is of the form (14) with $h = 1$ and $e_1 = i + k$, so we take s_1 to be a 1-dimensional array with subscript $i + k$, obtaining the extended AAC

$$
\begin{array}{l}
s[i] := 0; \\
\textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\
\quad s_1[i+k] := 0; \\
\quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\
\quad\quad s_1[i+k] := s_1[i+k] + a[i+k,l]; \\
\quad s[i] := s[i] + s_1[i+k]
\end{array}
\tag{15}
$$

The auxiliary information now occupies $O(n_1 + m)$ space, compared to $O(n_1 m)$ space in (12).

4.2 Stage II: Incrementalization

In general, we want to perform all AACs in an iteration efficiently using stored results of the previous iteration. As a basic case, we avoid performing AACs whose values have been computed completely in the previous iteration. This can be done by keeping track of all the AACs and the variables that store their values. We incrementalize other AACs using the algorithms in Section 3.2.

Example 4.4 Incrementalize the AACs A_k and A_l in (15) with respect to $\oplus_i$. First, we avoid performing AACs that have been performed in the previous iteration. Thus, we only need to compute A_l for elements in the set difference $\{s_1[i+1+k] \mid 0 \le k \le m-1\} - \{s_1[i+k] \mid 0 \le k \le m-1\} = \{s_1[i+1+k] \mid k = m-1] = \{s_1[i+m]\}$. Then, we incrementalize A_k with respect to $\oplus_i$. We have $decS(A_k, \oplus_i) = \{s_1[i+k] \mid k = 0\} = \{s_1[i]\}$ and $incS(A_k, \oplus_i) = \{s_1[i+1+k] \mid k = m-1\} = \{s_1[i+m]\}$. These sets both have size 1, so we unroll the loops over them and obtain the incrementalized AAC

$$\begin{array}{l} s_1[i+m] := 0; \\ \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\ \quad s_1[i+m] := s_1[i+m] + a[i+m, l]; \\ s[i+1] := s[i] - s_1[i] + s_1[i+m] \end{array} \tag{16}$$

4.3 Stage III: Pruning

Some of the additional information saved in Stage I might not be useful for the incrementalization in Stage II. Stage III analyzes dependencies in the incrementalized computation and prunes out useless information and the associated computations.

The analysis starts with the uses of such information in computing the original accumulating variables and follows dependencies back to the definitions of such information. The dependencies are transitive [39] and can be used to compute all the information that is useful. Pruning then eliminates useless data and code, saving both space and time.

4.4 Forming incrementalized loops

The incrementalized loop is formed as in Section 3.3, but using the AACs that have been extended with useful additional information.

Example 4.5 From the code in (15) and its incremental version in (16) that together compute and maintain useful additional information, we obtain the optimized program below that takes $O(n_1 n_2)$ time and $O(n_1)$ additional space. Compared with the program in (11), this program eliminates a constant factor of 2 in the execution time and thus is twice as fast. Our experimental results also support this speedup.

$$\begin{array}{rl} \text{init using (15) with } i = 0 & \left[\begin{array}{l} s[0] := 0; \\ \textbf{for } k := 0 \textbf{ to } m-1 \textbf{ do} \\ \quad s_1[k] := 0; \\ \quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\ \quad\quad s_1[k] := s_1[k] + a[k, l]; \\ \quad s[0] := s[0] + s_1[k]; \end{array}\right. \\ \textbf{for clause} & \textbf{for } i := 1 \textbf{ to } n_1 - m \textbf{ do} \\ \text{inc using (16) with } i \text{ dec } 1 & \left[\begin{array}{l} \quad s_1[i-1+m] := 0; \\ \quad \textbf{for } l := 0 \textbf{ to } n_2 - 1 \textbf{ do} \\ \quad\quad s_1[i-1+m] := s_1[i-1+m] + a[i-1+m, l]; \\ \quad s[i] := s[i-1] - s_1[i-1] + s_1[i-1+m] \end{array}\right. \end{array} \tag{17}$$

5 The optimization algorithm

The overall optimization algorithm aims to incrementalize aggregate array computations in every loop of a program.

Algorithm 5.1 (Eliminating overlapping aggregate array redundancies)

Consider nested loops from inner to outer and, for each loop L encountered, perform Steps 1-5.

