Optimizing Aggregate Array Computations in Loops

YANHONG A. LIU and SCOTT D. STOLLER
State University of New York at Stony Brook
NING LI
IBM Almaden
and
TOM ROTHAMEL
State University of New York at Stony Brook

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 computations. This article presents a method and algorithms that eliminate such overlapping aggregate array redundancies and shows analytical and experimental performance improvements. The method is based on incrementalization, that is, updating the values of aggregate array computations from iteration to iteration rather than computing them from scratch in each iteration. This involves maintaining additional values 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. For aggregate array computations that have significant redundancy, incrementalization produces drastic speedup compared to previous optimizations; when there is little redundancy, the benefit might be offset by cache effects and other factors. Previous methods for loop optimizations of arrays do not perform incrementalization, and previous techniques for loop incrementalization do not handle arrays.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.4 [**Programming Languages**]: Processes—*Code generation,optimization*; E.1 [**Data Structures**]: *Arrays*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*

This work was supported in part by the National Science Foundation (NSF) under grant CCR-9711253 and by the Office of Naval Research (ONR) under grants N00014-99-1-0132, N00014-99-1-0358, and N00014-01-1-0109.

This article is a revised and extended version of a paper that appeared in *Proceedings of the IEEE 1998 International Conference on Computer Languages* (Chicago, Illinois, May), IEEE Computer Society Press, Los Alamitos, Calif., 1998.

Authors' addresses: Y. A. Liu, S. D. Stoller, and T. Rothamel, Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794; email: {liu,stoller}@cs.sunysb.edu; tom@rothamel.org; N. Li, IBM Almaden, 650 Harry Road, San Jose, CA 95120; email: ningli@us.ibm.com.

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Array dependence analysis, caching intermediate results, incremental computation, loop optimization, program transformation

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 code (1) takes $O(n^2m^2)$ time, whereas the optimized code (2) takes $O(n^2)$ time.

```
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
    $b[i-1+m, j] := b[i-1+m, j-1] - a[i-1+m, j-1] + a[i-1+m, j-1+m]$;     (2)
    $sum[i, j] := sum[i-1, j] - b[i-1, j] + b[i-1+m, j]$;
```

For simplicity, the optimized code (2) does not include initializations of *sum* and b for array margins; the complete code (27) appears in Section 6.

Inefficiency in the straightforward code (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 code (2), $sum[i, j]$ is computed efficiently by updating $sum[i–1, j]$. Finding such incrementality is the subject of this article. It is beyond the scope of previous compiler optimizations.

There are many applications in which 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 [Garg and Mitchell 1996], serializing parallel programs [Bromley et al. 1991; Ernst 1992], etc. For example, in image processing, computing information about local neighborhoods is common [Fisher and Highnam 1988; Highnam 1991; Webb 1992; Wells III 1986; Zabih 1994; Zabih and Woodfill 1994]. The local summation problem above is a simple but typical example [Wells III 1986; Zabih 1994]; it arises, for example,

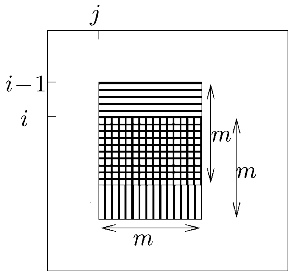

Fig. 1. The overlap of the two squares shows the redundancy in the straightforward code (1) for the local summation problem.

for blurring images, where local neighborhoods are most often 3-by-3, but 5-by-5, 7-by-7, 11-by-11, and larger neighborhoods are also used.

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 (e.g., Bird [1984], Cocke and Kennedy [1977], Grau et al. [1967], Liu et al. [1998, 2001], Liu and Teitelbaum [1995], Paige [1990], Paige and Koenig [1982], Smith [1990], and Yellin and Strom [1991]), but no systematic technique handles aggregate computations on arrays. At the same time, many optimizations have been studied for arrays (e.g., Ahmed et al. [2001], Allan et al. [1995], Allen and Cocke [1971], Allen [1983], Banerjee [1990], Faber et al. [2001], Franksen [1985], Goldberg and Paige [1984], Guibas and Wyatt [1978], Kelly and Pugh [1994], Sarkar and Thekkath [1992], and Warren [1984]), but none of them achieves incrementalization.

This article presents a method and algorithms for incrementalizing aggregate array computations. The method is composed of algorithms for four major problems: 1) identifying 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 values needed appropriately initialized. Both analytical and experimental results show drastic speedups that are not achievable by previous compiler optimizations.

Methods of explicit incrementalization [Liu and Teitelbaum 1995], cache-and-prune [Liu et al. 1998], and use of auxiliary information [Liu et al. 2001] were first formulated for a functional language. They have been adopted for optimizing recursive functions [Liu and Stoller 2002, 2003] and for optimizing loops with no arrays [Liu 1997]. The latter generalizes traditional strength reduction [Cocke and Kennedy 1977; Grau et al. 1967]. This article further extends it to handle programs that use arrays. It presents a broad generalization of strength reduction from arithmetic operations to aggregate computations in common high-level languages, such as Fortran, rather than to aggregates in special very-high-level languages, such as SETL [Fong 1979; Fong and Ullman 1976; Paige 1990; Paige and Koenig 1982]. 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 [Feautrier 1988, 1991; Maslov 1994; Pugh 1992; Pugh and Wonnacott 1998] and source-to-source transformation [Banerjee 1990; Kelly and Pugh 1994; Pugh 1991; Sarkar and Thekkath 1992] that were developed for parallelizing compilers.

Comparing the straightforward code (1) with the optimized code (27) in Section 6, one can see that performing the optimizations by hand is tedious and error-prone. One might not find as many existing programs that contain straightforward code like (1), because for practical applications, currently such code must be optimized by hand to avoid severe performance penalties. However, straightforward code like (1) is preferred over complicated code like (27) for all good programming principles except performance. 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.

The rest of this article is organized as follows. Section 2 covers preliminaries. Section 3 describes how to identify and incrementalize aggregate array computations and form incrementalized loops. Section 4 describes how to maintain additional values to facilitate incrementalization. Section 5 presents the overall optimization algorithm and discusses the cache behavior, space consumption, code size, and floating-point precision of the optimized programs. Section 6 describes examples and experimental results. Section 7 compares with related work and concludes.

2. PRELIMINARIES

This article considers an imperative language whose data types include multidimensional arrays. The language has variables that can be array references ($a[e_1, \ldots, e_n]$, where a is an array variable, and subscripts $e_1, \ldots, e_n$ are integer-valued expressions). To reduce clutter, we use indentation to indicate syntactic scopes and omit **begin** and **end**. We use the following code as a running example.

Example 2.1. Given an n_1-by-n_2 array a, the following code computes, for each row 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.

```
for $i := 0$ to $n_1 - m$ do
  $s[i] := 0;$
  for $k := 0$ to $m - 1$ do                    (3)
    for $l := 0$ to $n_2 - 1$ do
      $s[i] := s[i] + a[i + k, l];$
```

The basic idea of our method is illustrated using this example as follows. Note that the example computes, for each i from 0 to $n_1 - m$,

$$s[i] \; = \sum_{k=0}^{m-1} \; \sum_{l=0}^{n_2-1} a[i+k, l],$$

but, for each i from 1 to $n_1 - m - 1$,

$$s[i+1] \; = \sum_{k=0}^{m-1} \; \sum_{l=0}^{n_2-1} a[i+1+k, l] \; = \; s[i] - \sum_{l=0}^{n_2-1} a[i, l] + \sum_{l=0}^{n_2-1} a[i+m, l].$$

Thus, we can compute $s[i+1]$ from $s[i]$ by subtracting the sum of one row and adding the sum of another row, instead of summing m rows. We can further improve the running time by storing and using also the sum of each row, that is,

$$\sum_{l=0}^{n_2-1} a[i, l]$$

for each i from 0 to $n_1 - 1$, and computing each $s[i+1]$ by summing a single row and performing a subtraction and an addition.

Our primary goal is to reduce the running time, by storing and reusing appropriate, possibly additional, values. Because maintaining additional values takes extra space, our secondary goal is to reduce the space consumption, by storing only values that are useful for speeding up subsequent computations.

We use $a[.i.]$ to denote a reference of array a that contains variable i in a subscript. We use $e(a_1, \ldots, a_m)$ to denote an expression e that contains array references $a_1, \ldots, a_m$. 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 variables that the array subscripts depend on.

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

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

where e_1 and e_2 are expressions, f is a function of two 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) other than $a[.i.]$ on which e depends remain unchanged. An initial assignment to v may be included in an AAC; the value of v immediately before the loop is the starting value of the AAC.

The existence of four items in the loop body identifies an AAC. 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 i. Third, a *contributing expression*, e, an expression that contains the contributing array references but not the accumulating variable. Fourth, an *accumulating function*, f, a function that updates the accumulating variable using the result computed by the contributing expression.

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; the extensions are demonstrated through the examples.

The essential feature of an AAC A is that a set of computations on array elements are performed using the contributing expression, and their results are accumulated using 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 $S(A) = \{e(a[.i.]) : e_1 \le i \le e_2\}$. More generally, for an AAC A with contributing expression $e(a_1, \ldots, a_m)$,

$$S(A) = \{e(a_1, \ldots, a_m) : R\}, \tag{5}$$

where R is the conjunction of the constraints defined by the ranges of all the loop variables of A. For simplicity, we consider only AACs whose contributing array references $a_1, \ldots, a_m$ together refer to distinct tuples of array elements for distinct values of the AAC's loop variables.

Example 3.1. For the code (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 and the contributing expression, and $f(v, u) = v + u$. The contributing sets are

$$\begin{aligned} S(A_k) &= \{a[i+k, l] : 0 \le k \le m-1 \ \wedge \ 0 \le l \le n_2 - 1\}, \\ S(A_l) &= \{a[i+k, l] : 0 \le l \le n_2 - 1\}. \end{aligned}$$

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 an update to a parameter of A on which the subscripts of the contributing array references depend. 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 parameters that a SUO can update include variables in the subscript expressions of the contributing array reference as well as variables in the range expressions for the loop variable.

