On Minimizing Materializations of Array-Valued Temporaries

DANIEL J. ROSENKRANTZ, LENORE R. MULLIN and HARRY B. HUNT III
University of Albany—SUNY

We consider the analysis and optimization of code utilizing operations and functions operating on entire arrays. Models are developed for studying the minimization of the number of materializations of array-valued temporaries in basic blocks, each consisting of a sequence of assignment statements involving array-valued variables. We derive lower bounds on the number of materializations required, and develop several algorithms minimizing the number of materializations, subject to a simple constraint on allowable statement rearrangement. In contrast, we also show that when statement rearrangement is unconstrained, minimizing the number of materializations becomes NP-complete, even for very simple basic blocks.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*; E.1 [**Data Structures**]: *Arrays*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: Array operations, array temporary materializations, compiler optimization, copy minimization, program transformation

1. INTRODUCTION

We consider the analysis and optimization of code utilizing operations and functions operating on entire arrays or array sections (rather than just on the underlying scalar domains of the array elements). We call such operations *monolithic* array operations. A programming style using array operands and monolithic array operations can focus on what a computation does at a high level of abstraction, rather than on implementation details. For example, a single monolithic assignment statement with an array expression on the

This research was supported by National Science Foundation (NSF) Grant CCR-0105536.
A preliminary version containing some of the results in this article appeared in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing 2000 (LCPC 2000)* (Yorktown Heights, NY, Aug.), S. P. Midkiff et al. Eds. Lecture Notes in Computer Science, vol. 2017, Springer-Verlag, New York, 2000, pp. 127–146.
Authors' address: Computer Science Department, University of Albany—SUNY, Albany, NY 12222; email: {djr,lenore,hunt}@cs.Albany.edu.

righthand side is equivalent to nested loops using scalar operations on individual array elements. Additional advantages of using such monolithic statements include improved program clarity, increased exposure of potential parallelism, increased tunability to target architecture, and increased flexibility in potential optimizations. Indeed, many scientific programmers find it easier and more natural to program using monolithic array operations. For example, MATLAB, widely used as a prototyping language for the development of scientific software, makes heavy use of monolithic array operations. In addition, Fortran 90 has built-in monolithic array operations, and C++ is often accompanied by libraries that provide monolithic array operations. This demonstrated appeal of programming with monolithic array operations suggests that analysis and optimization of such programs will be of growing importance. An important aspect of such analysis and optimization is the elimination of array-valued temporaries. In this article, we study materializations of array-valued temporaries, with a focus on elimination of array-valued temporaries in basic blocks.[1]

Efficiency is very important in scientific computing, and consequently scientific software must be highly optimized. Ideally, scientific programmers should be able to program using monolithic array operations, and have their programs automatically optimized to produce very efficient object code. Currently, the dominant approach to compiler optimization of array languages, Fortran 90, HPF, etc., is to first scalarize the monolithic code, and then to do *transformational* optimizations on the resulting scalarized loops. However, much global information is obfuscated, or even lost, during the scalarization process. In contrast, the results presented here (and in Chamberlain et al. [1996], Humphrey et al. [1997], Hwang et al. [1996, 2001], Lewis et al. [1998], Roth [2000], Roth and Kennedy [1996], and Veldhuizen and Gannon [1998]) illustrate how a programming style using monolithic array operations, and program analysis of such programs prior to scalarization, can be used to perform high-level transformations.

There has been extensive research on nonmaterialization of array-valued subexpressions in evaluating array-valued *expressions*, as described in Section 5. Eliminating materializations of temporaries is an issue not only for partial results within an expression, but more generally also for assignment statements within a basic block. In particular, many programmers tend to avoid writing a single assignment statement whose right-hand side is a long complicated expression. Instead, they write a sequence of assignment statements, each with a relatively simple righthand side. The array definitions corresponding to the left-hand sides of these intermediate assignment statements are targets of opportunity for nonmaterialization. A complicating factor that occurs in basic blocks is that an array variable occurring on the righthand side of an assignment statement can itself be the target of a subsequent assignment statement. Consequently, nonmaterialization in basic blocks is more complicated than nonmaterialization in expressions. Some initial results on nonmaterialization in

[1]More generally, our results are applicable to the optimization of straight-line code, including C++-like template and macro bodies consisting of straight-line code.

basic blocks appear in Kennedy et al. [1995], Roth [1997], and Roth et al. [1997], as discussed in Section 2.1.

In Section 2, we develop a framework and techniques for the study of nonmaterialization in basic blocks. We give examples of how code can be improved via nonmaterialization in basic blocks, and then formalize the problem of minimizing materializations as an optimization problem. We also develop two graph-theoretic models that capture fundamental concepts of the relationship between array values in a basic block.

In Section 3, we study the problem of minimizing materializations subject to a natural constraint on the rearrangement of statements in a basic block. Several additional properties of the graph-theoretic models introduced in Section 2 are identified. These properties are used to develop two lower bounds on the number of materializations subject to the statement rearrangement constraint. Two classes of problem instances are identified, and algorithms are presented for these classes that actually minimize the number of materializations. In fact, the number of materializations produced by each algorithm equals the applicable lower bound. A nontrivial upper bound for arbitrary problem instances, and an associated algorithm obtaining this bound, are also presented.

In Section 4, we present results complementing those in Section 3, by showing that when the statement rearrangement constraint of Section 3 is removed, minimizing materializations is NP-complete, even for very simple basic blocks.

In Section 5, we discuss related work, and in Section 6, we give brief conclusions.

2. NONMATERIALIZATION IN BASIC BLOCKS

2.1 Examples of Nonmaterialization

We use the term, *materialization*, to refer to code that computes an array at run time. Many array operations, expressed in Fortran 90 and HPF in terms of sectioning and such intrinsics as `transpose`, `cshift`, `reshape`, and `spread`, involve the rearrangement and replication of array elements. We refer to these operations as (address) *shuffle operations*. (Note that the `eoshift` operation involves deletions, and for the purposes of this article is not regarded as a shuffle operator. In Section 6, we briefly discuss how our results can be extended to handle `eoshift` and other shuffle-like operations involving deletions.) Shuffle operations construct new arrays, all of whose elements are elements of their array operands. The construction of such a new array essentially only utilizes array indexing into the array operands, and is independent of the domain of values of the scalars involved. Often it is unnecessary to actually materialize the results of a shuffle operation. Rather, a compiler or other processor can keep track of how elements of the resulting array can be obtained by appropriately indexing the operands of the shuffle operation. Subsequent references to the result of the operation can be implemented in terms of suitably modified references to the operands of the operation. We call this technique *nonmaterialization*.

As an example of nonmaterialization, consider the following statement.

```
X = B + transpose(C).
```

Straightforward code to implement this statement would first materialize the result of the `transpose` operation in a temporary array, say named `Y`, and then add `Y` to `B`, assigning the result to `X`.

Even this single statement illustrates the need for nonmaterialization applied to code fragments at a higher level than single statements or expressions. Indeed, instead of using a single assignment statement, the programmer might well have expressed the above statement as a sequence of two statements, forming part of a basic block, as follows:

```
Y = transpose(C)
X = B + Y.
```

The longer the expression calculating the value of `X`, the more likely it is that a programmer will calculate `X` via a sequence of statements. Scalarizing and optimizing each assignment separately might produce the following code,[2] with a separate loop for each of the above statements:

```
forall (i = 1:N, j = 1:N)
  Y(i,j) = C(j,i)
end forall
forall (i = 1:N, j = 1:N)
  X(i,j) = B(i,j) + Y(i,j)
end forall
```

A code optimizer could *fuse* the two loops, producing the following single loop, in which `Y` and `X` are both materialized.

```
forall (i = 1:N, j = 1:N)
  Y(i,j) = C(j,i)
  X(i,j) = B(i,j) + Y(i,j)
end forall
```

In contrast, a compiler could determine that `Y` need not be materialized at all, and generate the following code:[3]

```
forall (i = 1:N, j = 1:N)
  X(i,j) = B(i,j) + C(j,i)
end forall
```

Nonmaterialization can also be used in optimizing distributed computation. Kennedy et al. [1995], Roth [1997], and Roth et al. [1997] develop methods for optimizing *stencil* computations, which are of importance in scientific computing, by nonmaterialization of selected `cshift` operations involving distributed arrays. These nonmaterializations are used to minimize communications, including both the amount of data transmitted, and the number of messages used. The amounts of the shifts in the stencils involved are constants, so a compiler can determine which `cshift` operations in the basic block are potential beneficiaries of nonmaterialization. A compiler can analyze a basic block and choose not to materialize some of these `cshift` operations. The nonmaterialization

[2]This code could alternately have been expressed using `DO loops`.

[3]This simple example is given to illustrate nonmaterialization. For this example, many compilers would produce this code using standard techniques, without the need for more sophisticated optimization techniques.

technique in Kennedy et al. [1995], Roth [1997], and Roth et al. [1997] uses a sophisticated form of replication, where a subarray on a given processor is enlarged by adding extra rows and columns that are replicas of data on other processors. The actual computation of the stencil on each processor refers to the enlarged array on that processor.

For instance, consider the following example, involving 2-dimensional arrays, taken from Roth et al. [1997].

```
(1)     RIP = cshift(U,shift=+1,dim=1)
(2)     RIN = cshift(U,shift=-1,dim=1)
(3)     T = U + RIP + RIN
```

The optimized version of this code is the following. `RIP` and `RIN` are not materialized. The `cshift` operations in statements (1) and (2) are replaced by `overlap_cshift` operations, which transmit enough data from `U` between processors so as to fill in the overlap areas on each processor. In statement (3), the references to `RIP` and `RIN` are replaced by references to `U`, each annotated with appropriate shift values, expressed using superscripts to indicate the shift value in each dimension.

```
(1)     call overlap_cshift(U,shift=+1,dim=1)
(2)     call overlap_cshift(U,shift=-1,dim=1)
(3)     T = U + U<+1,0> + U<−1,0>
```

An important issue for nonmaterialization is determining whether a given *shuffle* operation is a *candidate* for nonmaterialization. The criteria for a given shuffle operation to be a *candidate* are that nonmaterialization of the operation be both *safe* and *profitable*. The usual criteria for a nonmaterialization to be safe are that the source array of the shuffle is not modified while the definition of the destination array is live, and that the destination array is not partially modified while the source array is live [Schwartz 1975; Roth 1997]. (Note that these criteria implicitly assume that shuffle operations have a single array operand.) The criteria for being profitable depend on the optimization goal, the shuffle operation involved, the *shape* of the arrays involved, architectural features of the computing environment on which the computation will be performed, and the distribution/alignment of the arrays involved. For instance, Roth [1997] gives profitability criteria for stencil computations on distributed machines. For purposes of this article, we assume the availability of appropriate criteria for determining whether a given shuffle operation is a candidate for nonmaterialization.