1. Let w denote the loop variable of L. Identify all loops in the loop body of L that are AACs and for which $\oplus_w$ is a SUO, as defined in Section 3.1.
2. Extend these AACs to save all appropriate additional information in variables, if not saved already, as described in Section 4.1.
3. Incrementalize these AACs with respect to $\oplus_w$, as described in Section 4.2. If any of these AACs are nested, consider them from inner to outer.
4. Prune additional information that is not useful for the incrementalization.
5. If incrementalization is performed, then form incrementalized loops using incrementalized AACs, as described in Section 4.4.

This algorithm is expensive but automatic. A number of optimizations are possible. For example, Step 1 needs to consider only AACs whose contributing array references depend on the current loop variable. For another example, since we consider nested loops from inner to outer, Step 2 only needs to consider saving results of AACs outside of loops considered already.

Our optimization can achieve drastic program speedup. The additional space consumption may look worrisome, since it may affect cache performance for large data sets. While our optimization eliminates redundant computation, it also eliminates redundant data access, so it generally preserves or increases cache locality. In general, however, more rigorous study is needed for analysis of space consumption as well as various trade-offs.

Our optimization uses an exact inverse f^{-1} when $decS(A, \oplus) \neq \emptyset$. If the computations are floating-point, this might be computed only approximately, and thus the optimized program might produce less accurate results than the original program. Such inaccuracies also arise in other optimizations that reorganize loops. We do not expect this problem to be worse for our optimization than others, though experiments are needed to verify this.

6 Examples and performance results

The following examples and performance results show the speedups obtained by our method. The figures are obtained from running the original and optimized programs, coded in FORTRAN, on a dedicated SPARCstation 4. The programs were compiled using Sun Microsystems' f77 compiler, with optimization flags -O4 and -fast.

6.1 Partial sum

Partial sum is a simple but interesting and illustrative example. Given an array $a[1..n]$ of numbers, for each index i (line [1]), compute the sum of elements 1 to i (lines [2] to [4]). The straightforward program (18)

takes $O(n^2)$ time.

```
[1]  for i := 1 to n do
[2]    s[i] := 0;
[3]    for j := 1 to i do
[4]      s[i] := s[i] + a[j]
```
(18)

It can be optimized using our algorithm. First, consider the inner loop. Its loop body does not contain any AACs. Now, consider the outer loop. Step 1. Its loop body contains an AAC A_j, where $s[i]$ is the accumulating variable, and its loop increment is a SUO $\oplus_i$. Step 2. No additional values need to be saved. Step 3. $decS(A, \oplus_i) = \emptyset$ and $incS(A, \oplus_i) = \{\langle a[i+1]\rangle\}$. Thus, the computation of $s[i+1]$ is incrementalized by accumulating to the value of $s[i]$ the only contribution $a[i+1]$. We obtain $s[i+1] := s[i] + a[i+1]$. Step 4. Pruning leaves the code unchanged. Step 5. Initializing $s[1]$ to $a[1]$ and forming the rest of the loop for $i = 2..n$, we obtain the program (19).

```
s[1] := a[1];
for i := 2 to n do
  s[i] := s[i-1] + a[i]
```
(19)

This program takes only $O(n)$ time. Running times for programs (18) and (19) are plotted in Figure 2; the rate of increase of the running time of the optimized program is extremely small.

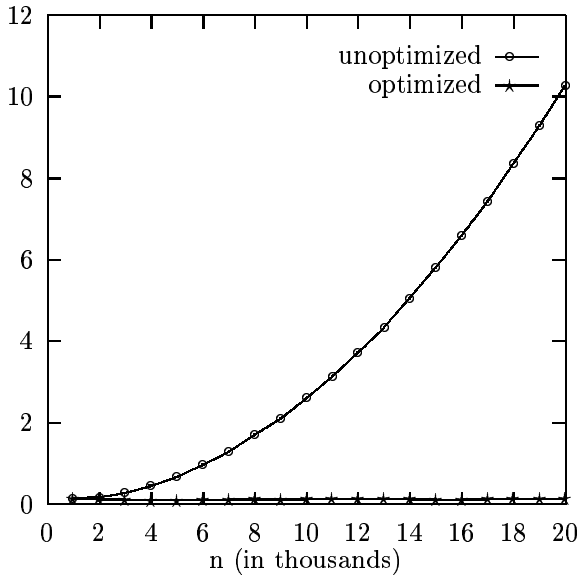

Figure 2: Running time (in seconds) for partial sum problem.

6.2 Local neighborhood problems

This problem was introduced in Section 1. We show that applying our optimization algorithm to the straightforward program (1) yields the efficient program (2) with appropriate initializations of the array margins.

First, consider the innermost loop L_l on l. There is no AAC in its body.