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 the code (3), variables i, m, n_2 are its parameters, and the update $\oplus_i$ is a SUO. For A_l in the code (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

```
for $i := e_1^{\oplus_w}$ to $e_2^{\oplus_w}$ do                              (6)
  $v^{\oplus_w} := f(v^{\oplus_w}, e(a[.i.]^{\oplus_w}))$;
```

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

Example 3.3. For A_k in the code (3) and update $\oplus_i$, $A_k^{\oplus_i}$ is simply

```
$s[i+1] := 0$;
for $k := 0$ to $m-1$ do                                  (7)
  for $l := 0$ to $n_2-1$ do
    $s[i+1] := s[i+1] + a[i+1+k, l]$;
```

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 1) the ranges of the subscripts of the array references on which the contributing function is computed and 2) 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

$$\begin{aligned} decS(A, \oplus) &= S(A) - S(A^{\oplus}), \\ incS(A, \oplus) &= S(A^{\oplus}) - S(A). \end{aligned} \tag{8}$$

Any method for computing such set differences can be used. Since this most often involves solving integer linear constraints, we first transform each of the contributing sets into a form that constrains the array subscripts explicitly based on the subscript expressions and the ranges of loop variables. Then, we use the methods and tools developed by Pugh et al. in the Omega project [Pugh 1992; Pugh and Wonnacott 1998] to simplify the constraints.

This method is embodied in the following algorithm. It first transforms both given sets by introducing fresh variables to capture subscripts of the array references and adding existential quantifiers to the loop variables. It then simplifies the constraints given by the difference of two such transformed sets. Finally, it puts the simplified subscripts into the original contributing array references, together with simplified constraints about the simplified subscripts.

Algorithm 3.1 (Difference of Contributing Sets)

```
Input: Two contributing sets $S_1 = \{e(a_{11}, \ldots, a_{1m}) : R_1\}$ and $S_2 = \{e(a_{21}, \ldots, a_{2m}) : R_2\}$.
Output: The set difference $S_1 - S_2$.
1. Transform set $S_1$ and $S_2$, each of the form $\{e(a_1, \ldots, a_m) : R\}$, into sets $T_1$ and $T_2$,
   respectively, each of a form that constrains the array subscripts explicitly.
   1.1. Let $e_{k1}, \ldots, e_{kn_k}$ be the subscripts of $a_k$ for $k = 1..m$.
   1.2. Let $v_{k1}, \ldots, v_{kn_k}$ be a set of fresh variables for $k = 1..m$.
   1.3. Let $i_1, \ldots, i_j$ be the loop variables constrained by $R$.
   1.4. Form the set
        $\{\langle v_{11}, \ldots, v_{1n_1}\rangle \ldots \langle v_{m1}, \ldots, v_{mn_m}\rangle : \exists\langle i_1, \ldots, i_j\rangle \bigwedge_{k=1..m} \bigwedge_{l=1..n_k} v_{kl} = e_{kl} \wedge R\}$   (9)
2. Simplify $T_1 - T_2$, using Omega, to $\{\langle v'_{11}, \ldots, v'_{1n_1}\rangle \ldots \langle v'_{m1}, \ldots, v'_{mn_m}\rangle : R'\}$.
3. Return $\{e(a'_1, \ldots, a'_m) : R'\}$ where $a'_k = a_k[e_{k1} := v'_{k1}, \ldots, e_{kn_k} := v'_{kn_k}]$ for $k = 1..m$.
```

Example 3.4. For the code (3), consider incrementalizing A_k with respect to $\oplus_i$, where $S(A_k)$ is as in Example 3.1, and

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

The set differences are computed as follows:

1. Transform $S(A_k)$ and $S(A_k^{\oplus_i})$ into the following sets T_1 and T_2 respectively:

$$\begin{aligned} T_1 = \{\langle v_1, v_2\rangle : \exists\langle k, l\rangle\ & v_1 = i + k \wedge v_2 = l \\ & \wedge\ 0 \le k \le m - 1 \wedge 0 \le l \le n_2 - 1\}, \\ T_2 = \{\langle v_1, v_2\rangle : \exists\langle k, l\rangle\ & v_1 = i + 1 + k \wedge v_2 = l \\ & \wedge\ 0 \le k \le m - 1 \wedge 0 \le l \le n_2 - 1\}, \end{aligned}$$

2. Simplify $T_1 - T_2$ and $T_2 - T_1$ using Omega:

$$\begin{aligned} T_1 - T_2 &= \{\langle v_1, v_2\rangle : v_1 = i \wedge 0 \le v_2 \le n_2 - 1\} \\ &= \{\langle i, l\rangle : 0 \le l \le n_2 - 1\}, \\ T_2 - T_1 &= \{\langle v_1, v_2\rangle : v_1 = i + m \wedge 0 \le v_2 \le n_2 - 1\} \\ &= \{\langle i+m, l\rangle : 0 \le l \le n_2 - 1\}. \end{aligned}$$

3. Return the differences in contributing sets:

$$\begin{aligned} \mathit{decS}(A_k, \oplus_i) &= \{a[i, l] : 0 \le l \le n_2 - 1\}, \\ \mathit{incS}(A_k, \oplus_i) &= \{a[i+m, l] : 0 \le l \le n_2 - 1\}. \end{aligned} \qquad (10)$$

The second step in incrementalizing an AAC with respect to a SUO 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 $\mathit{decS}(A, \oplus)$ and inserting the contributions from $\mathit{incS}(A, \oplus)$ in an appropriate order.

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

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

```
$v^{\oplus} := f(f^{-1}(v, v_0), v_0^{\oplus});$
for $i :=$ $\mathit{last}(\mathit{decS}(A, \oplus))$ downto $\mathit{first}(\mathit{decS}(A, \oplus))$ do
  $v^{\oplus} := f^{-1}(v^{\oplus}, e(a[.i.]));$
for $i :=$ $\mathit{first}(\mathit{incS}(A, \oplus))$ to $\mathit{last}(\mathit{incS}(A, \oplus))$ do
  $v^{\oplus} := f(v^{\oplus}, e(a[.i.]));$
```
(11)

where v contains the result of the previous execution of A, v_0 and $v_0^{\oplus}$ are the starting values of A and $A^{\oplus}$, respectively, and i is a reuse of the loop variable of the outermost loop in A. If A and $A^{\oplus}$ have the same starting value, then the initial assignment in (11) is simply $v^{\oplus} := v$. If f is not associative or not commutative, in which case $\mathit{decS}(A, \oplus)$ must be at the end of A, then the contributions from the elements of $\mathit{decS}(A, \oplus)$ must be removed from v in the opposite order from which they were added; this is why **downto** is used in (11).

The structure of the code in (11) 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 and the variable updated by $\oplus$, 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, consider incrementalizing $A_k^{\oplus_i}$ in (7), based on the set difference computed in (10). Since $+$ has an inverse $-$, and since $+$ is associative and commutative, we obtain the following incrementalized AAC:

```
$s[i+1] := s[i]$;
for $l := n_2 - 1$ downto $0$ do
  $s[i+1] := s[i+1] - a[i, l]$;                    (12)
for $l := 0$ to $n_2 - 1$ do
  $s[i+1] := s[i+1] + a[i+m, l]$;
```

The transformation from (6) to (11) is worthwhile only if the total cost of (11) 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 code (11) is asymptotically faster than the original code (6). 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 code is faster than the original code in an absolute sense. For the running example, this condition holds when $m > 2$. This speedup is supported by our experiments: the optimized program is consistently faster than the unoptimized program by a factor of $m/2$ or slightly more.

Given an AAC A and a SUO $\oplus_w$, the following algorithm is used to incrementalize A with respect to $\oplus_w$.

Algorithm 3.2 (Incrementalization of *A* with respect to $\oplus_w$)

1. Obtain $A^{\oplus_w}$ from A by symbolically updating w.
2. Compute the contributing sets $S(A)$ and $S(A^{\oplus_w})$.
3. Compute $decS(A, \oplus_w)$ and $incS(A, \oplus_w)$, using Algorithm 3.1.
4. If (1) $decS(A, \oplus_w) = \emptyset$, or f has an inverse, (2) $decS(A, \oplus_w)$ is at the end of A and $incS(A, \oplus_w)$ is at the end of $A^{\oplus}$, or f is associative and commutative, (3) A and $A^{\oplus}$ have the same starting values, or f has an inverse and is associative and commutative, and (4) the condition on complexity holds, then construct an incremental version of $A^{\oplus_w}$, of the form (11).

3.3 Forming Incrementalized Loops

To use incrementalized AACs, we transform the original loop. The basic algorithm is to unroll the first iteration of the original loop to form the initialization and, in the loop body for the remaining iterations, replace AACs with 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 requires a straightforward modification. 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 (12), we obtain the following code, which takes $O(n_1 n_2)$ time and no additional space.

```
  init using   s[0] := 0;
$A_k$ in (3)   for $k$ := 0 to $m-1$ do
$with i = 0$     for $l$ := 0 to $n_2-1$ do
                   s[0] := s[0] + $a[k,l]$;
  for-clause   for $i$ := 1 to $n_1-m$ do
                 $s[i] := s[i-1]$;                                    (13)
   inc using     for $l$ := $n_2-1$ downto 0 do
   (12) with       $s[i] := s[i] - a[i-1,l]$;
     $i$ dec 1     for $l$ := 0 to $n_2-1$ do
                   $s[i] := s[i] + a[i-1+m,l]$;
```

4. MAINTAINING ADDITIONAL VALUES

Additional values often need to be maintained for efficient incremental computation [Liu et al. 1998, 2001]. These values often come from *intermediate results* computed in the original computation [Liu et al. 1998]. They may also come from *auxiliary information* that is not computed at all in the original computation [Liu et al. 2001]. The central issues are how to find, use, and maintain appropriate values.

General methods have been proposed and formulated for a functional language [Liu et al. 1998, 2001]. Here we apply them to AACs, using a variant of the *cache-and-prune* method [Liu et al. 1998]. 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, for example, 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 of A_l in the code (3), we introduce a fresh array s_1 and add $s_1[i,k] = s[i]$ after A_l. We can see that, for each $k = 0..m-1$, $s_1[i,k]$ stores the value computed by AAC A_l in iteration k starting with the value in $s_1[i,k-1]$, or 0 if $k = 0$.