Definition 2.1. An *eligible statement* is an assignment statement whose right side is a shuffle operation that is a candidate for nonmaterialization. An *ineligible statement* is a statement that is not an eligible statement. We say that the array value occurring on the right side of an eligible statement is *mergeable* with the array value occurring on the left side of the statement.

In a given basic block, it may not be possible to avoid materializing all the eligible statements. Moreover, the decisions as to which eligible statements should be nonmaterialized can be interdependent. This interdependence is an impediment to using techniques for optimizing expression trees or dags. Roth [1997]

gives a "greedy" algorithm for choosing which eligible statements to nonmaterialize. At each step, Roth's algorithm chooses the first eligible statement, and modifies the basic block so that this statement is not materialized. Each modification can cause other statements that were eligible before the modification to become ineligible. The following example illustrates this phenomenom.

Example 2.1

```
(1)     B = cshift(A,shift=+5,dim=1)
(2)     C = cshift(A,shift=+3,dim=1)
(3)     D = cshift(A,shift=-2,dim=1)
(4)     B(i) = 50
(5)     R(2:199) = B(2:199) + C(2:199) + D(2:199)
```

At end of basic block: R *live;* A, B, C, *and* D *dead.*

Assume that statements (1), (2), and (3) are eligible statements. Roth's method would first consider the shuffle operation in statement (1), and consequently chooses to merge B with A. This merger would be carried out as shown below, changing the modification of B in statement (4) into a modification of A. In statements (4) and (5), the reference to B is replaced by a reference to A, annotated with the appropriate shift value.

```
(1)     call overlap_cshift(A,shift=+5,dim=1)
(2)     C = cshift(A,shift=+3,dim=1)
(3)     D = cshift(A,shift=-2,dim=1)
(4)     A<+5>(i) = 50
(5)     R(2:199) = A<+5>(2:199) + C(2:199) + D(2:199)
```

Statement (4) now modifies variable A, thereby making statements (2) and (3) unsafe, and therefore ineligible. This prevents C and D from being merged with A. The greedy algorithm would terminate at this point, since there are no eligible statements, leaving three copies of the array (in variables A, C, and D). But, it is better to optimize Example 2.1 by materializing a separate copy of A in B, and letting A, C, and D all share the same copy (in variables A), as follows:

```
(1)     B = cshift(A,shift=+5,dim=1)
(2)     call overlap_cshift(A,shift=+3,dim=1)
(3)     call overlap_cshift(A,shift=-2,dim=1)
(4)     B(i) = 50
(5)     R(2:199) = B(2:199) + A<+3>(2:199) + A<-2>(2:199)
```

The above example can be extended to show that the "greedy" algorithm can be arbitrarily worse than optimal. Suppose that instead of just C and D, there were n additional variables that could be merged with A. Roth's method would only merge B with A, resulting in a total of $n+1$ materialized arrays. In contrast, by making a separate copy for B, there would be a total of only two materialized arrays.

We now illustrate another technique that can facilitate minimizing materializations. Placing the materialized result of a nonshuffle array statement in a different variable than given in the statement may permit subsequent shuffle operations to be nonmaterialized. This freedom in selecting the variable in

which to place a materialization is necessary for certain optimizations to be possible, as illustrated by the following example:

Example 2.2

```
(1)     A = C + D
(2)     B = cshift(A,shift=+3,dim=1)
(3)     z = A(i) + B(i)
```

At end of basic block: A *dead,* B *live.*

Since variable A is dead at the end of the basic block, and B is live, it is advantageous to materialize the result computed by statement (1) in variable B instead of variable A. Thus, an optimized version of the basic block might be as follows. Here, the partial result C + D is annotated by the compiler to indicate that the value to be stored in variable B should be shifted appropriately.

```
(1′)        B = (C + D)^<+3>
(3)         z = B^<−3>(i) + B(i)
```

2.2 Definitions and Problem Statement

Next, we begin our development of a framework for considering the problem of minimizing the number of materializations in a basic block (equivalently, maximizing the number of shuffle operations that are not materialized).

Definition 2.2. A *def-value* in a basic block is either the initial value (at the beginning of the basic block) of an array variable or an occurrence of an array variable that is the destination of an assignment. (An assignment can be to the complete array, to a section of the array, or to a single element of the array.)

Definition 2.3. A *complete-def* is a def-value resulting from an assignment to a complete array. A *partial-def* is a def-value resulting from an assignment to a section or element of an array (where the other array elements retain their prior values). An *initial-def* is an initial value of an array variable.

Note that the set of def-values can be partitioned into the sets of complete-defs, partial-defs, and initial-defs. For convenience, we identify each def-value by the name of its variable, superscripted with either 0 for an initial-def, or the statement assigning the def-value for a noninitial-def. Using this notation, Example 2.1 involves initial-defs A^0 and R^0, complete-defs B^1, C^2, and D^3, and partial-defs B^4 and R^5.

Definition 2.4. A *materialization* is either an initial-def or a complete-def.

We say that a given materialization is *in* the variable of its def-value, since the def-value is stored in the location corresponding to this variable.

We now consider sets of mutually mergeable def-values, represented as trees in the *clone forest*, defined as follows:

Definition 2.5. The *clone forest* for a basic block is a graph with a node for each def-value occurring in the basic block, and a directed edge from the source def-value to the destination def-value of each eligible shuffle operation. The set of def-values occurring in a tree of the clone forest is called a *clone set*,

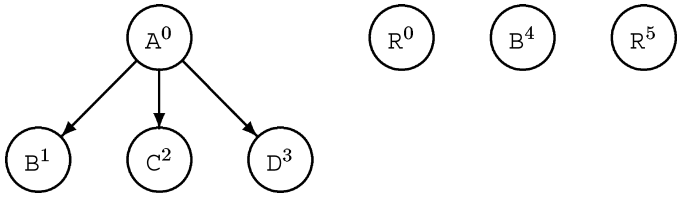

Fig. 1. Clone forest for Example 2.1.

and these def-values are said to be *clones* of each other. The *root def-value* of a given clone set is the def-value corresponding to the root node of the clone tree corresponding to the clone set.

As an example, the clone forest for Example 2.1 is shown in Figure 1. There are four clone sets, $\{A^0, B^1, C^2, D^3\}$, $\{R^0\}$, $\{B^4\}$ and $\{R^5\}$, with root def-values A^0, R^0, B^4 and R^5, respectively.

The interface between a given basic block and the rest of the program is through those variables that are live at the beginning or end of the basic block. These variables can be identified using data-flow analysis [Aho et al. 1986]. We assume that each such variable has an *official location*, where its value will be stored at block entry and/or exit.

We formalize the materialization problem via the following assumptions:

Materialization Problem Assumptions:

(1) *Only Eligible Nonmaterialized Assumption*. The only statements that can be nonmaterialized are eligible statements. A materialization can occur in a different variable than in the given code, and a reference to a def-value can be replaced by a reference to a clone, with appropriate annotation.
(2) *No Dead Code Assumption*. No dead code.
(3) *No Aliasing Assumption*. Arrays with different names are not aliased.[4]
(4) *Full Mergeability Assumption*. The relation between def-values in a basic block, based on being *mergeable*, is symmetric and transitive.
(5) *Coarse Analysis Assumption*. There is no fine-grained analysis of array indices. In particular, no analysis is made as to whether two given partial-defs overlap or whether a given use overlaps a given partial-def.
(6) *Live Entry Assumption*. The initial value of each variable that is live on entry to a basic block is stored in the variable's *official location*.
(7) *Live Exit Assumption*. A variable that is live at exit from a basic block, must have its final value at the end of the basic block stored in the variable's *official location*.
(8) *Restricted Rearrangement Assumption*. No rearrangement of the ineligible statements in a basic block.

[4]However, the techniques in this article can be suitably modified to use aliasing information.

In Section 3, we focus on the optimization problem of minimizing the number of materializations in a basic block, under all eight assumptions. In Section 4, we consider this problem under Assumptions (1)–(7) only.

Definition 2.6. The *Materialization Problem Under No Ineligible Statement Rearrangement* is the problem of minimizing the number of materializations in a basic block, under Assumptions (1)–(8) listed above. The *Materialization Problem Under Unconstrained Statement Rearrangement* is the problem of minimizing the number of materializations in a basic block, under Assumptions (1)–(7) listed above.

The overall optimization problem can be viewed as minimizing the total number of materializations for all the clone sets for a given basic block. Consider Example 2.1 again. Def-values $\mathtt{A}^0$ and $\mathtt{R}^0$ are initial-defs. From Assumption 6, there is a materialization for each initial-def, so materializations are needed for $\mathtt{A}^0$ and $\mathtt{R}^0$. Def-value $\mathtt{R}^5$ can be obtained by a partial-def that modifies the already materialized initial-def $\mathtt{A}^0$, and so does not require a new materialization. Def-value $\mathtt{B}^4$ (or, possiblly a clone of $\mathtt{B}^4$) must be obtained by a partial-def that modifies $\mathtt{A}^0$ or a clone of $\mathtt{A}^0$. Statement (5) uses values from both clone set $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$ and clone set $\{\mathtt{B}^4\}$. Thus, at the time statement (5) is executed, a member of each of these clone sets must be available. Since def-values from two different clone sets must be simultaneously available, two def-values used in statement (5) must be stored in different variables, and so are stored in the variables of different materializations. Clone set $\{\mathtt{B}^4\}$ is created by the partial-def in statement (4) that modifies a member of clone set $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$. Thus, at least two members of clone set $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$ must be materialized: one to hold a def-value from this clone set for use in statement (5), and one to hold a def-value that is modified by statement (4). The optimized code given for Example 2.1 materializes $\mathtt{A}^0$ and $\mathtt{B}^1$ from clone set $\{\mathtt{A}^0, \mathtt{B}^1, \mathtt{C}^2, \mathtt{D}^3\}$, and materializes $\mathtt{R}^0$ from clone set $\{\mathtt{R}^0\}$. Clone sets $\{\mathtt{B}^4\}$ and $\{\mathtt{R}^5\}$ do not require new materializations; they can be obtained by partial-defs to already materialized arrays.