Next, consider the loop L_k on k. Its loop body L_l is an AAC A_l, and its loop increment is a SUO $\oplus_k$. Array analysis yields $decS(A_l, \oplus_k) = S(A_l)$ and $incS(A_l, \oplus_k) = S(A_l^{\oplus_k})$, so incrementalization is not worthwhile. The algorithm leaves the code unchanged.

Next, consider the loop L_j on j. Step 1. Its loop body contains two AACs, A_l and A_k, and its loop increment is a SUO $\oplus_j$. Step 2. Since the accumulating function + is associative, saving the values of A_l in an array b yields a new loop body

```
sum[i,j] := 0;
for k := 0 to m-1 do
  b[i+k,j] := 0;
  for l := 0 to m-1 do
    b[i+k,j] := b[i+k,j] + a[i+k,j+l];
  sum[i,j] := sum[i,j] + b[i+k,j]
```
(20)

Step 3. Incrementalizing A_l in the body of the loop on k with respect to $\oplus_j$, we have $decS(A_l, \oplus_j) = \{\langle a[i+k, j+l]\rangle \mid l = 0\} = \{\langle a[i+k, j]\rangle\}$ and $incS(A_l, \oplus_j) = \{\langle a[i+k, j+1+l]\rangle \mid l = m-1\} = \{\langle a[i+k, j+m]\rangle\}$. Incrementalizing A_k with respect to $\oplus_j$, we have $decS(A_k, \oplus_j) = S(A_k)$ and $incS(A_k, \oplus_j) = S(A_k^{\oplus_j})$, so incrementalization is not worthwhile. We obtain

```
sum[i,j+1] := 0;
for k := 0 to m-1 do
  b[i+k,j+1] := b[i+k,j] - a[i+k,j] + a[i+k,j+m];
  sum[i,j+1] := sum[i,j+1] + b[i+k,j+1]
```
(21)

Step 4. Pruning (21) leaves the code unchanged. Step 5. Initialize using (20) with $j = 0$ and form loop for $j = 1..n-m$ using (21) as loop body. We obtain

```
init using (20) with j = 0:
  sum[i,0] := 0;
  for k := 0 to m-1 do
    b[i+k,0] := 0;
    for l := 0 to m-1 do
      b[i+k,0] := b[i+k,0] + a[i+k,l];
    sum[i,0] := sum[i,0] + b[i+k,0];
for clause:
  for j := 1 to n-m do
    inc using (21) with j dec 1:
      sum[i,j] := 0;
      for k := 0 to m-1 do
        b[i+k,j] := b[i+k,j-1] - a[i+k,j-1]
                              + a[i+k,j-1+m];
        sum[i,j] := sum[i,j] + b[i+k,j]
```
(22)

Finally, consider the outermost loop L_i. Step 1. Its loop body is now (22); the first half contains AACs A_k and A_l, and the second half contains, in the body of the loop on j, incrementalized AAC of $b[i+k, j]$ and AAC $A_{k'}$ of $sum[i, j]$ by the loop over k. Its loop increment is a SUO $\oplus_i$. Step 2. No additional values need to be saved. Step 3. Incrementalize AACs in (22) with respect to $\oplus_i$. In the first half, only A_l in $\{b[i+1+k, 0] \mid k = m-1\} = \{b[i+m, 0]\}$ needs to be computed; also, $decS(A_k, \oplus_i) = \{\langle b[i+k, 0]\rangle \mid k = 0\} = \{\langle b[i, 0]\rangle\}$ and $incS(A_k, \oplus_i) = \{\langle b[i+1+k, 0]\rangle \mid k = m-1\} = \{\langle b[i+m, 0]\rangle\}$. In the second half, in the body of the loop on j, only $A_{k'}$ in $\{b[i+1+k, j] \mid k = m-1\} = \{b[i+m, j]\}$ needs to be computed; also, $decS(A_j, \oplus_i) = \{\langle b[i+k, j]\rangle \mid k = 0\} = \{\langle b[i, j]\rangle\}$ and $incS(A_j, \oplus_i) = \{\langle b[i+1+k, j]\rangle \mid k = m-1\} = \{\langle b[i+$

$m, j]\rangle\}$. We obtain

```
b[i+m,0] := 0;
for l := 0 to m−1 do
  b[i+m,0] := b[i+m,0] + a[i+m,l];
sum[i+1,0] := sum[i,0] − b[i,0] + b[i+m,0];
for j := 1 to n−m do
  b[i+m,j] := b[i+m,j−1]−a[i+m,j−1]
                        +a[i+m,j−1+m];
  sum[i+1,j] := sum[i,j] − b[i,j] + b[i+m,j]
```
(23)