The values computed by A_l's in the code (3) can be stored in a related class of auxiliary information to better facilitate incrementalization. This class of auxiliary information can be introduced if the accumulating function f is associative and has a zero element 0 (that is, $f(v,0) = f(0,v) = v$). In this case, we save in a fresh array values of the AAC starting from 0, rather than storing accumulated intermediate results, that is, 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 code (3), storing such auxiliary information yields the following code 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 $s[i]$:

```
$s[i] := 0;$
for $k := 0$ to $m - 1$ do
  $s_1[i,k] := 0;$
  for $l := 0$ to $n_2 - 1$ do
    $s_1[i,k] := s_1[i,k] + a[i+k,l];$
  $s[i] := s[i] + s_1[i,k];$
                                                  (14)
```

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 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 values 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 array entries. We detect such redundancies as follows. Let $\bar{w}$ be the subscript vector of the fresh array for an AAC A, that is, $\bar{w}$ 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, that is, $\bar{w}_1 \equiv_A \bar{w}_2$ iff $S(A)[\bar{w} := w_1] = S(A)[\bar{w} := \bar{w}_2]$.

Example 4.2. For A_l in (14), we have

$$\begin{aligned} \langle i_1, k_1 \rangle \equiv_{A_l} \langle i_2, k_2 \rangle \quad \text{iff} \quad & (\{a[i_1{+}k_1, l] : 0 \le l \le n_2 - 1\} \\ & = \{a[i_2{+}k_2, l] : 0 \le l \le n_2 - 1\}) \\ \text{iff} \quad & (i_1 + k_1 = i_2 + k_2). \end{aligned} \tag{15}$$

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

$$\begin{aligned}\bar{w}_1 \equiv_A \bar{w}_2 \quad \text{iff} \quad & (e_1[\bar{w} := \bar{w}_1] = e_1[\bar{w} := \bar{w}_2]) \wedge \cdots \wedge \\ & (e_h[\bar{w} := \bar{w}_1] = e_h[\bar{w} := \bar{w}_2])\end{aligned} \tag{16}$$

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 (14), the equivalence $\equiv_{A_l}$ in (15) is of the form (16) 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

```
$s[i] := 0;$
for $k := 0$ to $m - 1$ do
  $s_1[i + k] := 0;$
  for $l := 0$ to $n_2 - 1$ do                         (17)
    $s_1[i + k] := s_1[i + k] + a[i + k, l];$
  $s[i] := s[i] + s_1[i + k];$
```

The auxiliary information now occupies $O(n_1 + m)$ space, compared to $O(n_1 m)$ space in (14).

4.2 Stage II: Incrementalization

In general, we want to perform all AACs in an iteration efficiently using stored results of the previous iteration. This is done by keeping track of all AACs in the loop body with the accumulating variables that store their values. As a basic case, we avoid performing AACs whose values have been computed completely in the previous iteration. We incrementalize other AACs using the algorithms in Section 3.2. AACs whose values cannot be computed incrementally using stored results of the previous iteration are computed from scratch as in the original program.

For a conceptually simpler transformation, different AACs may be separated by splitting a loop into multiple loops. This separation is not strictly necessary for the implementation.

Example 4.4. Incrementalize the code (17) with respect to $\oplus_i$. The code (17) contains AAC A_l, with accumulating variable $s_1[i + k]$ and contributing set $\{a[i+k, l] : 0 \le l \le n_2 - 1\}$, for each $k = 0..m-1$, and AAC A_k, with accumulating variable $s[i]$ and contributing set $\{s_1[i + k] : 0 \le k \le m - 1\}$. They may be separated explicitly, yielding:

```
for $k := 0$ to $m - 1$ do
  $s_1[i + k] := 0;$
  for $l := 0$ to $n_2 - 1$ do
    $s_1[i + k] := s_1[i + k] + a[i + k, l];$
$s[i] := 0;$
for $k := 0$ to $m - 1$ do
  $s[i] := s[i] + s_1[i + k];$
```

First, we avoid performing A_l for values of k that have been performed in the previous iteration. So we only need to compute A_l whose accumulating variable is in the set difference $\{s_1[i{+}1{+}k] : 0 \le k \le m{-}1\} - \{s_1[i{+}k] : 0 \le k \le m{-}1\} = \{s_1[i{+}m]\}$, as follows:

```
$s_1[i+m] := 0;$
for $l := 0$ to $n_2 - 1$ do
  $s_1[i+m] := s_1[i+m] + a[i+m, l];$
```

Then, we incrementalize A_k with respect to $\oplus_i$. We have

$$\begin{aligned} decS(A_k, \oplus_i) &= \{s_1[i+k] : 0 \le k \le m{-}1\} - \{s_1[i+1+k] : 0 \le k \le m{-}1\} \\ &= \{s_1[i]\}, \\ incS(A_k, \oplus_i) &= \{s_1[i+1+k] : 0 \le k \le m{-}1\} - \{s_1[i+k] : 0 \le k \le m{-}1\} \\ &= \{s_1[i{+}m]\}. \end{aligned}$$

Both sets have size 1, so we unroll the loops over them and obtain

$$s[i+1] := s[i] - s_1[i] + s_1[i+m];$$

Putting the two parts together, we obtain the following incrementalized code for (17) with respect to $\oplus_i$.

```
$s_1[i+m] := 0;$
for $l := 0$ to $n_2 - 1$ do
  $s_1[i+m] := s_1[i+m] + a[i+m, l];$                    (18)
$s[i+1] := s[i] - s_1[i] + s_1[i+m];$
```

4.3 Stage III: Pruning

Some of the additional values 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 values and the associated computations.

The analysis starts with the uses of such values in computing the original accumulating variables and follows dependencies back to the definitions that compute such values. The dependencies are transitive [Liu et al. 1998] and can be used to compute all the values that are useful. Pruning then eliminates useless data and code, saving space and time.

Example 4.5. In the code (18), computing $s[i+1]$ uses $s_1[i]$ and $s_1[i+m]$. Therefore, a simple dependence analysis determines that incrementally computing $s[1]$ through $s[n_1{-}m]$ uses $s_1[0]$ through $s_1[n_1{-}1]$, so no saved additional values are pruned.

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 values.

Example 4.6. From the code in (17) and its incremental version in (18) that together compute and maintain useful additional values, we obtain the following optimized code, which takes $O(n_1 n_2)$ time and $O(n_1)$ additional space.

```
       init ⎡ $s[0] := 0;$
      using ⎢ for $k := 0$ to $m - 1$ do
       (17) ⎢   $s_1[k] := 0;$
       with ⎢   for $l := 0$ to $n_2 - 1$ do
    $i = 0$ ⎢     $s_1[k] := s_1[k] + a[k, l];$
            ⎣   $s[0] := s[0] + s_1[k];$                                   (19)
 for-clause   for $i := 1$ to $n_1 - m$ do
            ⎡   $s_1[i-1+m] := 0;$
  inc using ⎢   for $l := 0$ to $n_2 - 1$ do
  (18) with ⎢     $s_1[i-1+m] := s_1[i-1+m] + a[i-1+m, l];$
  $i$ dec 1 ⎣   $s[i] := s[i-1] - s_1[i-1] + s_1[i-1+m];$
```

Compared with the code in (13), this code eliminates a constant factor of 2 in the execution time and thus is twice as fast. Our experimental results support this speedup.

5. THE OVERALL ALGORITHM AND IMPROVEMENTS

The overall optimization algorithm aims to incrementalize AACs 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 body of L that are AACs and for which $\oplus_w$ is a SUO, as defined in Section 3.1.
2. Extend the AACs identified in Step 1 to save in fresh variables appropriate additional values not saved already, as described in Section 4.1.
3. Incrementalize all AACs obtained from Step 2 with respect to $\oplus_w$, as described in Sections 4.2 and 3.2.
4. Prune saved additional values that are not useful for the incrementalization in Step 3, as described in Section 4.3.
5. If the incrementalized AACs after Step 4 are faster than the original AACs, form an incrementalized loop for L using the incrementalized AACs, as described in Sections 4.4 and 3.3.

This algorithm is restricted by the requirements on AACs and SUOs in Step 1 and the conditions for incrementalization in Step 3. Since we use Omega in constraint simplification and code generation, loop bounds and array subscripts in an AAC must be affine functions of the loop variables of the AAC and the variable updated by the SUO.

This algorithm may be expensive on deeply nested loops, but it is fully automatable, and a number of optimizations can make it more efficient. In particular, Step 1 needs to consider only AACs whose contributing array references depend on the loop variable of L. Also, since we consider nested loops from inner to outer, Step 2 only needs to consider saving results of AACs outside of loops considered already.

This algorithm can scale well since it considers only expensive AACs and updates to their parameters. These AACs and updates may be scattered through large programs but are localized in nested loops that are typically separate from each other. Separately incrementalizing these AACs with respect to the updates in large programs does not blow up the complexity.

Even if only scattered AACs are optimized, the overall speedup may be large, because typically small fragments of a program, all in loops, account for the majority of its running time [Booth 1997; Knuth 1971].

5.1 Speedup

The amount of speedup depends on many factors, including the amount of redundancy in the unoptimized code, the size of additional values maintained in the optimized code, the optimizations done by the compiler, and the machine's cache size.

As analyzed at the end of Section 3.2, incrementalization reduces the number of redundant operations by a factor of $|S(A^{\oplus})|$ over $|decS(A, \oplus)| + |incS(A, \oplus)|$, which can be asymptotic. When no additional values are maintained, the speedup of the optimized code over the unoptimized code is determined by this factor, and this is confirmed by our experiments.

However, since incremental computation accesses array elements differently and may use additional space, the running times may also be affected by cache behavior and compiler optimizations. Cache behavior of the optimized code may degrade at rare boundary cases, as discussed below in Section 5.4. Parallelizing compiler optimizations may benefit the unoptimized code more because it has less computation dependency. The optimized program is faster overall when the factor of redundancy reduction is not dominated by these other factors.