We now establish a lower bound on the number of separate copies required for a given clone set. First, we identify two important kinds of def-values.

Definition 2.7. A given def-value in a basic block is *transient* if the basic block contains a subsequent def-value in its variable, and the next such def-value in its variable is a partial-def.

A given def-value in a basic block is *persistent* if it is the last def-value in its variable in the basic block, and its variable is live at the end of the basic block.

For instance, in Example 2.1, def-value $\mathtt{B}^1$ is transient, because the next def-value in its variable B is a partial-def, namely $\mathtt{B}^4$. Def-value $\mathtt{R}^0$ is transient because of the subsequent partial-def $\mathtt{R}^5$. Def-value $\mathtt{R}^5$ is persistent. Note that def-values $\mathtt{A}^0$, $\mathtt{C}^2$, $\mathtt{D}^3$, and $\mathtt{B}^4$ are neither transient nor persistent.

PROPOSITION 2.1. *Consider the clone tree for a given clone set. There needs to be a separate materialization for each def-value in the clone tree that is either transient or persistent.*

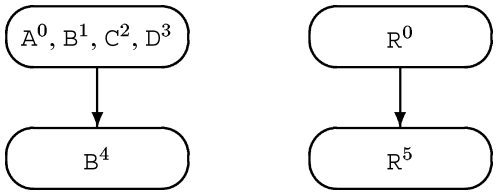

Fig. 2. Version forest for Example 2.1.

PROOF. Consider a given clone tree. We need to consider the following three cases of pairs of def-values in the clone tree: (1) two transient def-values, (2) two persistent def-values, and (3) a transient def-value and a persistent def-value.

Case 1. By Definition 2.7, two transient def-values in the clone tree are each modified by subsequent partial-defs. Under Assumption 5, these partial-defs possibly can assign new values to a common element of the array. From Assumption 2, the new arrays resulting from these partial-defs are not dead. From Assumption 5 again, the two values for a common array element might both be needed. Thus, the materializations containing these common elements must be different.

Case 2. Two persistent def-values in the same clone tree cannot share the same materialization because, from Assumption 7, the two persistent def-values must be in different variables.

Case 3. A transient def-value and a persistent def-value cannot share the same materialization because from Assumption 2 each is needed, from Assumption 7 the persistent def-value must be in the variable corresponding to its official location, and this value must be unchanged at the end of the basic block, but the transient def-value is modified by a subsequent partial-def. □

For each transient def-value of a given clone set, there is a subsequent partial-def in its variable, and this partial-def is the root def-value of another clone set. We now define a graph that represents the relationship between clone sets based on partial-defs. This graph contains a node for each clone set, and an edge for each transient-def, as formalized by the following definition.

Definition 2.8. The *version forest* for a basic block is a graph with a node for each clone set in the basic block, and a directed edge from clone set α to clone set β if the root def-value of clone set β is a partial-def that modifies a member of α. The *root def-value* of a given tree in the version forest is the root def-value of the clone set of the root node of the version tree.

For example, the version forest for Example 2.1 is shown in Figure 2. There are two trees in the version forest, with root def-values $\mathtt{A}^0$ and $\mathtt{R}^0$, respectively.

Definition 2.9. A node of a version forest is *persistent* if any of its def-values are persistent.

For instance, in Figure 2, the only persistent node in the version forest is the node for clone set $\{\mathtt{R}^5\}$.

3. MINIMIZING MATERIALIZATIONS UNDER NO INELIGIBLE STATEMENT REARRANGEMENT

3.1 Overview of Section

In this section, we develop our main techniques and results on the *Materialization Problem Under No Ineligible Statement Rearrangement*. This problem consists of minimizing the number of materializations in a basic block, under the eight assumptions listed in Section 2.2. Here, of particular importance is the constraint of no rearrangement of ineligible statements imposed by Assumption 8. (Note that Assumption 8 places no constraint on the rearrangement of eligible statements.) We develop both lower bounds and algorithms for this problem. For reasons of clarity, both the lower bounds and the algorithms are developed in two stages. The first-stage results (Sections 3.2 and 3.4) are conceptually simpler and explicate the underlying logic more clearly. The second-stage results (Sections 3.3 and 3.5) are stronger, but more complicated.

In Section 3.2, we develop the simpler lower bound. To express this lower bound, we first formulate key concepts that are fundamental to the minimization problem. The concept of an *essential node* (Definition 3.3) is of particular importance. We show (Corollary 3.1) that the number of essential nodes of a basic block is a lower bound on the number of materializations occurring in the basic block. Next we show (Theorem 3.2) that several properties of a basic block are invariant under any transformation of a basic block permitted by the assumptions of Section 2.2. These properties include the number of essential nodes. Consequently, we conclude (Theorem 3.3) that the number of essential nodes in a basic block is a lower bound on the number of materializations in any optimized version of the basic block.

In Section 3.3, we first formulate additional concepts in order to generalize the lower bound from Section 3.2. Theorem 3.4 generalizes Corollary 3.1 to provide a stronger lower bound on the number of materializations occurring in a basic block. Then Theorem 3.5 generalizes Theorem 3.2 to include additional properties of a basic block that are invariant under any transformation of a basic block permitted by the assumptions of Section 2.2. We conclude (Theorem 3.6) that the lower bound from Theorem 3.4 is a lower bound on the number of materializations in any optimized version of the basic block.

In Section 3.4, we present an algorithm (Theorem 3.7) minimizing materializations in basic blocks with no persistent def-values. For such basic blocks, the number of materializations produced by the algorithm equals the lower bound of Theorem 3.3. Thus (Theorem 3.8), the algorithm constructs an optimum solution to the minimization problem for these basic blocks.

In Section 3.5, we present two algorithms for basic blocks with persistent def-values. We define the concept of a *persistence conflict* between two version trees. These conflicts occur when there is a possible interaction preventing simultaneous optimization of both version trees. A persistence conflict can also occur within a version tree if an initial def-value must be shuffled to produce a persistent def-value in the same variable. We present an algorithm (Theorem 3.9) minimizing materialiations in basic blocks with no persistence conflicts. For

such basic blocks, the number of materializations produced by this algorithm equals the lower bound of Theorem 3.6. Thus (Theorem 3.10), the algorithm constructs an optimum solution to the minimization problem for these basic blocks. Finally, we note that each persistence conflict in a given basic block can be broken by adding an additional materialization. This provides an algorithm and upper bound (Theorem 3.11) for basic blocks with persistence conflicts.

3.2 A Simple Lower Bound

Next we formalize the concept of the point in a basic block where a clone set is created. Without loss of generality, we assume that the statements in a given basic block are numbered consecutively, beginning with 1.

Definition 3.1. The *origin point* of an initial-def is just before the basic block, and is denoted by (0). The *origin point* of a complete-def or a partial-def is the statement in which it receives its value. The *origin point* of a clone set is the origin point of the root def-value of the clone set.

In Figure 2, the origin point of clone set $\{A^0, B^1, C^2, D^3\}$ is (0), of $\{B^4\}$ is (4), of $\{R^0\}$ is (0), and of $\{R^5\}$ is (5). In general, we note that the origin point of a given clone set is either just before the basic block, or is an ineligible statement. Consequently, the ordering of the origin points of the clone sets in a given basic block cannot be altered by any transformation permitted by Assumption 8.

Definition 3.2. The *demand point* of a persistent def-value in a basic block is just after the basic block, and is denoted by the statement number one greater than the statement number of the last statement in the basic block. The *demand point* of a nonpersistent def-value is the last ineligible statement that contains a use of the def-value, and is defined to be (0) if there is no ineligible statement that uses it. The *demand point* of a clone set is the maximum demand point of the def-values in the clone set.

For instance, in Figure 2, the demand point of $\{A^0, B^1, C^2, D^3\}$ is (5), of $\{B^4\}$ is (5), of $\{R^0\}$ is (0), and of $\{R^5\}$ is (6).

Next, we formalize the concept of an *essential node* of a version tree. Informally, an essential node is a node whose value is required *after* the values of all its child nodes (if any) have been produced. In order to satisfy this requirement, the materialization used to hold this value cannot be the same as any materialization used to hold a transient def-value that is modified to produce the value of a child node.

Definition 3.3. A node of a version tree is an *essential node* if it is either a leaf node, or a non-leaf node whose demand point is greater than the maximum origin point of its children.

For instance, in Figure 2, node $\{A^0, B^1, C^2, D^3\}$ is essential because its demand point (5) exceeds the origin point (4) of its child $\{B^4\}$. Node$\{R^0\}$ is not essential because its demand point (0) does not exceed the origin point (5) of its child $\{R^5\}$. Nodes $\{B^4\}$ and $\{R^5\}$ are leaf nodes, and so are essential.

PROPOSITION 3.1. *Every persistent node of a version tree is essential.*

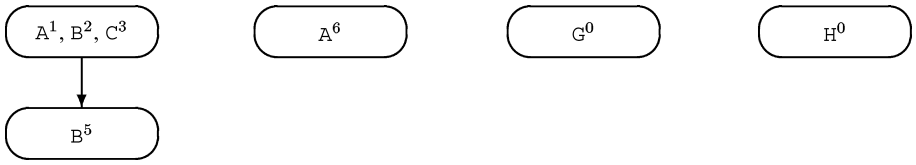

Fig. 3. Version forest for Example 3.1.