Step 4. Pruning (23) leaves the code unchanged. Step 5. Initialize using (22) with $i = 0$ and form loop for $i = 1..n-m$ using (23) as loop body. We obtain the optimized code in (24). Starred lines correspond to the code in (2); other lines perform array margin initialization.

```
init using (22) with i = 0:
  sum[0,0] := 0;
  for k := 0 to m−1 do
    b[k,0] := 0;
    for l := 0 to m−1 do
      b[k,0] := b[k,0] + a[k,l];
    sum[0,0] := sum[0,0] + b[k,0];
  for j := 1 to n−m do
    sum[0,j] := 0;
    for k := 0 to m−1 do
      b[k,j] := b[k,j−1]−a[k,j−1] + a[k,j−1+m];
      sum[0,j] := sum[0,j] + b[k,j];
for clause:
  for i := 1 to n−m do                                  ⋆
inc using (23) with i dec 1:
    b[i−1+m,0] := 0;
    for l := 0 to m−1 do
      b[i−1+m,0] := b[i−1+m,0] + a[i−1+m,l];
    sum[i,0] := sum[i−1,0]−b[i−1,0]+b[i−1+m,0];
    for j := 1 to n−m do                                ⋆
      b[i−1+m,j] := b[i−1+m,j−1]−a[i−1+m,j−1]
                          +a[i−1+m,j−1+m];              ⋆
      sum[i,j]:= sum[i−1,j]−b[i−1,j]+b[i−1+m,j]         ⋆
```
(24)

The cost analysis in both incrementalization steps (21) and (23) ensures that the transformations are worthwhile when $m > 2$. In the resulting code (24), only four $\pm$ operations are performed for each pixel, independent of m. Thus, the optimized code takes $O(n^2)$ time. Running times for programs (1) and (24) are shown in Figure 3. As expected, the running time for the optimized program is approximately independent of m.

7 Related work and conclusion

The basic idea of incrementalization is at least as old as Babbage's difference machine [27]. *Strength reduction* is the first realization of this idea in optimizing compilers [12, 28, 56]. The idea is to compute certain multiplications in loops incrementally using additions. Our work extends traditional strength reduction from arithmetic operations to aggregate array computations.

Finite differencing generalizes strength reduction to handle set operations in very-high-level languages like SETL [15, 22, 23, 43, 45]. The idea is to replace aggregate operations on sets with incremental operations. Similar ideas are also used in the language INC [63], which allows programs to be written using operations on bags, rather than sets. Our work exploits the semantics underlying finite differencing to handle aggregate computations on arrays, which are more common in high-level languages and are more convenient for expressing many application problems.

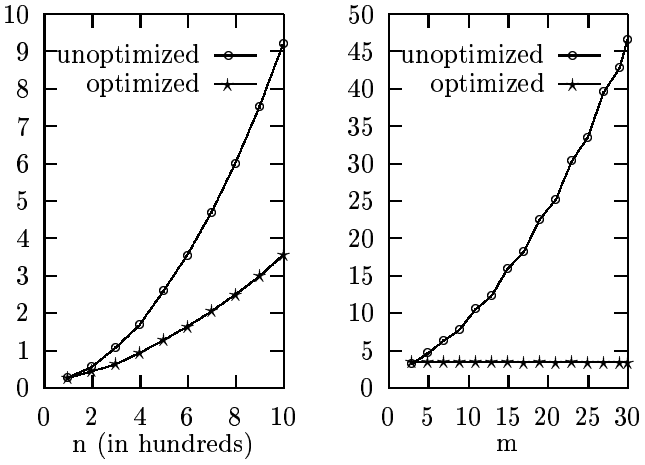

Figure 3: Running time (in seconds) for the local summation problem. For the graph on the left, $m = 10$. For the graph on the right, $n = 1000$.

APL compilers optimize aggregate array operations by performing computations in a piece-wise and on-demand fashion, avoiding unnecessary storage of large intermediate results in sequences of operations [24, 31, 61]. The same basic idea underlies techniques such as *fusion* [2, 3, 11, 26, 58], *deforestation* [57], and transformation of *series expressions* [59]. These optimizations do not aim to compute each piece of the aggregate operations incrementally using previous pieces and thus cannot produce as much speedup as our method can.