The analysis at the end of Section 3.2 currently does not consider these other factors. Nevertheless, in all our experiments with all the examples, the analysis correctly predicted performance improvements for all except one case: the image local neighborhood problem (for which relatively many additional values are maintained), at the boundary case of $m = 3$ when both relatively small cache and the compiler's highest optimization level are used.

5.2 Maintaining Additional Values and Space Consumption

Maintaining additional values for incremental computation leads to increased space consumption. Pruning helps reduce such space, but can we do better? Is there an alternative to cache-and-prune, and what are the trade-offs?

First, more sophisticated pruning might reduce the additional space consumption. For example, a more powerful analysis may determine for the code (18) that after each $s[i]$ is computed, all $s_1[0]$ through $s_1[i-1]$ will no longer be needed, and therefore an array that keeps only $s_1[i]$ through $s_1[i-1+m]$ is needed. Based on this analysis result, a more sophisticated transformation may replace the array s_1 of size n_1 with an array of size m and use a rotating index computed using the modulo operation to access the elements. Details of the analysis and transformation are a subject for further study.

Another approach is to use *selective caching* instead of cache-and-prune. The idea is to determine what additional values to cache based on what is needed for incremental computation. In particular, for AACs whose accumulating function is associative and has a zero element *zero*, we may determine what to cache by incrementalizing $A^{\oplus}$ just as in Section 3 except that, instead of (11), we form the following incrementalized version:

```
$v^{\oplus} := f(f^{-1}(v, v_0), v_0^{\oplus});$
$v_1^{\oplus} := zero;$
for $i := first(decS(A, \oplus))$ to $last(decS(A, \oplus))$ do
  $v_1^{\oplus} := f(v_1^{\oplus}, e(a[.i.]));$
$v_2^{\oplus} := zero;$                                                    (20)
for $i := first(incS(A, \oplus))$ to $last(incS(A, \oplus))$ do
  $v_2^{\oplus} := f(v_2^{\oplus}, e(a[.i.]));$
$v^{\oplus} := f(f^{-1}(v^{\oplus}, v_1^{\oplus}), v_2^{\oplus});$
```

and thus exposing additional AACs whose values (in fresh variables $v_1^{\oplus}$ and $v_2^{\oplus}$) are useful for the incremental computation. Caching and reusing the values of these additional AACs will yield optimized programs similar to those in Section 4, that is, having the same asymptotic running times.

Interestingly, alternative caching and incrementalization methods may yield programs having different space consumption or absolute running times. For example, for the image local neighborhood problem, applying our overall algorithm as in Section 6.2 yields an optimized program that performs four ±'s per pixel and caches n^2 sums of local rows; more powerful pruning could reduce the space from n^2 to mn. Suppose we use selective caching with powerful pruning, then if we optimize inner loops first as in our overall algorithm, we could obtain an optimized program that caches n sums of local *columns* for one row of the image and performs four ±'s per pixel; if we optimize *outer* loops first, we could obtain an optimized program that caches only two sums of local rows but performs six ±'s per pixel. Optimizing outer loops first while using cache-and-prune yields the same program as optimizing inner loops first since what's cached remains the same.

5.3 Code Size

In general, there may be multiple AACs and possibly other computations in the body of an original loop. Our overall algorithm handles this. A remaining problem is, if the body of the original loop contains many other computations and is large, then unrolling the first iteration of the loop to form the final optimized program may increase the code size by the size of the loop body. To remedy this, one can move computations that are independent of AACs into a separate loop. For this separate loop, there is no need to unroll the first iteration.

For expensive AACs, incrementalization may still increase the code size. It is possible sometimes to exploit special initial values and algebraic properties of the computations involved so as to fold the first iteration back into the loop body, as done in Cai and Paige [1988/89] and Liu [1997]. In general, code size might have to increase. In fact, it is exactly the optimization studied in this article that allows a programmer to write the original program that is much more succinct and clearer than the optimized program that must perform much more complex initializations and incremental computations and that he or she would otherwise need to write manually to achieve the desired efficient computation.

5.4 Cache Behavior

In the incremental version (11), variable $v^{\oplus}$ is repeatedly accessed for incremental updates. When variable v, and thus $v^{\oplus}$, is an array reference, these repeated accesses are to a memory location and result in unfavorable memory and cache behavior. This problem can easily be solved by using a temporary variable, say r, in place of $v^{\oplus}$ in (11) and assigning r to $v^{\oplus}$ at the end of (11). The resulting register-variant version allows a compiler to allocate r in a register, avoiding repeated accesses of $v^{\oplus}$ in the memory. This optimization can also be done with general scalar replacement techniques [Carr and Kennedy 1994].

Still, incremental computation accesses array elements differently and may use additional space. How does this affect cache behavior? It is easy to see that, while incrementalization eliminates redundant computation, it also eliminates redundant data accesses and reduces the total number of data accesses, even though it may use additional space. This generally preserves or improves cache locality, and the improvements are often significant. The only exception is when pruning is not powerful enough to eliminate unnecessary space consumption, or when all three of the following conditions hold simultaneously: 1) all elements of the contributing set of an AAC just fit into the cache, 2) incremental computation needs to remove contributions from elements before the contributing set, and 3) subsequent computations need to access all elements in the contributing set. These three conditions together only cause slightly worse cache behavior at rare points. These reasons explain the increases in cache misses in the sequence local average example in Section 6.3, whereas in the partial sum and image local neighborhood (except for very small m) examples in Sections 6.1 and 6.2, respectively, cache misses are significantly reduced.

Even in the rare cases where cache misses increase, the optimized programs may still be significantly faster because of the large number of operations saved. Section 6 reports running times and cache misses for both the original and optimized versions of example programs. The measured cache misses are also consistent with the results of exact static cache analysis [Chatterjee et al. 2001] applied by Erin Parker to some of our examples.

5.5 Floating-Point Precision

Our optimization uses an exact inverse f^{-1} when $decS(A, \oplus) \neq \emptyset$. If the computations involve floating-point numbers, this might be computed only approximately, and thus the optimized program might produce less accurate results than the original program. Inaccuracies also arise in other optimizations that reorganize loops, but inaccuracies caused by incrementalization are easy to remedy.

Basically, since incremental computation of AACs is performed in each iteration of a loop, inexact removal of contributions from $decS(A, \oplus)$ in each iteration causes inaccuracies to accumulate gradually over iterations. Thus, to keep inaccuracies within an allowed range, one can simply compute AACs from scratch periodically in selected iterations.

To determine how often AACs should be computed from scratch, an analysis is needed. It can be based on work on analyzing precision of floating-point computation [Higham 1996]. An easy, conservative analysis could simply sum the maximum possible inaccuracies that are accumulated over the iterations and compute the maximum number of iterations after which the inaccuracies are still within allowed errors. Recent results on analyzing precision for programs that use loops [Goubault 2001] might help do the analysis both accurately and automatically.

6. EXAMPLES AND EXPERIMENTAL RESULTS

The following examples and performance results show the speedups obtained by our optimization. We also show measurements for cache misses and floating-point errors.

The overall optimization algorithm, plus a method for selective caching, is implemented in a prototype system. The examples can be optimized automatically using the system. The implementation uses the Synthesizer Generator [Reps and Teitelbaum 1988] and Omega [Pugh 1992]. Synthesizer Generator is used to implement the interfaces and the algorithms described in this article. Omega is used for simplifying inequality constraints on loop variables and array subscripts and for generating loops from simplified constraints. The system consists of 3900 lines of code, mostly in SSL, the Synthesizer Generator language for specifying editors, and the rest in STk, a dialect of Scheme as part of the Synthesizer Generator Scripting Language, and C. The system supports source language syntax of several languages, including C and Fortran, for a subset that involves loops and arrays; this was easily implemented thanks to the Synthesizer Generator.

Graphs illustrating the behavior of the original and optimized code were created by running programs implemented in the C subset of C++. The programs were run on arrays whose elements were floating-point numbers between 0 and 1. An Athlon XP 1600+ running at 1364 MHz with a 64 KB L1 data cache and a 64 KB exclusive L2 data cache were used, on a computer equipped with 1 gigabyte of RAM, running Debian GNU/Linux with the 2.4.18 Linux kernel. The programs were compiled using g++ 3.1.1, with the optimization flag -O3, the highest documented level of optimization. All array and register variables were declared to be of 64-bit precision, but the actual calculations were performed by the FPU that uses 80-bit precision. Each program for each input size was run on 5 different arrays of data, and the means of the measurements are reported.

Where running times are given, CPU times in seconds are reported. The timers used had a natural precision of 10 milliseconds, so in some cases (the partial sum example) the optimized program was placed inside a loop that caused it to be repeated 20 times, and the total time for the loop was divided by 20; cache misses for each of the 20 times are roughly the same. Cache misses were measured using the performance monitoring counters on the Athlon to tally the number of L1 data cache misses. The performance monitoring counters were accessed using the perfctr patch applied to the Linux kernel. Cache measurements were taken on an actual system and thus may include some error introduced by context switching. No actions were taken to initialize the cache. Floating-point relative error [Higham 1996] was obtained by taking the results of the original and optimized implementations, summing the absolute difference element-wise, and dividing by the total sum of all the elements in the result array created by the unoptimized program.

Besides the examples below, we also found AACs in a number of other programs. One is a real-time application for vibration testing in mechanical engineering [Li 1997] and consists of 664 lines of Fortran code. All four of the large expensive **do**-loops are AACs similar to the sequence local average example below, and each of them can be improved to be about five times faster.

Although the examples below use only 1- or 2-dimensional arrays, the same optimization method applies just as well to 3- and higher-dimensional computations. For example, consider blurring a 3-dimensional space. Essentially, incrementalization is done one dimension at a time, from inner to outer loops in the program. Specifically, incrementalization with respect to an increment in one dimension essentially replaces each 3-dimensional local neighborhood computation with two 2-dimensional local neighborhood computations. Incrementalization of these 2-dimensional computations with respect to increments in the two remaining dimensions is similar to incrementalization for the image local neighborhood example.