PROOF. Every leaf node is essential by definition. A persistent non-leaf node is essential because its demand point is just after the basic block, but the origin point of each of its children is within the basic block. □

Recall that there are three materializations in the optimized code for Example 2.1. There is the materialization of A^0, which is also used for C^2 and D^3. There is a materialization of transient def-value B^1, which is subsequently modified by the partial-def that creates B^4. Finally, there is a materialization of transient def-value R^0, which is subsequently modified by the partial-def that creates R^5. In this example, the number of materializations in the optimized code equals the number of essential nodes in the version forest. Each of the three essential nodes of the version forest (shown in Figure 2) for this basic block can be associated with one of these materializations. Node $\{A^0, B^1, C^2, D^3\}$ is associated with materialization A^0. Node $\{B^4\}$ is associated with materialization B^1 (via the partial-def to B^4). Node $\{R^5\}$ is associated with materialization R^0 (via the partial-def to R^5). Nonessential node $\{R^0\}$ is associated with the same materialization (R^0) as is associated with its child node, so that the partial-def R^5 that creates the child node can modify the variable R associated with the parent node.

Now consider the following example, which illustrates changing the destination variable of a complete-def, so that a nonessential parent node utilizes the same variable as its child node.

Example 3.1

```
(1)     A = G + H
(2)     B = cshift(A,shift=+5,dim=1)
(3)     C = cshift(A,shift=+3,dim=1)
(4)     x = C(i) + 3
(5)     B(i) = 50
(6)     A = G - H
```

At end of basic block: A and B live; C, G and H dead.

The version forest for this basic block is shown in Figure 3.

The basic block can be optimized by merging materializations A^1, B^2, and C^3 into a single materialization stored in B's official location, as shown below. (Thus, the references to A, B, and C in statements (1) through (5) are all replaced by references to B.)

```
(1′)      B = (G + H)<+5>
(4)       x = B<−2>(i) + 3
(5)       B(i) = 50
(6)       A = G - H
```

The complete-def A^1 is transformed into a complete-def to variable B, since this permits the version tree with root def-value A^1 to be evaluated using only one materialization. The materialization is in variable B, rather than variable A, because def-value B^5 is persistent. Def-value A^6 is persistent, and so uses the official location of A.

Note that the version forest contains four essential nodes, $\{B^5\}$, $\{A^6\}$, $\{G^0\}$, and $\{H^0\}$; and the optimized code uses four materializations. In the optimized code, essential node $\{B^5\}$ utilizes variable B, and is associated with new materialization $B^{1'}$ (via the partial-def to B^5). Essential node $\{A^6\}$ is associated with materialization A^6. Nonessential node $\{A^1, B^2, C^3\}$ is associated with the same materialization ($B^{1'}$) as is associated with its child node, so that the partial-def B^5 that creates the child node can modify the variable B utilized by the parent node. The reference in statement (4) to def-value C^3 from node $\{A^1, B^2, C^3\}$ is replaced by a reference to variable B, which holds def-value $B^{1'}$.

To clarify the relationship between def-values and materializations, we define the following function from the def-values in a basic block to the materializations in the basic block.

Definition 3.4. We say that a given def-value *overlays* a given materialization if the def-value is obtained from the materialization via a sequence of zero or more partial-defs to the same variable, with no intervening complete assignments to that variable. An *overlay set* is the set of all def-values that overlay a given materialization.

In the given basic block of Example 3.1, materialization B^2 is overlayed by def-values B^2 and B^5. In the optimized code, materialization $B^{1'}$ is overlayed by def-values $B^{1'}$ and B^5.

We make the following observations about overlay sets.

Observation 3.1. The def-values in a given overlay set have the same variable name, and the version forest nodes containing these def-values form a directed path in the version forest. The def-values in the overlay set can be envisioned as being ordered by their position along this path.

Observation 3.2. Each overlay set contains exactly one def-value that is not transient, namely the final def-value in the overlay set.

Observation 3.3. The number of materializations in a basic block equals the number of overlay sets, and consequently by Observation 3.2, equals the number of def-values in the basic block that are not transient.

THEOREM 3.1. *Every essential node contains at least one def-value that is not transient.*

PROOF. Consider a given essential node v. If v is a leaf node, then every def-value in v is not transient. If v is a nonleaf node containing a persistent def-values, then that def-values in v is not transient. Otherwise, v is a nonleaf node containing no persistent def-values. Since v is not a leaf node, it has at least one child. Since v is essential, the demand point of v exceeds the maximum origin point of its children. Since v is not persistent, the demand point of v is

Fig. 4. (a) Version forest for Example 3.2. (b) Version forest for optimized code.

an ineligible statement. This demand point statement contains a reference to at least one def-value in v. In the list of ineligible statements, this demand point statement occurs after all the statements containing a partial-def that modifies a def-value in v. Consequently, each def-value in v that is referenced in the demand point statement of v is not transient. □

The following Corollary relates the number of materializations in the code for a basic block B to the number of essential nodes in the version forest for B.

COROLLARY 3.1. *The number of materializations in a basic block is at least the number of essential nodes in its version forest.*

PROOF. From Observation 3.3 and Theorem 3.1 □

The following example illustrates how the freedom to move eligible statements, as permitted by Assumption 8 (Restricted Rearrangement Assumption), can be exploited to reduce the number of materializations.

Example 3.2

```
(1)     B = cshift(A,shift=+5,dim=1)
(2)     A(i) = 120
(3)     x = B(j) + 3
(4)     C = cshift(A,shift=+2,dim=1)
```

At end of basic block: `C` live; `A` and `B` dead.

This basic block uses three materializations: $\mathtt{A}^0$ (which is overlayed by def-values $\mathtt{A}^0$ and $\mathtt{A}^2$), $\mathtt{B}^1$, and $\mathtt{C}^4$. The version forest for this basic block is shown in Figure 4(a). Note that both nodes of the version forest are essential. The basic block can be optimized by moving the complete-def $\mathtt{C}^4$, which is an eligible statement, forward so that it occurs before statement (2), and letting the partial-def in statement (2) modify variable `C`. The optimized code is shown below, where old statement (4) is relabeled (4′) and is now the first statement in the basic block. In addition, statements (2) and (3) are transformed into statements (2′) and (3′), respectively, with appropriate annotations on the array variables used.

```
(4′)        C = cshift(A,shift=+2,dim=1)
(2′)        C<−2>(i) = 120
(3′)        x = A<+5>(j) + 3
```

The version forest for the optimized code is shown in Figure 4(b). Both nodes are essential. The optimized code contains two materializations: $\mathtt{A}^0$ (which is

overlayed only by partial-def A^0) and $C^{4'}$ (which is overlayed by partial-defs $C^{4'}$ and $C^{2'}$). The optimization is made possible by changing the reference to B^1 in the demand point statement (3) to be a reference to its clone A^0. The optimized code can be envisioned as utilizing materialization A^0 for node $\{A^0, B^1\}$, and utilizing, via partial-def $C^{2'}$, new materialization $C^{4'}$ for node $\{A^2, C^4\}$.

We now relate the version forests of a given basic block and an equivalent basic block, where the pair of basic blocks satisfy the *Materialization Problem Assumptions* given in Section 2.2.

THEOREM 3.2. *Let B be a basic block, and let V be its version forest. Let B' be an equivalent basic block, obtained from B subject to the constraints imposed by the* Materialization Problem Assumptions *given in Section* 2.2. *Let V' denote the version forest corresponding to B'. Then:*

(1) *The sequence of ineligible statements is identical in B and B', with the possible exception of the names of the array variables occurring in these statements.*
(2) *There is a one-to-one correspondence between the clone sets of B and the clone sets of B', based on a one-to-one correspondence between the root def-values of these clone sets. Namely, two clone set root def-values correspond if they are the same initial-def or are def-values created in corresponding ineligible statements.*
(3) *Version forests V and V' are isomorphic under the correspondence of version tree nodes based on the one-to-one correspondence between clone sets.*
(4) *V and V' have the same set of persistent def-values.*
(5) *The origin points of corresponding nodes of V and V' are the same (corresponding ineligible statements, or corresponding initial values).*
(6) *The demand points of corresponding nodes of V and V' are the same (just after the basic block, corresponding ineligible statements, or (0)).*
(7) *V and V' have the same set of essential nodes, that is, a node of V is essential iff the corresponding node of V' is essential.*

PROOF. Because of Assumptions 2 and 8, the sequence of ineligible statements is identical in B and B', with the possible exception of the names of the array variables occurring in these statements. The root def-value of a given clone set is either an initial-def or a def-value created by an ineligible statement. Consequently, there is a one-to-one correspondence between the nodes (each of which corresponds to a clone set) of version forests V and V'. Moreover, corresponding nodes have the same origin point. Since each version forest edge is created by an ineligible statement containing a partial-def, V and V' have corresponding edges, with corresponding endpoints. Because of Assumption 7, V and V' have the same set of persistent def-values. Corresponding uses of an array variable in corresponding ineligible statements in B and B' are uses of def-values from corresponding clone sets. Thus, for corresponding clone sets, the set of ineligible statements that contain uses of def-values from these clone sets is the same. Consequently, the demand points of corresponding nodes of V and V' are the same. Since V and V' have corresponding nodes and edges, and

corresponding nodes have the same origin point and demand points, a node of V is essential iff the corresponding node of V' is essential. □

THEOREM 3.3. *For the Materialization Problem Under No Ineligible Statement Rearrangement, the number of materializations needed to evaluate a given basic block is at least the number of essential nodes in its version forest.*

PROOF. Let B denote the given basic block, and V denote its version forest. Let B' denote an optimized evaluation of the basic block, where the optimization is constrained by the assumptions given in Section 2.2. Let V' denote the version forest corresponding to B'.

From Corollary 3.1, the number of materializations in B' is at least the number of essential nodes in V'. From Theorem 3.2, V and V' have the same number of essential nodes. Therefore, the number of materializations in B' is at least the number of essential nodes in V. □

3.3 A Lower Bound Addressing Persistence

We now develop a stronger version of Theorem 3.3, that takes into account more properties of a version forest.

Definition 3.5. The *excess persistence* of a clone set is zero if the clone set contains at most one persistent def-value, and otherwise is one less than the number of persistent def-values in the clone set. The *excess persistence* of a version forest is the sum of the excess persistence of its clone sets.

Definition 3.6. A given version tree has a *wasted root* if every essential node of the version tree is persistent, its root def-value is an initial def-value, and the variable of the root def-value does not have a persistent def-value in this version tree.

We now generalize Corollary 3.1, as a step toward generalizing Theorem 3.3.