Specialization techniques, such as *data specialization* [35], *run-time specialization and code generation* [13, 36], and *dynamic compilation and code generation* [4, 16], have been used in program optimizations and achieved certain large speedups. These optimizations allow subcomputations repeated on fixed dynamic values to be computed once and reused in loops or recursions. Our optimization exploits subcomputations whose values can be efficiently updated, in addition to directly reused, from one iteration to the next. Thus, it allows far more speedup.

General *program transformations* [10, 33] can be used for optimization, as demonstrated in projects like CIP [6, 9, 46]. In contrast to such manual or semi-automatic approaches, our optimization of aggregate array computations can be automated and requires no user intervention or annotations. Our method for maintaining additional information is an automatic method for *strengthening loop invariants* [14, 29, 30, 53].

Directionals are unary operations, such as LEFT and UP, invented by Fisher and Highnam [20, 21, 32], to describe computations involving small numbers of neighboring nodes on grid structures. Such computations are optimized by directional rule-based transformations and common subexpression elimination, which essentially eliminate overlapping subcomputations. Their experiments show that the Cray Fortran compiler cannot perform these optimizations. Since local computations are written using directionals but

no loops, their optimizations can potentially exploit more associativities than ours. Their work has also some limitations. They optimize only computations involving a small number of neighbors, not other overlapping computations, such as those in the partial sum example. Also, programs must be written using directionals to take advantage of their optimizations; this is inconvenient when more than a few neighbors are involved. Finally, they do not give general methods for handling grid margins.

Loop reordering [5, 34, 42, 49, 54], *pipelining* [1], and *array data dependence analysis* [18, 19, 41, 42, 50, 51, 52] have been studied extensively for optimizing—in particular, parallelizing—array computations. While they aim to determine dependencies among uses of array elements, we further seek to determine exactly how subcomputations differ from one another. We reduce our analysis problem to symbolic simplification of constraints on loop variables and array subscripts, so methods and techniques developed for such simplifications for parallelizing compilers can be exploited. In particular, we have used tools developed by Pugh's group [50, 51, 52]. Interestingly, ideas of incrementalization are used for optimizations in serializing parallel programs [8, 17].

In conclusion, this work describes a method and algorithms that allow more drastic optimizations of aggregate array computations than previous methods. Besides achieving optimizations not previously possible, our techniques fall out of one general approach, rather than simply being yet another new but *ad hoc* method. Future work includes implementation, faster optimization algorithms, and more general classes of aggregate computations.

Applying incrementalization to loop optimization on arrays will enable us to study important issues of cost, performance, and trade-offs of time, space, and locality more explicitly, precisely, and empirically than before. This is due to the large body of previously studied and implemented techniques and the availability of benchmarks for optimizing and parallelizing compilers.

References

[1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):366–432, Sept. 1995.

[2] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, N.J., 1971.

[3] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, 1983.

[4] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In PLDI 1996 [47], pages 149–159.

[5] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, Aug. 1990.

[6] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng.*, 15(2):165–180, Feb. 1989.

[7] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. and Syst.*, 6(4):487–504, Oct. 1984.

[8] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 145–156. ACM, New York, June 1991.

[9] M. Broy. Algebraic methods for program construction: The project CIP. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 199–222. Springer-Verlag, Berlin, 1984.

[10] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

[11] W.-N. Chin. Safe fusion of functional expressions. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 11–20. ACM, New York, June 1992.

[12] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Commun. ACM*, 20(11):850–856, Nov. 1977.

[13] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL 1996 [48].

[14] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[15] J. Earley. High level iterators and a method for automatically designing data structure representation. *J. Comput. Lang.*, 1:321–342, 1976.

[16] D. R. Engler. VCODE: A retrogetable, extensible, very fast dynamic code generation system. In PLDI 1996 [47], pages 160–170.

[17] M. D. Ernst. Serializing parallel programs by removing redundant computation. Master's thesis, MIT, August 1992, Revised August 1994.

[18] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, Sept. 1988.

[19] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), Feb. 1991.

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

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

[22] A. C. Fong. Inductively computable constructs in very high level languages. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 21–28. ACM, New York, Jan. 1979.

[23] A. C. Fong and J. D. Ullman. Inductive variables in very high level languages. In *Conference Record of the 3rd Annual ACM Symposium on Principles of Programming Languages*, pages 104–112. ACM, New York, Jan. 1976.

[24] O. I. Franksen. *Mr. Babbage's Secret : The Tale of a Cypher and APL.* Prentice-Hall, Englewood Cliffs, N.J., 1985.