6.1 Partial Sum

Partial sum is a simple illustrative example. Given an array $a[0..n-1]$ of numbers, for each index i from 0 to $n-1$ (in the outer **for**-clause), compute the sum of elements 0 to i (in the inner **for**-loop). The straightforward code (21) takes $O(n^2)$ time.

```
for $i := 0$ to $n-1$ do
  $s[i] := 0$;
  for $j := 0$ to $i$ do
    $s[i] := s[i] + a[j]$;
```
(21)

This straightforward code 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 $incS(A, \oplus_i) = \{a[i+1]\}$. 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 $s[i+1] := s[i] + a[i+1]$.

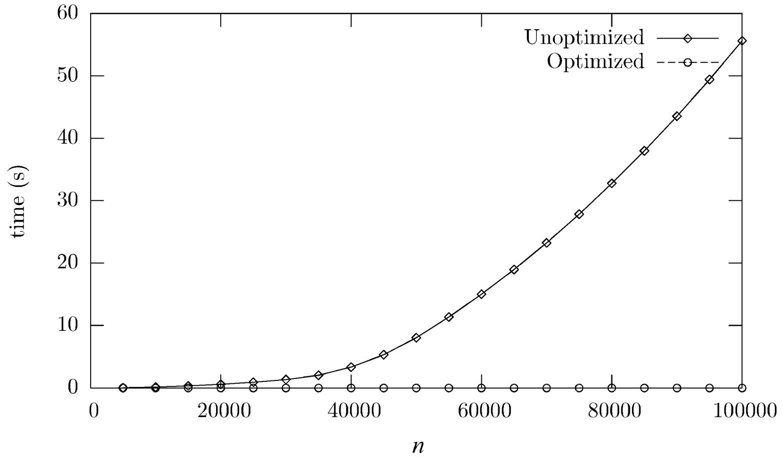

Fig. 2. Running times for partial sum. Running time for the optimized code grows roughly linearly from 0.04 to 3.36 milliseconds. Running time for the unoptimized code grows from 0.038 to 55.614 seconds.

Step 4. Pruning leaves the code unchanged.

Step 5. Initializing $s[0]$ to $a[0]$ and forming the rest of the loop for $i = 1..n{-}1$, we obtain the code (22). This optimized code takes only $O(n)$ time.

```
$s[0] := a[0];$
**for** $i := 1$ **to** $n-1$ **do**                    (22)
  $s[i] := s[i-1] + a[i];$
```

Running times for the unoptimized version (21) and the optimized version (22) are plotted in Figure 2. It shows that the rate of increase of the optimized program is very small. Figure 3 shows similar improvements in cache misses. Figure 4 shows the relative error of the results. The error is introduced by a compiler optimization to the unoptimized version that keeps the intermediate value of $s[i]$ in an 80-bit floating-point register and omits converting it to the specified 64 bits, and error accumulates as n increases. When g++ was given an option to force these conversions in the unoptimized version, no error was found.

6.2 Image Local Neighborhood

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

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

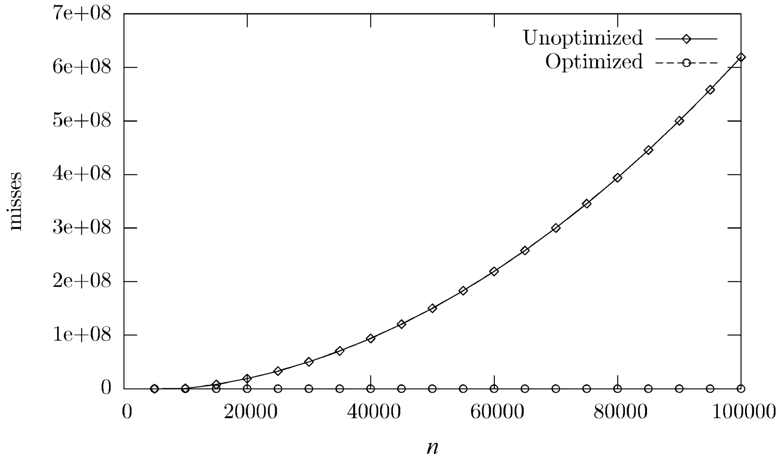

Fig. 3. Cache misses for partial sum. Cache misses for the optimized code grows roughly linearly from 664 to 25070. Cache misses for the unoptimized code grows from 1362 to 619,058,758.

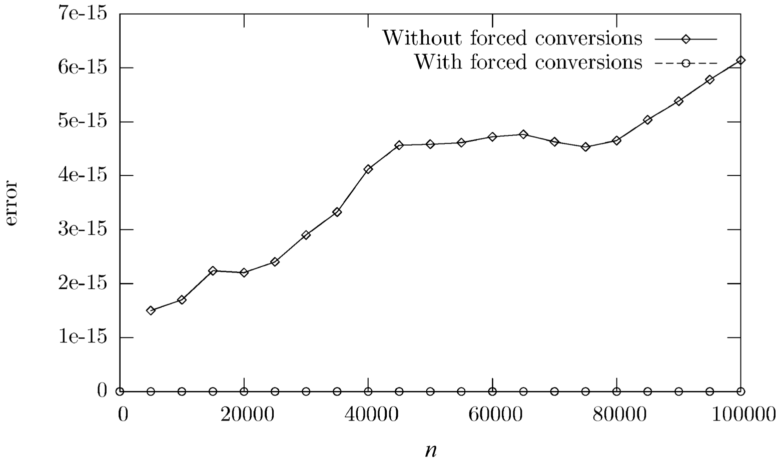

Fig. 4. Relative error for partial sum.

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
    $b[i+k, j] := b[i+k, j] + a[i+k, j+l];$
  $sum[i, j] := sum[i, j] + b[i+k, j];$
```
(23)

Step 3. Incrementalizing A_l in the body of the loop on k with respect to $\oplus_j$, we have $decS(A_l, \oplus_j) = \{a[i+k, j]\}$ and $incS(A_l, \oplus_j) = \{a[i+k, j+m]\}$. 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];                    (24)
```

Step 4. Pruning (24) leaves the code unchanged.

Step 5. Initialize using (23) with $j = 0$ and form loop for $j = 1..n–m$ using (24) as loop body. We obtain

```
      init ⌈ sum[i, 0] := 0;
     using | for k := 0 to m−1 do
      (23) |   b[i+k, 0] := 0;
      with |   for l := 0 to m−1 do
     j = 0 |     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                                  (25)
           ⌈   sum[i, j] := 0;
 inc using |   for k := 0 to m−1 do
 (24) with |     b[i+k, j]:= b[i+k, j−1]−a[i+k, j−1]
   j dec 1 |                            +a[i+k, j−1+m];
           ⌊     sum[i, j] := sum[i, j] + b[i+k, j];
```

Finally, consider the outermost loop L_i.

Step 1. Its loop body is now (25); the first half contains the AACs A_k and A_l, and the second half contains, in the body of the loop on j, the incrementalized AAC of $b[i+k, j]$ and the 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 (25) with respect to $\oplus_i$. In the first half of the code, only A_l in $\{b[i+m, 0]\}$ needs to be computed; also, $decS(A_k, \oplus_i) = \{b[i, 0]\}$ and $incS(A_k, \oplus_i) = \{b[i+m, 0]\}$. In the second half, in the body of the loop on j, only $A_{k'}$ in $\{b[i+m, j]\}$ needs to be computed; also, $decS(A_j, \oplus_i) = \{b[i, j]\}$ and $incS(A_j, \oplus_i) = \{b[i+m, j]\}$. 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                                           (26)
  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];
```

Step 4. Pruning (26) leaves the code unchanged.

Step 5. Initialize using (25) with $i = 0$ and form loop for $i = 1..n–m$ using (26) as loop body. We obtain the optimized code in (27). Starred lines correspond to

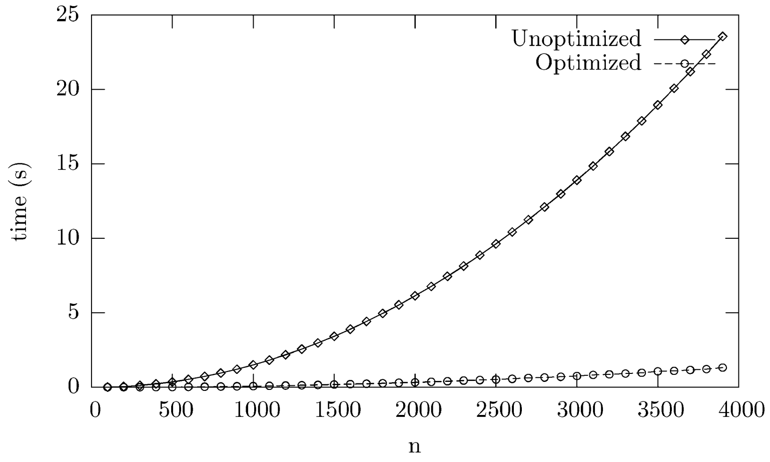

Fig. 5. Running time for image local neighborhood with $m = 20$.

the code in (2); other lines perform array margin initialization, mostly incrementally also.