THEOREM 3.4. *The number of materializations in a basic block is at least the number of essential nodes in its version forest, plus the excess persistence, plus the number of version trees with a wasted root.*

PROOF. From Observation 3.3, each essential node contains at least one def-value that is not transient.

Consider an essential node v that contains at least one persistent def-value. The number of persistent def-value in v is one plus the excess persistence of v. Each of these persistent def-values is not transient, so the number of def-values in the node that are not transient is at least one plus the excess persistence of the node. Thus, the number of nontransient def-values in a given version tree is at least the number of essential nodes plus the excess persistence in that version tree.

Now consider a version tree T with a wasted root. Since T has a wasted root, every essential node in T is persistent. Moreover, the root def-value of T is an initial def-value, and the variable of this root def-value is different than the variables of all the persistent def-values in T. Consequently, the overlay set containing the initial def-value is distinct from the overlay sets containing the

persistent def-values in T. The final def-value in the overlay set containing the initial def-value is not transient and is distinct from all the persistent def-values in T. Since this holds for the root def-value of each version tree with a wasted root, the number of def-values in the basic block that are not transient is at least the number of essential nodes in its version forest, plus the excess persistence, plus the number of version trees with a wasted root. From Observation 3.3, the number of materializations in the basic block is at least this number. □

We now present two additional implications of the assumptions of Theorem 3.2.

THEOREM 3.5. *Let B, V, B', and V' be as described in the statement of Theorem* 3.2. *Then, in addition to conclusions* (1) *through* (7) *of Theorem* 3.2, *the following hold:*

(8) *V and V' have the same excess persistence.*
(9) *A tree T of V has a wasted root if and only if the corresponding tree T' of V' has a wasted root.*

PROOF. From Theorem 3.2 (1), V and V' have the same set of persistent def-values, so they have the same excess persistence.

From Theorem 3.2 (2) and (3), a version tree T of V has an initial def-value as its root def-value if and only if the corresponding tree T' of V' has the same initial def-value as its root def-value. Suppose the root def-value of T is an initial def-value. From Theorem 3.2 (7), V and V' have the same set of essential nodes. From Theorem 3.2 (4), they have the same set of persistent def-values. Consequently, T has a wasted root if and only if T' has a wasted root. □

We now state a stronger version of Theorem 3.3.

THEOREM 3.6. *For the Materialization Problem Under No Ineligible Statement Rearrangement, the number of materializations needed to evaluate a given basic block is at least the number of essential nodes in its version forest, plus the excess persistence, plus the number of version trees with a wasted root.*

PROOF. Let B denote the given basic block, and V denote its version forest. Let B' denote an optimized evaluation of the basic block, where the optimization is constrained by the assumptions given in Section 2.2. Let V' denote the version forest corresponding to B'.

From Theorem 3.4, the number of materializations in B' is at least the number of essential nodes in its version forest V', plus the excess persistence, plus the number of version trees with a wasted root. From Theorems 3.2 and 3.5, V and V' have the same number of essential nodes, the same excess persistence, and the same number of version trees with a wasted root. Therefore, the number of materializations in B' is at least the number of essential nodes in V, plus the excess persistence in V, plus the number of version trees in V with a wasted root. □

3.4 Algorithm: No Persistence

We next show that for a version forest with no persistent nodes, the lower bound of Theorem 3.3 has a matching upper bound. As part of the proof, we provide an algorithm for producing the optimized code. This algorithm is subsumed by the algorithm in Theorem 3.9, but is presented here as a useful intermediate step towards developing the more general algorithm, and aids in understanding the latter algorithm.

THEOREM 3.7. *A version forest with no persistent nodes can be computed using one materialization for each essential node, and no other materializations.*

PROOF. We describe an algorithm with input a basic block B and its version forest V. It produces as output optimized code B'. The version forest V' for B' will be isomorphic to V. This algorithm associates a variable name with each version forest node. We call this variable name the *utilization-variable* of the node. In the code produced as output by the algorithm, the root def-value of the node's clone set will be stored in the node's utilization-variable. Moreover, all the uses of def-values from the node will reference the node's utilization-variable. In the portion of the algorithm that determines the utilization-variable of each node (Steps (1) and (2)), we envision the children of each node being ordered from left to right in terms of increasing value of the child's origin point. (Consequently, the node's rightmost child is the child with the maximum origin point.) The algorithm consists of the following seven steps.

Step (1). Associate a *unique* utilization-variable with each essential node, as follows.

(a) For each version tree whose root def-value is an initial-def, set the utilization-variable for the first essential node on the path from the root of the version tree to the rightmost leaf to be the variable of the initial-def.
(b) For each remaining essential node in the version forest, set the utilization-variable for that node to be a unique new temporary variable.

Step (2). Determine the utilization-variables of nonessential nodes as follows. Each tree in the version forest is processed in a bottom-up manner. Note that for each version tree, the essential nodes (which, from Definition 3.3, include all the leaf nodes) have all been assigned an associated utilization-variable in Step (1). In the bottom-up processing of each version tree, whenever a nonessential node is encountered, set the utilization-variable of the nonessential node to be the same variable as the utilization-variable of its rightmost child. (By construction, this is the child with the maximum origin point.)

Step (3). In the output code B' for the basic block, place the ineligible statements, in the same order as in the given input basic block.

Step (4). In the output code B', insert a materialized shuffle statement for each version forest nonroot node whose utilization-variable is different from the utilization-variable of its parent node. This shuffle statement is inserted just after the origin point of the parent node. Delete all other eligible statements.

Step (5). For each shuffle statement included in Step (4), set the source of the shuffle statement to be the utilization-variable of the parent node, and set the

destination of the shuffle statement to be the utilization-variable of the child node.

Step (6). (Note that in the given basic block, each assignment to a complete array is the origin point of a unique root node in the version forest, and each partial-def assignment is the origin point of a unique nonroot node in the version forest.) For each complete-def and each partial-def statement, if the utilization-variable of the corresponding version forest node is different than the left side array variable in the statement, change the left side array variable to be the utilization-variable, with appropriate shift annotation as needed.

Step (7). In the ineligible statements, replace each use of a def-val from the version forest by a reference to the utilization-variable of the version forest node containing the def-value, with appropriate shift annotation as needed.

Consider the code produced by the above algorithm. In the evaluation of the version forest, the number of variable names used equals the number of essential nodes, and the number of materializations equals the number of variable names used.

We note that in Step (1)(b), instead of using a new temporary variable, one of the variable names occurring in the node's clone set can be used, subject to the constraint that each variable name is associated with at most one essential node. This can lead to more natural-looking code, and was done in the examples.

We also note that in the application to stencil computations involving distributed arrays, Step (4) could also insert appropriate `overlap_cshift` operations into the output code for def-alues that are not materialized, but for which some shifted array elements are needed at adjacent processors. □

Consider the optimized code for Example 2.1, with the version forest shown in Figure 2. The root def-values of each of the two version trees is an initial-def: A^0 and R^0, respectively. Because of initial-def A^0, Step (1)(a) sets the utilization-variable of essential node $\{A^0, B^1, C^2, D^3\}$ to be the variable A, overlaying materialization A^0. Because of initial-def R^0, Step (1)(a) sets the utilization-variable of essential node $\{R^5\}$ to be the variable R, overlaying materialization R^0. Step (1)(b) applies to essential node $\{B^4\}$. In the optimized code for the example, instead of using a new temporary variable, the utilization-variable of this node was set to be the variable B, requiring a materialization in this variable. Step (2) set the utilization-variable of nonessential node $\{R^0\}$ to be R, the utilization-variable of its rightmost child node. (Note that the root def-values of both nodes overlay materialization R^0.) Step (3) places the ineligible statements (4) and (5) into the output code. Step (4) inserts a shuffle statement that creates materialization B^1. Note that root def-value B^4 of node $\{B^4\}$ is obtained by a partial-def to def-value B^1 and overlays materialization B^1.

In Example 2.2, the optimized code for version forest node $\{A^1, B^2\}$ has utilization-variable B, and root def-value $B^{1'}$. In the optimized code for Example 3.1, nodes $\{A^1, B^2, C^3\}$ and $\{B^5\}$ both have utilization-variable B, and their root def-values $B^{1'}$ and B^5 both overlay materialization B^5. Node $\{A^6\}$ has utilization-variable A, and its root def-value A^6 overlays materialization A^6. In the optimized code for Example 3.2, node $\{A^0, B^1\}$ has utilization-variable A, and its root

def-value A^0 overlays materialization A^0. Node $\{A^2, C^4\}$ has utilization-variable C, and its root def-value $C^{2'}$ overlays materialization $C^{4'}$.

THEOREM 3.8. *For an instance of the Materialization Problem Under No Ineligible Statement Rearrangement where the version forest of the given basic block has no persistent nodes, the number of materializations needed is exactly the number of essential nodes of the version forest.*

PROOF. Immediate from Theorems 3.3 and 3.7. □

3.5 Algorithms: Persistence

We note that it is not always possible to independently optimize each version tree, because of possible interactions between the initial and persistent def-values of the same variable. This interaction is captured by the concept of "persistence conflict," as defined below.

Definition 3.7. A *persistence conflict* in a basic block is a variable that is live on both entry to and exit from the basic block, such that either the initial def-value and persistent def-value for the variable occur in different version trees and the version tree containing the variable's initial def-value does not have a wasted root, or the initial def-value and persistent def-value for the variable occur in the same version tree and the chain of def-values from the initial def-value to this persistent def-value includes a shuffle operation. A given version tree has a *conflicted root* if its root def-value is an initial def-value that is part of a persistence conflict.

Persistence conflicts may prevent all the version trees for a given basic block from being evaluated with a minimum number of materializations, as illustrated by the following example.

Example 3.3.

```
(1)     B = cshift(A,shift=+5,dim=1)
(2)     A = G + H
(3)     B(i) = 50
(4)     x = B(j) + 3
```

At end of basic block: A live, B dead.

The version forest for this basic block is shown in Figure 5. The only essential nodes are $\{B^3\}$ and $\{A^2\}$. Each of the two version trees can by itself be evaluated using only a single materialization, corresponding to def-values A^0 and A^2, respectively. However, there is a persistence conflict involving variable A. Assumption 8 prevents ineligible statements (2), (3), and (4) from being reordered, so in this example, three materializations are necessary.