[25] V. K. Garg and J. R. Mitchell. An efficient algorithm for detecting conjunctions of general global predicates. Technical Report TR-PDS-1996-005, University of Texas at Austin, 1996.

[26] A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 53–62. ACM, New York, Aug. 1984.

[27] H. H. Goldstine. Charles Babbage and his analytical engine. In *The Computer from Pascal to von Neumann*, chapter 2, pages 10–26. Princeton University Press, Princeton, New Jersey, 1972.

[28] A. A. Grau, U. Hill, and H. Langmaac. *Translation of ALGOL 60*, volume 1 of *Handbook for automatic computation*. Springer, Berlin, 1967.

[29] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[30] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Sci. Comput. Program.*, 2:207–214, 1984.

[31] L. Guibas and K. Wyatt. Compilation and delayed evaluation in APL. In *Conference Record of the 5th Annual ACM Symposium on POPL*, pages 1–8. ACM, New York, Jan. 1978.

[32] P. T. Highnam. *Systems and Programming Issues in the Design and Use of a SIMD Linear Array for Image Processing*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Apr. 1991.

[33] S. Katz. Program optimization using invariants. *IEEE Trans. Softw. Eng.*, SE-4(5):378–389, Nov. 1978.

[34] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In *Proceedings of the 7th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, Ithaca, New York, Aug. 1994.

[35] T. B. Knoblock and E. Ruf. Data specialization. In PLDI 1996 [47].

[36] M. Leone and P. Lee. Optimizing ML with run-time code generation. In PLDI 1996 [47], pages 137–148.

[37] Y. A. Liu. Principled strength reduction. In *Proceedings of the IFIP TC2 Working Conference on Algorithmic Languages and Calculi*. Chapman & Hall, London, U.K., Feb. 1997.

[38] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Discovering auxiliary information for incremental computation. In POPL 1996 [48], pages 157–170.

[39] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. and Syst.*, 20(2), March 1998.

[40] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.

[41] V. Maslov. Lazy array data-flow dependence analysis. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 1994.

[42] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 1993.

[43] R. Paige. Transformational programming—Applications to algorithms and systems. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 73–87. ACM, New York, Jan. 1983.

[44] R. Paige. Symbolic finite differencing—Part I. In *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 36–56. Springer-Verlag, Berlin, May 1990.

[45] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. and Syst.*, 4(3):402–454, July 1982.

[46] H. A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.

[47] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. ACM, New York, May 1996.

[48] *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 1996.

[49] W. Pugh. Uniform techniques for loop optimization. In *International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[50] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM*, 31(8), Aug. 1992.

[51] W. Pugh and D. Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. Technical Report CS-TR-3191, Department of Computer Science, University of Maryland, College Park, Maryland, Dec. 1992. An earlier version of this paper appeared at the ACM SIGPLAN '92 Conference on PLDI.

[52] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, Portland, Oregon, Aug. 1993.

[53] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[54] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187. ACM, New York, June 1992.

[55] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, Sept. 1990.

[56] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *Proceedings of the 4th International Joint Conference on TAPSOFT*, volume 494 of *Lecture Notes in Computer Science*, pages 394–415. Springer-Verlag, Berlin, 1991.

[57] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer-Verlag, Berlin, Mar. 1988.

[58] J. Warren. A hierarchical basis for reordering transformations. In *Conference Record of the 11th Annual ACM Symposium on POPL*, pages 272–282. ACM, New York, Jan. 1984.

[59] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Program. Lang. and Syst.*, 13(1):52–98, Jan. 1991.

[60] J. A. Webb. Steps towards architecture-independent image processing. *IEEE Computer*, 25(2):21–31, Feb. 1992.

[61] B. Wegbreit. Goal-directed program transformation. *IEEE Trans. Softw. Eng.*, SE-2(2):69–80, June 1976.

[62] W. M. Wells, III. Efficient synthesis of Gaussian filters by cascaded uniform filters. *IEEE Trans. Patt. Anal. Mach. Intell.*, 8(2):234–239, Mar. 1986.

[63] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Trans. Program. Lang. and Syst.*, 13(2):211–236, Apr. 1991.

[64] R. Zabih. *Individuating Unknown Objects by Combining Motion and Stereo*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1994.

[65] R. Zabih and J. Woodfill. Non-parametric local transforms for computing visual correspondence. In J.-O. Eklundh, editor, *Proceedings of the 3rd European Conference on Computer Vision*, volume 801 of *Lecture Notes in Computer Science*, pages 151–158. Springer-Verlag, 1994.