```
             $sum[0,0] := 0$;
             for $k := 0$ to $m-1$ do
               $b[k,0] := 0$;
init           for $l := 0$ to $m-1$ do
using            $b[k,0] := b[k,0] + a[k,l]$;
(25)           $sum[0,0] := sum[0,0] + b[k,0]$;
with         for $j := 1$ to $n-m$ do
$i = 0$        $sum[0,j] := 0$;
               for $k := 0$ to $m-1$ do
                 $b[k,j] := b[k,j-1] - a[k,j-1] + a[k,j-1+m]$;
                 $sum[0,j] := sum[0,j] + b[k,j]$;                                   (27)
for-clause   for $i := 1$ to $n-m$ do                                         ⋆
               $b[i-1+m,0] := 0$;
               for $l := 0$ to $m-1$ do
inc              $b[i-1+m,0] := b[i-1+m,0] + a[i-1+m,l]$;
using          $sum[i,0] := sum[i-1,0] - b[i-1,0] + b[i-1+m,0]$;
(26)           for $j := 1$ to $n-m$ do                                       ⋆
with             $b[i-1+m,j] := b[i-1+m,j-1] - a[i-1+m,j-1]$
$i$ dec 1                       $+ a[i-1+m,j-1+m]$;                            ⋆
                 $sum[i,j] := sum[i-1,j] - b[i-1,j] + b[i-1+m,j]$;             ⋆
```

The cost analysis in incrementalization steps (24) and (26) ensures that the transformations are worthwhile when $m > 2$. In the resulting code (27), only four ± operations are performed for each pixel, independent of m. Thus, the optimized code takes $O(n^2)$ time.

Figures 5 and 6 show the running times and cache misses, respectively, for the unoptimized version (1) and the optimized version (27) when $m = 20$. Both measures confirm with the constant-factor improvements when a fixed factor based on m is eliminated. They also show that the rate of improvements for cache misses is smaller. The cache misses are reduced because AACs that sum m^2 array elements are replaced with two pairs of ±'s where each pair accesses

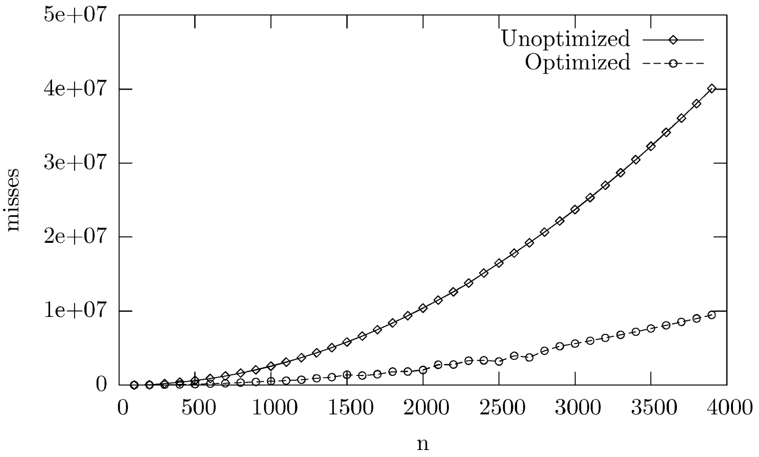

Fig. 6. Cache misses for image local neighborhood with $m = 20$.

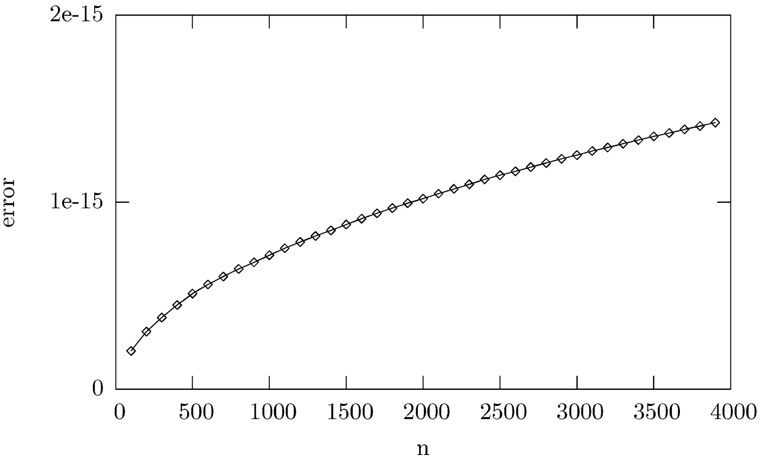

Fig. 7. Relative error for image local neighborhood with $m = 20$.

two elements that are only $m + 1$ positions apart. Figure 7 shows the relative error for $m = 20$. The error goes up gradually, as expected.

The effects of the optimization are also illustrated through measurements for $n = 2000$ and varying m. Figure 8 shows that, as m increases, the running time of the unoptimized code increases quadratically, while the optimized program's running time remains constant, as expected. Figure 9 shows that the number of cache misses increases mostly linearly first and then faster for the unoptimized code, but it remains nearly a constant for the optimized version. Finally, Figure 10 does not show any sort of simple correlation between m and the relative error. It seems that the error has a bound but it cycles and its period and amplitude increase; we hope that future work will help explain this precisely.

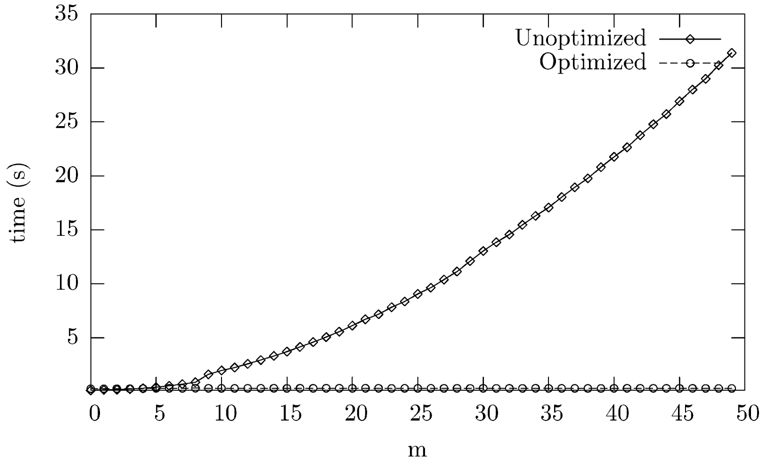

Fig. 8. Running time for image local neighborhood with $n = 2000$.

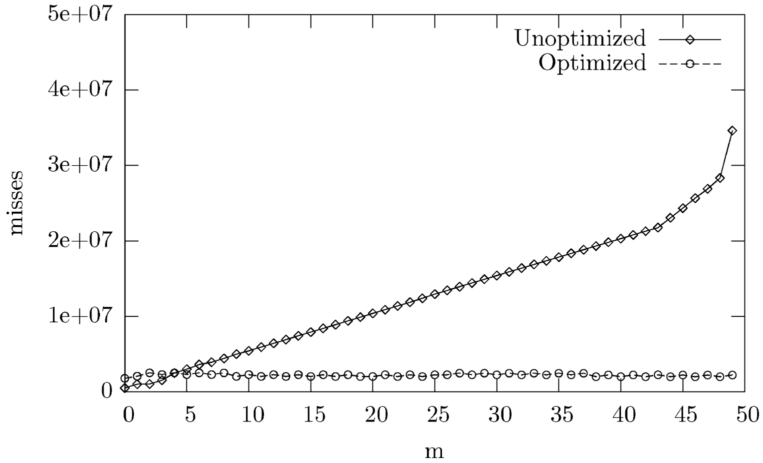

Fig. 9. Cache misses for image local neighborhood with $n = 2000$.

6.3 Sequence Local Average

Sequence local average is commonly used in sequence processing, for example, in computing 100-day stock averages and in taking averages of sampled data. Given an array $a[0..n{-}1]$ of numbers, for each index i from 0 to $n{-}k$ (in the outer **for**-clause), compute the sum of elements i to $i + k - 1$ (in the inner **for**-loop). The straightforward code (21) takes $O(nk)$ time.

```
for $i := 0$ to $n-k$ do
  $s := 0$;
  for $j := i$ to $i+k-1$ do                    (28)
    $s := s + a[j]$;
  $ave[i] := s/k$;
```

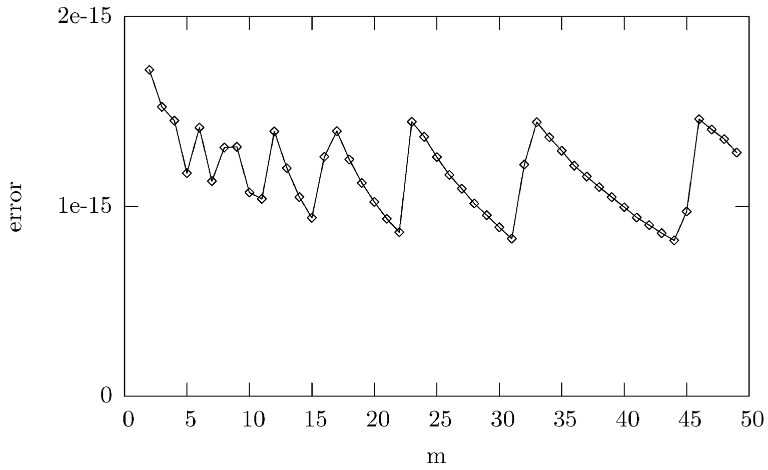

Fig. 10. Relative error for image local neighborhood with $n = 2000$.

The above code can be optimized using our algorithm as follows. 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 is the accumulating variable, and its loop increment is a SUO $\oplus_i$.

Step 2. Save the values of A_j in an array s indexed by i, yielding the same body of the loop over i except with each occurrence of s replaced by $s[i]$.

Step 3. $decS(A, \oplus_i) = \{a[i]\}$ and $incS(A, \oplus_i) = \{a[i + k]\}$. We obtain the incrementalized AAC $s[i + 1] := s[i] - a[i] + a[i + k]$.

Step 4. Pruning leaves the code unchanged.

Step 5. Initialize $s[0]$ using the body of the outer loop except with i replaced by 0 and s replaced by $s[0]$, and form the rest of the loop for $i = 1..n{-}1$ using the incrementalized AAC, we obtain the code (29). The optimized code takes only $O(n)$ time.