We next generalize Theorems 3.7 and 3.8 by showing that for a version forest with no persistence conflicts, the lower bound of Theorem 3.6 has a matching upper bound. As part of the proof, we provide an algorithm for producing the optimized code.

THEOREM 3.9. *A version forest with no persistence conflicts can be computed using a number of materializations equal to the number of essential nodes in*

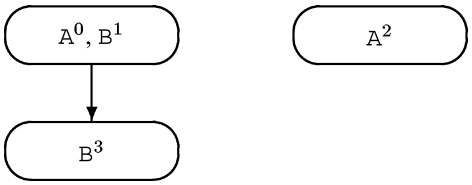

Fig. 5. Version forest for Example 3.3.

the version forest, plus the excess persistence, plus the number of version trees with a wasted root.

PROOF. We describe an algorithm with input a basic block B and its version forest V. It produces as output optimized code B'. The version forest V' for B' will be isomorphic to V. The algorithm given in the proof of Theorem 3.7 is modified as follows. The algorithm now associates two variable names (not necessarily distinct) with each version forest node. We call these two variable names the *origin-variable* and the *utilization-variable* of the node. In the code produced by the algorithm, the root def-value of the node's clone set either will be an initial-def in the origin-variable, or will be created by an assignment to the origin-variable. If the utilization-variable and origin-variable are distinct, a clone of this def-value will be stored in the utilization-variable. All the uses of def-values from the node will reference the node's utilization-variable. The algorithm consists of the following nine steps.

Step (1). Associate a *unique* utilization-variable with each essential node, as follows: (Substeps (a) and (b) are for nonpersistent essential nodes. Substeps (c) and (d) are for persistent essential nodes, and ensure that the utilization-variable for such a node is the variable from one of the persistent def-values in the node's clone set.)

(a) For each version tree such that its root def-value is an initial-def, the variable of this initial-def does not occur in any persistent def-value, and the version tree contains at least one nonpersistent essential node, arbitrarily select one of these nonpersistent essential nodes in the version tree. Set the utilization-variable for the selected nonpersistent essential node to be the variable of the initial-def.

(b) For each remaining nonpersistent essential node in the version forest, set the utilization-variable for that node to be a unique new temporary variable.

(c) For each version tree such that its root def-value is an initial-def, and the variable of this initial-def is the same as the variable of a persistent def-value in the version tree, set the utilization-variable for the version tree node containing this persistent def-value to be the variable of the initial-def. (Note that no two persistent def-values are in the same variable, so that for a given initial-def, there is at most one one persistent def-value in its variable.)

(d) For each remaining persistent essential node in the version forest, arbitrarily choose the variable of a persistent def-value in the node's clone set, and set the utilization-variable for the node to be this variable.

Step (2). Same as Step (2) in the algorithm of Theorem 3.7.

Step (3). Associate an origin-variable with each node of the version forest, as follows.

(a) For each version tree whose root def-value is an initial-def, set the origin-variable of the root node of the version tree to be the variable of this initial-def. Moreover, if this variable is the utilization-variable of any essential node in the version tree, then for all nodes along the path from the root node to this essential node, set the origin-variable for nodes along this path to be this variable.
(b) For each remaining node in the version forest, set the origin-variable of the node to be the same as the utilization-variable of the node.

Step (4). In the output code B' for the basic block, place the ineligible statements, in the same order as in the given input basic block. (Same as Step (3) in the algorithm of Theorem 3.7.)

Step (5). For each node of version forest V', construct the following clone set:

(a) A def-value whose variable is the origin-variable of the node.
(b) If the utilization-variable of the node is different than the origin variable, then a def-value whose variable is the utilization-variable.
(c) If the node's excess persistence is nonzero, then a def-value for each of the node's persistent def-values whose variable is not the node's utilization-variable.
(d) For each child of the node whose origin-variable is neither the origin-variable nor the utilization-variable of the node, a def-value whose variable is the origin-variable of this child node. (This def-value will be transient, and will be modified by the partial-def statement that creates the root def-value of the child node.)

Step (6). Include a materialized shuffle statement for each member of a clone set whose variable is different than the origin-variable of the clone set. (These are clone set members from Step 5(b), (d) and (d).) Place this shuffle statement just after the origin point of the clone set. Delete all other eligible statements for the version tree.[5]

Step (7). For each shuffle statement included in Step (6), set the source of the shuffle statement to be the origin-variable of the clone set, and set the destination of the shuffle statement to be the variable of the clone set member.

Step (8). Same as Step (6) in the algorithm of Theorem 3.7, except that the origin-variable is used in the left side of the statement at the origin point of the node.

Step (9). Same as Step (7) in the algorithm of Theorem 3.7.

Consider the code produced by the above algorithm. In the evaluation of the version tree, each materialization uses a distinct variable name, so the

[5]However, a `cshift` operation involving distributed arrays is replaced by an `overlap_cshift` operation placed just after the origin point of the parent node.

number of materializations equals the number of variable names used for array variables. Step (1) entails a unique variable (the utilization-variable) for each essential node. Step (5)(c) entails additional variables, and the total number of these additional variables equals the excess persistence of the version forest. The only additional names for array variables in B' are variables from initial-defs, such that the variable is not the utilization-variable of any essential node. Each such initial-def is the root def-value of a version tree with a wasted root. Thus, the number of variable names used for array variables equals the number of essential nodes in the version forest, plus the excess persistence, plus the number of version trees with a wasted root.

In Step (1)(b), instead of using a new temporary variable, one of the variable names occurring in the node's clone set can be used, subject to the constraints that the variable of a persistent def-value is not chosen, and that at the end of Step (1), each variable name is the utilization-variable of most one essential node. □

THEOREM 3.10. *For an instance of the Materialization Problem Under No Ineligible Statement Rearrangement where the given basic block has no persistence conflicts, the number of materializations needed is exactly the number of essential nodes of the version forest, plus the excess persistence, plus the number of version trees with a wasted root.*

PROOF. Immediate from Theorems 3.6 and 3.9. □

The following result addresses the case when persistence conflicts are present in a basic block.

THEOREM 3.11. *For the Materialization Problem Under No Ineligible Statement Rearrangement, the number of materializations required to evaluate a basic block is at least the sum of the number of essential nodes in the version forest, plus the excess persistence, plus the number of version trees with a wasted root, and is at most this number plus the number of persistence conflicts.*

PROOF. Each persistence conflict can be eliminated by introducing an extra materialization for the variable involved. This extra materialization is to a new temporary variable that is a clone of the conflicted root. References to the conflicted root are replaced by references to this materialized clone. □

4. MINIMIZING MATERIALIZATIONS UNDER UNCONSTRAINED STATEMENT REARRANGEMENT

If arbitrary rearrangement of the statements in a basic block is permitted, then interaction between version trees can occur even when there are no persistence conflicts. Indeed, we show that when unconstrained rearrangements of statements is permitted, the problem of determining if a given basic block can be implemented with at most K materializations is NP-complete. Moreover, it is NP-complete even when restricted to a basic block whose version forest has no persistent nodes.

For an example of how the freedom to rearrange statements can be used to eliminate materializations, consider the following basic block.

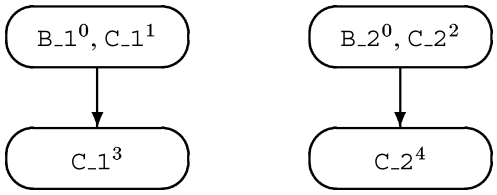

Fig. 6. Version forest for Example 4.1.

Example 4.1

```
(1)     C_1 = cshift(B_1,shift=+1)
(2)     C_2 = cshift(B_2,shift=+1)
(3)     C_1(1) = B_1(1) * 5
(4)     C_2(2) = B_2(2) * 5
(5)     x_1_2 = C_1(k) + B_2(k)
(6)     x_2_1 = C_2(k) + B_1(k)
```

At end of basic block: B_1, B_2, C_1, and C_2 all dead.

The version forest for this basic block is shown in Figure 6. There are two trees in the version forest, with root def-values B_1^0 and B_2^0, respectively. Each of the four version forest nodes is essential, so under Assumption 8 (Restricted RearrangementAssumption), Theorem 3.3 applies, and four materializations are necessary.

If the ineligible statements can be rearranged, the basic block can be evaluated with three materializations. In particular, here are two alternate versions of optimized code for the basic block, each with three materializations. Each evaluation materializes the two initial def-values B_1^0 and B_2^0, and then materializes either C_1^1 or C_2^2.

```
(1)     C_1 = cshift(B_1,shift=+1)
(3)     C_1(1) = B_1(1) * 5
(5)     x_1_2 = C_1(k) + B_2(k)
(4)     B_2<+1>(2) = B_2(2) * 5
(6)     x_2_1 = B_2<+1>(k) + B_1(k)
```

and

```
(2)     C_2 = cshift(B_2,shift=+1)
(4)     C_2(2) = B_2(2) * 5
(6)     x_2_1 = C_2(k) + B_1(k)
(3)     B_1<+1>(1) = B_1(1) * 5
(5)     x_1_2 = B_1<+1>(k) + B_2(k)
```

The following reasoning shows that the basic block cannot be evaluated with only two materializations. If there were only two materializations, they would have to be the two initial def-values B_1^0 and B_2^0. Corresponding to statements (3) and (4), the evaluation of the basic block would have partial-def statements modifying variables B_1 and B_2. Statement (5) of the basic block uses the values of clone sets $\{C\_1^3\}$ and $\{B\_2^0, C\_2^2\}$, and therefore uses the new value of variable B_1 and the old value of variable B_2. Thus, in the evaluation of the basic block, statement (5) must appear after the partial-def to B_1, and before the partial-def

to B_2. Thus, placing statement (5) in the computation requires that the partial-def to B_1 precede the partial-def to B_2. Similarly, statement (6) uses the new value of variable B_2 and the old value of variable B_1, and so must appear after the partial-def to B_2, and before the partial-def to B_1. Thus, placing statement (6) in the computation requires that the partial-def to B_2 precede the partial-def to B_1. However, the partial-defs to B_1 and B_2 cannot each precede the other.

We now consider the decision problem associated with minimizing the number of materializations.

Definition 4.1. The *Materialization Decision Problem Under Unconstrained Statement Rearrangement* is the problem of given a basic block and a positive integer K, determining if the basic block can be optimized, with arbitrary rearrangements of statements permitted, so that there are at most K materializations.

THEOREM 4.1. *The Materialization Decision Problem Under Unconstrained Statement Rearrangement is* NP*-complete, even for basic blocks whose version forests have no persistent nodes.*

PROOF. The problem is in NP because the optimized code can be guessed and verified in polynomial time.

The proof of NP-hardness is via a reduction from the Feedback Vertex Set Problem. This problem is: given a directed graph $G = (V, A)$, and positive integer K, is there is subset $V' \subseteq V$ with $|V'| \leq K$ such that V' contains at least one vertex from every directed cycle in G?

Given graph G, a basic block is constructed as follows: For each vertex v_i in V, there are a pair of array variables, B_i and C_i. For each arc (v_i, v_j), there is a scalar variable $x_{i,j}$. Let n be the number of vertices and m be the number of arcs in G. The constructed basic block contains $2n + m$ statements. First, for each v_i, there is a statement of the form

$$C_i = \texttt{cshift}(B_i, \texttt{shift=+1}).$$

Next, for each v_i, there is a statement of the form

$$C_i(i) = B_i(i) * 5.$$

Finally, for each arc (v_i, v_j), there is a statement of the form

$$x_{i,j} = C_i(k) + B_j(k).$$

The B_i and C_i arrays are all dead at the end of the basic block. (Example 4.1 gives a basic block that would be constructed given a graph G with two arcs, (v_1, v_2) and (v_2, v_1).)

We claim that there exists a cardinality K feedback vertex set for G if and only if the constructed basic block can be evaluated using $n + K$ materializations.

First, let V' be a feedback vertex set of cardinality K. The basic block can be evaluated as follows. Only retain the C variables and cshift statements corresponding to the members of V'. Follow these statements by the partial-def

statements corresponding to the members of V'. Consider a topological sort of the graph with V' (and adjacent arcs) deleted. Evaluate the partial-def statements for $V - V'$ in accordance with the topological sort. However, for v_i in $V - V'$, the partial-def statement is changed to

$$B_i^{<+1>}(i) = B_i(i) * 5.$$

For each vertex v_i, the scalar assignment statements corresponding to arcs exiting v_i are placed just after the partial-def statement for either B_i or C_i. If v_i is in V', these scalar assignments are unchanged, and if v_i is in $V - V'$, the occurence of $C_i(k)$ is replaced by $B_i^{<+1>}(k)$

Now, suppose the basic block can be evaluated with $n + K$ materializations. The n initial-defs are necessary materializations. Let V' be the set of vertices corresponding to variables that are the source of any retained `cshift` operations. Consider an arc (v_i, v_j), such that v_i and v_j are both in $V - V'$. The partial-def statements to C_i and C_j must have been changed to suitably annotated partial-defs to B_i and B_j, respectively. Thus, the scalar assignment statement corresponding to (v_i, v_j) must occur after the partial-def to B_i, and before the partial-def to B_j. Thus, the graph for $V - V'$ is acyclic. □

Theorem 4.1 contrasts with the polynomial-time optimization algorithm of Theorem 3.7 . Finally, we note that basic blocks built up via monolithic statements, rather than statements over scalars, may often turn out to be short. If so, it will often be feasible to systematically search the solution space, thereby significantly moderating the practical effect of the NP-completeness of the optimization problem.

5. RELATED WORK

Since work on optimizing APL, there has been extensive research on nonmaterialization of array-valued subexpressions in evaluating array-valued *expressions*. Such nonmaterialization in expressions is done in APL [Abrams 1970; Budd 1984; Guibas and Wyatt 1978; Hassitt and Lyon 1972], Fortran 90, HPF, ZPL [Lin and Snyder 1993], POOMA [Humphrey et al. 1997], C++ templates [Veldhuizen 1995a, 1995b], Blitz++ [Veldhuizen 1998], the Matrix Template Library (MTL) [Siek and Lumsdaine 1998], active libraries [Veldhuizen and Gannon 1998], etc. In the HPF compiler described in Gupta et al. [1995], the scalarizer for monolithic statements in-lines Fortran 90 intrinsic functions. This enables the subsequent elimination of some array temporaries by the optimizer for the scalarized code. Gupta et al.'s [1995] compiler also does message coalescing in optimizing the communications required for the same array with different shift values occurring in the same monolithic statement. In addition, Mullin's *Psi Calculus* model [Mullin 1988, 1993] provides a uniform framework for eliminating materializations in expressions built up using operations involving array *addressing* and arbitrary *shuffle*-like operations, including generalized *reshape*, and *transpose*. Indeed, *Psi Calculus* is applicable to more general monolithic array expressions and statements than those occurring in Fortran 90.

In the rest of this section, we discuss related work that addresses, at least in part, optimizations on basic blocks. In contrast to the work presented here, the

work discussed below generally does not directly address the problem of which inter-statement intermediate arrays to materialize in the presence of multiple assignment statements targeting members of the same clone set.

Hwang et al. [1995, 1998, 2001] optimize compositions of array operations, including shuffle operations, focusing primarily on expressions. However, by using a *statement merge* mechanism as described by Ju [1992], the expression for computing an intermediate array in a basic block can be substituted into uses of the array later in the basic block. The resulting composite expression is then optimized. This approach enables the substitution of an intermediate array variable by its earlier definition in the basic block, so that the resulting composite expression can be optimized.

Codeboost [Dinesh et al. 2000] is a source-to-source rule-based optimizer targeted on PDE solvers. The rules are centered on optimizing expressions. However, it also includes some transformation rules that maintain information about the relationship between array values that are created by shift operations within a basic block. It use this information to change the shift amount in some shift operations.

Various techniques for scalarizing code, performing dependence analysis on either the unscalarized or the scalarized code, and using dependency information to optimize code are described in Allen and Kennedy [2002] and Wolfe [1996]. An important type of optimizations is *loop fusion*, and a number of authors have considered loop fusion analysis. Operating on either unscalarized or scalarized code, an optimizer can eliminate some unnecessary intermediate arrays. Examples of loop fusion analysis and optimizations include the following. Gao et al. [1992] and Lewis et al. [1998] use loop fusion and subsequent optimization of the resulting loop nest to achieve array *contraction*, where each element of an array result is computed and used immediately, without materializing the entire array at one time.[6] Bacon et al. [1994] and Manjikian and Abdelrahman [1997] consider the fusion of scalarized loop nests, with the goal of increasing parallelism and locality. A number of authors have considered loop fusion analysis of monolithic level statements in a basic block. Roth and Kennedy [1996] consider dependence analysis of statements that use arrays and array sections. Such dependency analysis enables the identification of fusable statements and facilitates scalarization. Roth [2000] considers combining the analysis for scalarization and fusion. In Roth [2000], a group of statements can be fused if they are consecutive statements in a basic block, are conformable (involve isomorphic array sections), and can be combined into a single loop nest without creating additional temporary arrays. Kennedy and McKinley [1993] consider the problem of loop fusion in a basic block, providing algorithms for certain cases, and proving that performing fusion to maximize data locality is NP-hard. Kennedy [2001] provides an efficient heuristic that given a benefit function for pairs of fusable loops, chooses groups of loops to fuse. Lewis et al.

[6]More formally, following Wolfe [1996], an array that is assigned and used in a loop with only loop dependence relations, and is not used after the loop, can be replaced by privatized scalars. Such a replacement is called *array contraction*.

[1998] consider a form of loop fusion and array contraction analysis done at the monolithic level, focused on stencil-like computations. Roth and Kennedy [1998] consider incompatibilities that can arise between fusing certain loop nests and generating efficient code for distributed memory architectures. Their article addresses loop fusion in this context.

The work cited above on loop fusion is generally applicable to statements whose arrays are conformable and whose operations are done component-wise in a similar way, so that the statements can be combined into a common loop nest. The approach can be extended to situations where the loops have similar control but may only partially overlap, for example by weighting the benefits of candidate fusable loops [Kennedy 2001]. The loop fusion approach has mainly been applied to stencil-like computations, where the statements being fused involve conformablele or nearly conformable arrays and array sections. Conformability is applicable to shift operations, but not to general shuffle operations, such as `reshape` and `transpose`. Thus, these techniques do not apply directly to arbitrary shuffle operations, such as reshape and transpose. The work in this article is orthogonal to work on loop fusion (including that on monolithic-level loop fusion), and there are potential benefits to combining both approaches in an optimizer. Indeed, it appears that these fusion techniques would not produce the code transformations based on nonmaterialization analysis, as done in Examples 2.1–3.3.

Static single assignment (SSA) form [Cytron et al. 1991; Knobe and Sarkar 1998] is a technique whereby each definition is assigned a unique variable name, the resulting code is optimized, and the code is transformed back to a form where multiple definitions can utilize the same variable name. As discussed in Cytron et al. [1991], after the SSA representation of code has been optimized, the single assignment variables can be mapped into program variables, whereby single assignment variables with nonoverlapping lifetimes can share the same program variable. This mapping can be modeled as a graph coloring problem. Our problem formulation and approach focuses on clone sets rather than on individual defs, and effectively treats references and partial-defs as applicable to clone sets. Consequently, there is interchangeability between the defs occurring in a given clone set, and considerable freedom as to which members of a clone set to materialize.

As far as we know, the optimization problem in this article has only been addressed, and only in part, in Kennedy et al. [1995], Roth [1997], and Roth et al. [1997]. We observe that their approach first converts the code to SSA form before determining which array-defs are to be nonmaterialized.

6. CONCLUSIONS

Minimizing materializations is potentially an important problem, with a number of aspects. The complexity of these aspects increases with the level of program granularity over which the minimization is to be carried out. In particular in a single expression, minimizing materializations is mainly an issue of proper array accessing. In contrast in a basic block, decisions as to possible

nonmaterializations interact, so minimizing materializations becomes a combinatorial optimization problem. The concepts of clone sets, version forests, and essential nodes introduced here model fundamental aspects of this problem. Under the assumptions listed in Section 2.2, each essential node of a version forest requires a distinct materialization. This establishes a lower bound on the number of materializations required. When there are no persistent def-values, Theorem 3.7 provides an algorithm that produces an optimum solution with one materialization per essential node of the version tree. Theorem 3.6 gives a lower bound that addresses persistent def-values. Theorem 3.9 provides an algorithm when there are persistent def-values, but no persistence conflicts. This algorithm minimizes the number of materializations in this case. Theorem 3.11 provides an algorithm and upper bound for arbitrary instances.