```
$s[0] := 0;$
**for** $j := 0$ **to** $k-1$ **do**
  $s[0] := s[0] + a[j];$
$ave[0] := s[0]/k;$                                        (29)
**for** $i := 1$ **to** $n-k$ **do**
  $s[i] := s[i-1] - a[i-1] + a[i-1+k];$
  $ave[i] := s[i]/k;$
```

This optimized code takes $O(n)$ extra space to store the value of $s[i]$. Using a more powerful pruning method, as discussed in Section 5.2, even a degenerate version, or using selective caching, also as discussed in Section 5.2, we could obtain a further optimized program that is exactly as in (29) except with all occurrences of $s[0]$ and $s[i]$ replaced with s and therefore uses no extra space. We call this the *further pruned* version.

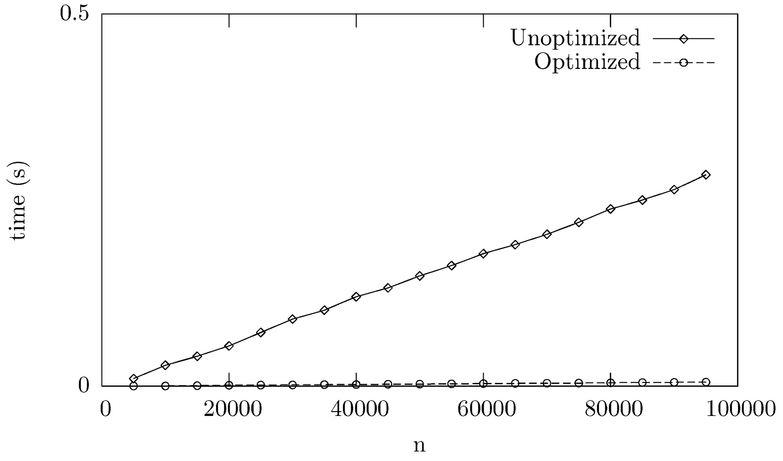

Fig. 11. Running time for sequence local average with $k = 1000$.

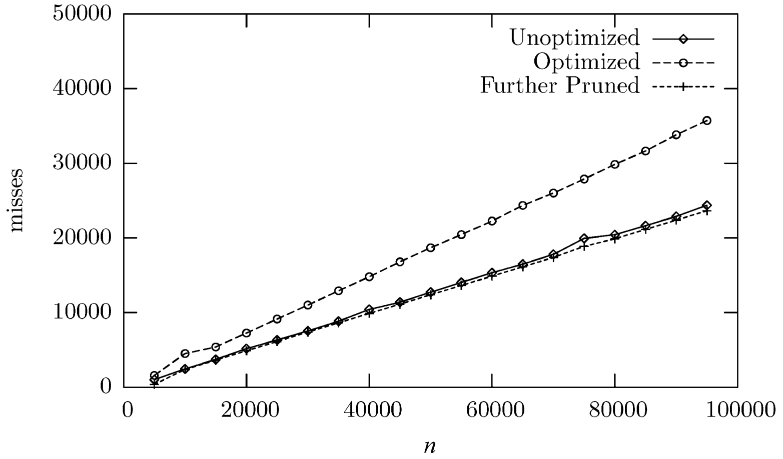

Fig. 12. Cache misses for sequence local average with $k = 1000$.

Running times for the unoptimized version (28) and the optimized version (29) are plotted in Figures 11 and 14, for $k = 1000$ and for $n = 20000 + k$, respectively. Figure 11 shows that the running time of the optimized code is extremely small when n is small. Figure 14 shows that the running time does not increase when k is increased.

Figure 12 shows cache misses of unoptimized, optimized, and further pruned version for $k = 1000$ and varying n. The cache behavior of the optimized version is actually worse than that of the unoptimized version, due to the use of array s to store cached values, but the cache behavior of the further pruned version is basically identical to the unoptimized version. Note that, even though the optimized version has more cache misses than the unoptimized version, the optimized program runs much faster, as shown in Figure 11.

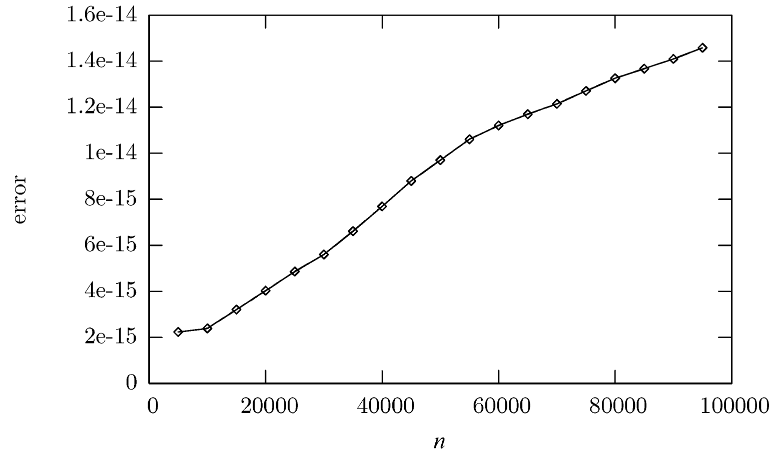

Fig. 13. Relative error for sequence local average with $k = 1000$.

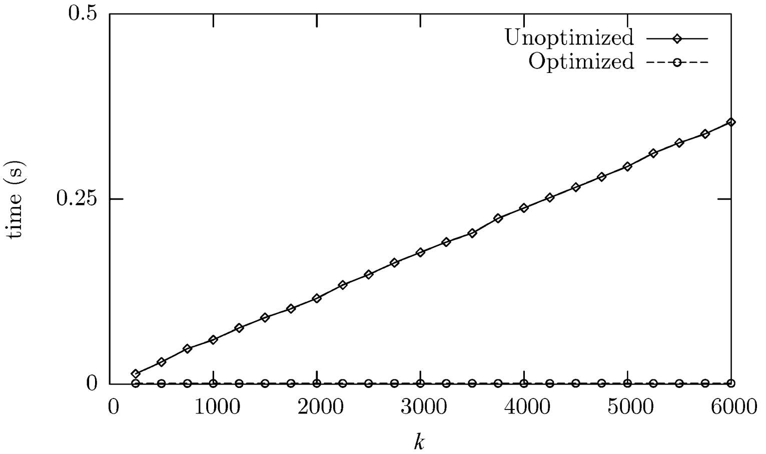

Fig. 14. Running time for sequence local average with $n = 20000 + k$.

Figure 15 shows cache misses for varying k and $n = 20000 + k$. For the unoptimized code, the number of misses stays small until k reaches 4000, after which the amount of memory accessed to compute each average exceeds the cache size, causing increasingly more cache misses; the irregularity in the extra cache misses appears to be the effect of a particular cache replacement strategy. For the optimized code, the number of cache misses remains basically constant except for the periodic spikes; it is larger than for the unoptimized code initially due to the use of the extra array s. We think that the periodic spikes are from loading chunks of memory into the cache: with three arrays interfering with one another, the misses add up periodically. For the further pruned version, the number of cache misses is smallest and stays constant except for the slight increase after $k = 4000$; it might be larger than for the unoptimized for one value of k slightly above 4000, but this effect is so small and rare that we did not observe in our experiments.

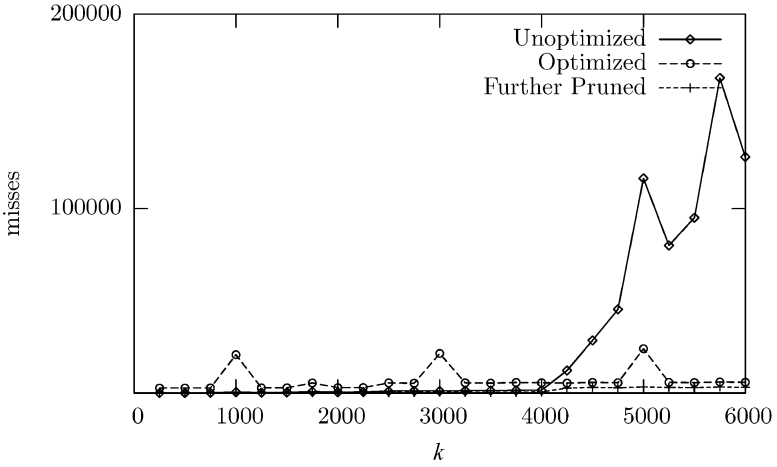

Fig. 15. Cache misses for sequence local average with $n = 20000 + k$.

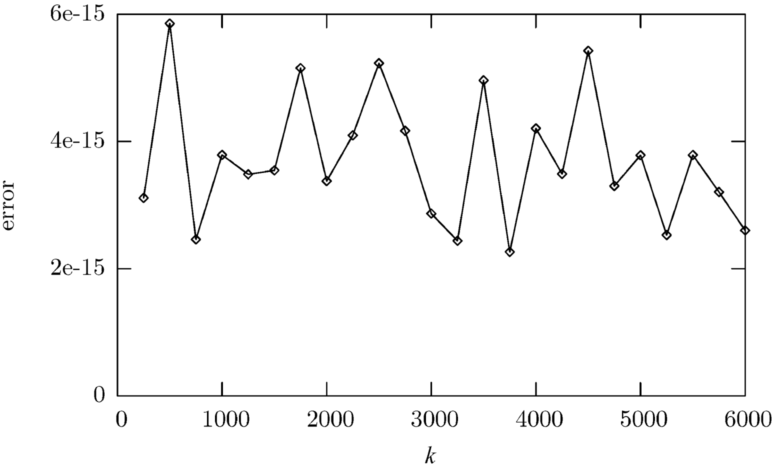

Fig. 16. Relative error for sequence local average with $n = 20000 + k$.

Figure 13 shows that the relative error in the result increases with n, that is, the error accumulates, as expected. Figure 16 shows no correlation between the value of k and the relative error.

7. RELATED WORK AND CONCLUSION

The basic idea of incrementalization is at least as old as Babbage's difference machine [Goldstine 1972]. *Strength reduction* is the first realization of this idea in optimizing compilers [Cocke and Kennedy 1977; Grau et al. 1967]. The idea is to compute certain multiplications in loops incrementally using additions. This work extends traditional strength reduction from arithmetic operations to aggregate array computations. Comparison with our other work on incrementalization [Liu 2000] appears in Section 1.

Finite differencing generalizes strength reduction to handle set operations in very-high-level languages like SETL [Earley 1976; Fong 1979; Fong and Ullman 1976; Paige and Koenig 1982]. The idea is to replace aggregate operations on sets with incremental operations. Similar ideas are used in the language INC [Yellin and Strom 1991], 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.

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 [Franksen 1985; Guibas and Wyatt 1978; Wegbreit 1976]. The same basic idea underlies techniques such as *fusion* [Allen and Cocke 1971; Allen 1983; Chin 1992; Goldberg and Paige 1984; Warren 1984], *deforestation* [Wadler 1988], and transformation of *series expressions* [Waters 1991]. These optimizations do not aim to compute each piece of the aggregate operations incrementally using previous pieces.

Specialization techniques such as *data specialization*, *run-time specialization and code generation*, and *dynamic compilation and code generation* [Consel and Noël 1996; ACM 1996] 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.

General *program transformations* [Burstall and Darlington 1977; Katz 1978] can be used for optimization, as demonstrated in projects like CIP [Bauer et al. 1989; Broy 1984]. In contrast to such manual or semi-automatic approaches, our optimization of aggregate array computations is fully automatable and requires no user intervention or annotations. Our method for maintaining additional values is an automatic method for *strengthening loop invariants* [Dijkstra 1976; Gries 1984].

Directionals are unary operations, such as LEFT and UP, invented by Fisher and Highnam [Fisher and Highnam 1988; Fisher et al. 1990; Highnam 1991], to describe computations involving small numbers of neighboring nodes on grid structures. Such computations are optimized by 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 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 [Banerjee 1990; Kelly and Pugh 1994; Sarkar and Thekkath 1992], *pipelining* [Allan et al. 1995], and *array data dependence analysis*

[Feautrier 1988, 1991; Maslov 1994; Pugh 1992; Pugh and Wonnacott 1998] 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 [Pugh 1992; Pugh and Wonnacott 1998]. Interestingly, ideas of incrementalization are used for optimizations in serializing parallel programs [Bromley et al. 1991; Ernst 1992].

In conclusion, this work describes a method and algorithms that allow more drastic optimizations of aggregate array computations than previous methods. Our techniques fall out of one general approach, optimization by incrementalization [Liu 2000], rather than simply being yet another new but ad hoc method. Applying incrementalization allows us to study important issues of programming cost, program performance, and trade-offs among running time, space consumption, code size, and cache behavior more explicitly and precisely than before. Future work includes improved analyses for costs and trade-offs, and optimizations for more general classes of aggregate computations.

ACKNOWLEDGMENTS

We would like to thank Erin Parker and Sid Chatterjee for applying their exact cache analysis to some of our examples, which suggested the improvements described in the first paragraph of Section 5.4 and which helped us understand the effect of our optimization on cache.

REFERENCES

AHMED, N., MATEEV, N., AND PINGALI, K. 2001. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Paral. Prog. 29*, 5 (Oct.), 493–544.

ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. J. 1995. Software pipelining. *ACM Comput. Surv. 27*, 3 (Sept.), 366–432.

ALLEN, F. E. AND COCKE, J. 1971. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*. R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J. pp. 1–30.

ALLEN, J. R. 1983. Dependence analysis for subscripted variables and its application to program transformations. Ph.D. dissertation, Rice Univ.