In Section 4, we showed that relaxing the constraint on statement rearrangement increases the interaction between version trees, and this increases the computational difficulty of the optimization problem. For practical reasons, it should be worthwhile to investigate the effect of relaxing each of the other assumptions listed in Section 2.2. Thus for example, if the Coarse Analysis Assumption is relaxed, it should be possible to increase the number of eligible statements. As a second example, which may be significant in practice, when only a relatively small number of elements of a given array have been changed by partial-defs, it should be possible to use a differential buffer to hold the new values of these array elements, without materializing an entire new copy of the array. Third, the approach of this article can be extended to apply to the `eoshift` operation (and more generally, operations with some deletions of values) by relaxing the Full Mergeability Assumption. In particular, the materialized operand of an `eoshift` operation can be used as the source of values for the result of the operation, provided a differential buffer is used to hold the values deleted by the operation.

Finally, other aspects of materialization/nonmaterialization merit investigation. One issue is determining the profitability of a given possible nonmaterialization. As pointed out in Hwang et al. [1995, 2001], nonmaterialization of array intermediate results can sometimes be unprofitable. In general, the profitability of a possible nonmaterialization depends on both the code and the target architecture. In addition to the possible elimination of materializations appearing in the given code, introduced materializations are potentially very important for increasing locality of reference. The judicious introduction of materializations of certain array-valued temporaries can sometimes significantly improve the efficiency of memory access at various levels of the memory hierarchy. For example, suppose there is a loop that repeatedly accesses the elements of an array in a different order than that in which they are stored. It may well pay to materialize a version of the array in which the array elements have been reordered in a way consistent with the access pattern in the loop. An appropriate shuffle operation can be used to produce a materialized temporary, and the loop can then access the array elements from this temporary.[7]

[7]More generally, there may be a choice between reordering the loop statements via retiling the loop iterations, reordering the data in the array, or some combination of both.

Detailed understanding of how and when to introduce such materializations may well prove important both for program design and mechanized program transformations.

REFERENCES

ABRAMS, P. S. 1970. An APL machine. Ph.D. dissertation. Stanford University, Stanford, CA.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, Reading, MA.

ALLEN, R. AND KENNEDY, K. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan-Kaufmann Publishers, San Francisco, CA.

BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (Dec.), 345–420.

BUDD, T. A. 1984. An APL compiler for a vector processor. *ACM Trans. Prog. Lang. Syst. 6*, 3 (July), 297–313.

CHAMBERLAIN, B. L., CHOI, S.-E., LEWIS, E. C., LIN, C., SNYDER, L., AND WEATHERSBY, W. D. 1996. Factor-join: A unique approach to compiling array languages for parallel machines. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, D. Padua, A. Nicolau, D. Gelernter, U. Banerjee, and D. Sehr, Eds. Lecture Notes in Computer Science, vol. 1239. Springer-Verlag, New York, pp. 481–500.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst. 13*, 4 (Oct.), 451–490.

DINESH, T. B., HAVERAAEN, M., AND HEERING, J. 2000. An algebraic programming style for numerical software and its optimization. *Sci. Prog. 8*, 4, 247–259.

GAO, G. R., OLSEN, R., SARKAR, V., AND THEKKATH, R. 1992. Collective loop fusion for array contraction. In *Proceedings of 5th International Workshop on Languages and Compilers for Parallel Computing* (New Haven, CT, Aug.), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 757. Springer-Verlag, pp. 281–295.

GUIBAS, L. J. AND WYATT, D. K. 1978. Compilation and delayed evaluation in APL. In *Conference Record of the 5th Annual ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages (POPL '78)* (Tucson, AZ, Jan). ACM, New York, pp. 1–8.

GUPTA, M., MIDKIFF, S., SCHONBERG, E., SESHADRI, V., SHIELDS, D., WANG, K.-Y., CHING, W.-M., AND NGO, T. 1995. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*. (San Diego, CA, Dec.). ACM, New York.

HASSITT, A. AND LYON, L. E. 1972. Efficient evaluation of array subscripts of arrays. *IBM J. Res. Devl. 16*, 1 (Jan.), 45–57.

HUMPHREY, W., KARMESIN, S., BASSETTI, F., AND REYNDERS, J. 1997. Optimization of data-parallel field expressions in the POOMA framework. In *Proceedings of the 1st International Conference on Scientific Computing in Object–Oriented Parallel Environments (ISCOPE '97)* (Marina del Rey, CA, Dec.), Y. Ishikawa, R. R. Oldehoeft, J. Reynders, and M. Tholburn, Eds. Lecture Notes in Computer Science, vol. 1343. Springer-Verlag, New York, pp. 185–194.

HWANG, G.-H., LEE, J. K., AND JU, D.-C. 1995. An array operation synthesis scheme to optimize Fortran 90 programs. *ACM SIGPLAN Notices, Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 30*, 8 (Aug.), 112–122.

HWANG, G.-H., LEE, J. K., AND JU, R. D.-C. 1998. A function-composition approach to synthesize Fortran 90 array operations. *J. Paral. Dist. Comput. 54*, 1 (Oct.), 1–47.

HWANG, G.-H., LEE, J. K., AND JU, R. D.-C. 2001. Array operation synthesis to optimize HPF programs on distributed memory machines. *J. Paral. Dist. Comput. 61*, 4 (Apr.), 467–500.

JU, D.-C. 1992. The optimization and parallelization of array language programs. Ph.D. dissertation, University of Texas at Austin, Austin.

KENNEDY, K. 2001. Fast greedy weighted fusion. *Int. J. Paral. Prog. (IJPP) 29*, 5 (Oct.), 463–491.

KENNEDY, K. AND MCKINLEY, K. S. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages*

and Compilers for Parallel Computing (Portland, OR, Aug.), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 768. Springer-Verlag, New York, pp. 301–320.

KENNEDY, K., MELLOR-CRUMMEY, J., AND ROTH, G. 1995. Optimizing Fortran 90 shift operations on distributed-memory multicomputers. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing* (Columbus, OH, Aug.). Lecture Notes in Computer Science, vol. 1033. Springer-Verlag, New York, pp. 161–175.

KNOBE, K. AND SARKAR, V. 1998. Array SSA form and its use in parallelization. In *Conference Record 25th ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages (POPL '98)* (San Diego, CA, Jan.). ACM, New York, pp. 107–120.

LEWIS, E. C., LIN, C., AND SNYDER, L. 1998. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Que., Canada, June). ACM, New York, pp. 50–59.

LIN, C. AND SNYDER, L. 1993. ZPL: An array sublanguage. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing* (Portland, OR, Aug.), U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 768. Springer-Verlag, New York, pp. 96–114.

MANJIKIAN, N. AND ABDELRAHMAN, T. S. 1997. Fusion of loops for parallelism and locality. *IEEE Trans. Paral. Dist. Syst. 8*, 2 (Feb.), 193–209.

MULLIN, L. 1993. The Psi compiler project. In *Workshop on Compilers for Parallel Computers*. TU Delft, Holland.

MULLIN, L. M. R. 1988. A mathematics of arrays. Ph.D. dissertation. Syracuse University, Syracuse, New York.

ROTH, G. 1997. Optimizing Fortran90D/HPF for distributed-memory computers. Ph.D. dissertation, Dept. of Computer Science, Rice University.

ROTH, G. 2000. Advanced scalarization of array syntax. In *Proceedings of the 9th International Compiler Construction Conference (CC '2000)* (Berlin, Germany, Mar.). Lecture Notes in Computer Science, vol. 2017. Springer-Verlag, New York, pp. 219–231.

ROTH, G. AND KENNEDY, K. 1996. Dependence analysis of Fortran90 array syntax. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96)* (Sunnyvale, CA, Aug.). CSREA Press, pp. 1225–1235.

ROTH, G. AND KENNEDY, K. 1998. Loop fusion in high-performance Fortran. In *Proceedings of the 12th International Conference on Supercomputing (ICS '98)* (Melbourne, Australia, July). ACM, New York, pp. 125–132.

ROTH, G., MELLOR-CRUMMEY, J., KENNEDY, K., AND BRICKNER, R. G. 1997. Compiling stencils in high performance Fortran. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (SC '97): High Performance Networking and Computing* (San Jose, CA, Nov.). ACM, New York.

SCHWARTZ, J. T. 1975. Optimization of very high level languages—I. Value transmission and its corollaries. *Comput. Lang. 1*, 2 (June), 161–194.

SIEK, J. G. AND LUMSDAINE, A. 1998. The matrix template library: A generic programming approach to high-performance numerical linear algebra. In *Proceedings of the 2nd International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE '98)* (Santa Fe, NM, Dec.), D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds. Lecture Notes in Computer Science, vol. 1505. Springer-Verlag, New York, pp. 59–70.

VELDHUIZEN, T. 1995a. Using C++ template metaprograms. *C++ Report 7*, 4 (May), 36–43. (Reprinted in *C++ Gems: Programming Pearls from the C++ Report*, S. R. Lippman, Ed. Cambridge University Press, Cambridge, UK, pp. 459–474.)

VELDHUIZEN, T. L. 1995b. Expression templates. *C++ Report 7*, 5 (June), 26–31. (Reprinted in *C++ Gems: Programming Pearls from the C++ Report*, S. S. Lippman, Ed. Cambridge University Press, Cambridge, UK, pp. 459–474.)

VELDHUIZEN, T. L. 1998. Arrays in Blitz++. In *Proceedings of the 2nd International Symposium on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE '98)* (Santa Fe, NM. Dec.). D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds. Lecture Notes in Computer Science, vol. 1505. Springer-Verlag, New York, pp. 223–230.

VELDHUIZEN, T. L. AND GANNON, D. 1998. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO '98)* (Yorktown Heights, NY.). SIAM, Philadelphia, PA.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA.

Received May 2004; revised July 2005; accepted October 2005