BANERJEE, U. 1990. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing* (Aug.), pp. 192–219.

BAUER, F. L., MÖLLER, B., PARTSCH, H., AND PEPPER, P. 1989. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng. 15*, 2 (Feb.), 165–180.

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

BOOTH, R. 1997. *Inner Loops*. Addison-Wesley Developers Press, Reading, Mass.

BROMLEY, M., HELLER, S., MCNERNEY, T., AND STEELE JR, G. L. 1991. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. ACM, New York, 145–156.

BROY, M. 1984. Algebraic methods for program construction: The project CIP. In *Program Transformation and Programming Environments*, P. Pepper, Ed., Springer-Verlag, Berlin, Germany. pp. 199–222.

BURSTALL, R. M., AND DARLINGTON, J. 1977. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan.), 44–67.

CAI, J., AND PAIGE, R. 1988/89. Program derivation by fixed point computation. *Sci. Comput. Program. 11* (Sept.), 197–261.

CARR, S., AND KENNEDY, K. 1994. Scalar replacement in the presence of conditional control flow. *Softw. Pract. Exp. 24*, 1, 51–77.

CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*. ACM, New York, pp. 286–297.

CHIN, W.-N. 1992. Safe fusion of functional expressions. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. ACM, New York, pp. 11–20.

COCKE, J., AND KENNEDY, K. 1977. An algorithm for reduction of operator strength. *Commun. ACM 20*, 11 (Nov.), 850–856.

CONSEL, C., AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

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

EARLEY, J. 1976. High level iterators and a method for automatically designing data structure representation. *J. Comput. Lang.* 1, 321–342.

ERNST, M. D. 1992. Serializing parallel programs by removing redundant computation. Master's thesis. MIT, Cambridge, Mass. (Revised August 1994.)

FABER, P., GRIEBL, M., AND LENGAUER, C. 2001. Loop-carried code placement. In *Proceedings of the 7th International Euro-Par Conference*. Lecture Notes in Computer Science, vol. 2150. Springer-Verlag, New York.

FEAUTRIER, P. 1988. Parametric integer programming. *Operationnelle/Oper. Res. 22*, 3 (Sept.), 243–268.

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Int. J. Parall. Prog. 20*, 1 (Feb.).

FISHER, A. L. AND HIGHNAM, P. T. 1988. Communication and code optimization in SIMD programs. In *International Conference on Parallel Processing* (Aug.).

FISHER, A. L., LEON, J., AND HIGHNAM, P. T. 1990. Design and performance of an optimizing SIMD compiler. In *Frontiers of Massively Parallel Computation*.

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

FONG, A. C. AND ULLMAN, J. D. 1976. Inductive variables in very high level languages. In *Conference Record of the 3rd Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, pp. 104–112.

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

GARG, V. K. AND MITCHELL, J. R. 1996. An efficient algorithm for detecting conjunctions of general global predicates. Tech. Rep. TR-PDS-1996-005. University of Texas at Austin, Austin, Tex.

GOLDBERG, A. AND PAIGE, R. 1984. Stream processing. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*. ACM, New York, pp. 53–62.

GOLDSTINE, H. H. 1972. Charles Babbage and his analytical engine. In *The Computer from Pascal to von Neumann*, Princeton University Press, Princeton, N.J., Chap. 2, 10–26.

GOUBAULT, E. 2001. Static analyses of the precision of floating-point operations. In *Proceedings of the 8th International Static Analysis Symposium*, Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, Berlin, Germany.

GRAU, A. A., HILL, U., AND LANGMAAC, H. 1967. *Translation of ALGOL 60*. Volume 1 of *Handbook for Automatic Computation*. Springer, Berlin, Germany.

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

Guibas, L. and Wyatt, K. 1978. Compilation and delayed evaluation in APL. In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York. pp. 1–8.

Higham, N. J. 1996. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, Pa.

Highnam, P. T. 1991. *Systems and Programming Issues in the Design and Use of a SIMD Linear Array for Image Processing*. Ph.D. dissertation. Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa.

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

Kelly, W. and Pugh, W. 1994. Finding legal reordering transformations using mappings. In *Proceedings of the 7th Annual Workshop on Programming Languages and Compilers for Parallel Computing*. (Ithaca, N.Y. Aug.) Lecture Notes in Computer Science, vol. 829. Springer-Verlag, New York.

Knuth, D. E. 1971. An empirical study of FORTRAN programs. *Softw. Pract. Exp. 1*, 2, 105–133.

Li, K. 1997. Interferometric strain/slope rosette for static and dynamic measurements. *Exper. Mech. 37*, 2, 111–118.

Liu, Y. A. 1997. Principled strength reduction. In *Algorithmic Languages and Calculi*, R. Bird and L. Meertens, Ed. Chapman & Hall, London, U.K. 357–381.

Liu, Y. A. 2000. Efficiency by incrementalization: An introduction. *High. Ord. Symb. Comput. 13*, 4 (Dec.), 289–313.

Liu, Y. A. and Stoller, S. D. 2002. Program optimization using indexed and recursive data structures. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, New York, 108–118.

Liu, Y. A. and Stoller, S. D. 2003. Dynamic programming via static incrementalization. *High. Order Symb. Comput. 16* 1-2 (Mar.–June), 37–62. (An earlier version appeared in *Proceedings of the 8th European Symposium on Programming*, 1999.)

Liu, Y. A., Stoller, S. D., and Teitelbaum, T. 1998. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst. 20*, 3 (May), 546–585. (An earlier version appeared in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1995.)

Liu, Y. A., Stoller, S. D., and Teitelbaum, T. 2001. Strengthening invariants for efficient computation. *Sci. Comput. Program. 41*, 2 (Oct.), 139–172. (An earlier version appeared in *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, 1996.)

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

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

Paige, R. 1990. Symbolic finite differencing—Part I. In *Proceedings of the 3rd European Symposium on Programming*, N. D. Jones, Ed. Lecture Notes in Computer Science, vol. 432. Springer-Verlag, Berlin, Germany, pp. 36–56.

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

Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation. ACM, New York, 1996.

Pugh, W. 1991. Uniform techniques for loop optimization. In *International Conference on Supercomputing* (Cologne, Germany, June), pp. 341–352.

Pugh, W. 1992. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Commun. ACM 31*, 8 (Aug.), 102–114.

Pugh, W. and Wonnacott, D. 1998. Constraint-based array dependence analysis. *ACM Trans. Program. Lang. Syst. 20*, 3, 635–678.

Reps, T. and Teitelbaum, T. 1988. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York.

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

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

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

Warren, J. 1984. A hierarchical basis for reordering transformations. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York. 272–282.

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

Webb, J. A. 1992. Steps towards architecture-independent image processing. *IEEE Comput. 25*, 2 (Feb.), 21–31.

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

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

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

Zabih, R. 1994. Individuating unknown objects by combining motion and stereo. Ph.D. dissertation. Department of Computer Science, Stanford University, Stanford, Calif.

Zabih, R. and Woodfill, J. 1994. Non-parametric local transforms for computing visual correspondence. In *Proceedings of the 3rd European Conference on Computer Vision*, J.-O. Eklundh, Ed. Lecture Notes in Computer Science, vol. 801. Springer-Verlag, New York, pp. 151–158.

Received June 2002; revised April 2003 and June 2003; accepted October 2003