Rutgers University Department of Computer Science Technical Report 386
Submitted 24 Jun 1999

Constant Propagation Through Array Variables *

David Wonnacott
Haverford College, Haverford, PA 19041 U.S.A.
davew@cs.haverford.edu
(610) 896-4973; (610) 896-4904 (FAX)

Revised to March 18, 1999

Abstract

Constant propagation can provide significant improvements in program speed, both by directly enabling optimizations such as constant folding and algebraic simplification and by providing information that increases the effectiveness of other compile-time analyses. This technique has generally not been applied to arrays, as it relies on information about the flow of values, but standard array analysis produces only memory aliasing information.

The development of techniques for analyzing the flow of values in array variables raises the possibility of applying constant propagation (as well as other traditional value-based scalar analyses and optimizations) to values in arrays. In the case of constant propagation, analysis may benefit from an extension of our definition of what constitutes a constant. In this work, we propose a definition of constant array that subsumes those used for scalars and in previous work on constant propagation for arrays. We then show that this definition lets us perform analysis of arrays that fall outside the domain of previous work on constant propagation. We also discuss generalizations of dead code elimination and forward substitution.

Keywords: constant propagation, dead code elimination, forward substitution, static analysis, value-based array dependences

*This is supported by NSF grant CCR-9808694

```
      REAL*8 A(0:3)
      . . .
      A(0) = -8.0D0/3.0D0
      A(1) =  0.0D0
      A(2) =  1.0D0/6.0D0
      A(3) =  1.0D0/12.0D0
```

Figure 1: Definition of Array `A` in `107.MGRID`, from [SK98]

```
      REAL*8 A(0:3)
      . . .
      DO 10 I=1,4
        A(I) = 5*I+3
10    CONTINUE
```

Figure 2: Array Definition Not Considered Constant by [SK98]

1 Introduction

Traditional scalar analysis techniques produce information about the flow of values, whereas traditional array analysis gives information only about memory aliasing. This difference has lead to the restriction of many analysis and optimization techniques, such as constant propagation [Muc97], to scalar variables. A number of techniques have recently been developed for accurately describing the flow of values in array variables [Fea91, MAL92, TP92, PW93, PW94, Tu95, BCF97, GLL97, PW98, SK98]. Of these, only the work of Sarkar and Knobe discussed the use of their analysis for constant propagation.

Constant propagation provides two kinds of benefits during optimization. It enables optimizations such as constant folding and algebraic simplification, which provide direct speedups. Additionally, it may improve the performance of other analyses, which may then perform an optimization. In some cases, this latter effect is more significant: Constant propagation can provide information about array subscript expressions, which is important for automatic parallelization, locality optimization, and the elimination of unnecessary array bounds checks.

Knobe and Sarkar require that the expression assigned to an array element meet the definition of constant that is traditionally used in scalar analysis: a single value that is known at compile time. They demonstrate the value of their algorithm in analyzing and optimizing the `107.MGRID` benchmark from the SPEC95fp benchmark set [Cor97]. Their system keeps track of the values stored in `A` (see Figure 1). Thus, it can greatly speed up the computation loop in `RESID`, in which a complicated expression is multiplied by `-A(1)`. However, the definition of constant used by Knobe and Sarkar excludes some arrays that intuitively should be labelled as constant, such as `A` in our Figure 2.

We propose to broaden the definition of constant as applied to arrays. Our definition includes the array in Figure 2, and also allows for constant arrays with symbolic sizes, arrays that are constant only under some conditions or have conditional constant values, and arrays for which only some elements are

```
      DO 10 J = 1, JMAX
         JPLUS(J) = J+1
10    CONTINUE

      IF ( .NOT. PERIDC ) THEN
         JPLUS(JMAX) = JMAX
      ELSE
         JPLUS(JMAX) = 1
      ENDIF
```

Figure 3: Definition of `JPLUS` from `ARC2D` [B+89]

constant. For example, the `ARC2D` benchmark of the perfect club benchmark set [B+89] contains the definition of `JPLUS` shown in Figure 3 (as well as the definition of a similar array `JMINU`). These arrays occur frequently in array subscripts in two of the most significant loops of the `ARC2D` program.

The rest of this paper is organized as follows: In Section 2 we review our system for analyzing the flow of values in arrays through the use of constraints on integer variables. Section 3 gives a brief algorithm for dead code elimination, and serves as a "warm-up" for Section 4, which gives our algorithm for constant propagation. Sections 5 discusses the challenges of generalizing forward substitution for arrays. In Section 6 we discuss related work, and in Section 7 we give our conclusions.

2 Constraint-Based Analysis of Arrays

Traditional array data dependence analysis provides information about memory aliasing, but not the flow of values in array variables. More recent techniques have been developed to provide this "value-based" dependence information about arrays. This section describes the representation produced by our algorithms for value-based array dependence analysis [PW93, PW94, PW98].

Most systems for analysis of scalars provide information about which definitions reach which uses. To provide accurate information about arrays, we must also describe either (a) which loop iterations of the definition and use are connected, or (b) which elements of the array carry the flow of values. The array subscript expressions provide a mapping between these two representations.

We represent a dependence as a relation between two tuples of integer variables corresponding to the loop index variables at the source and sink of the dependence. These relations are defined in terms of affine constraints on the loop index variables, a set of symbolic constants, and a set of function symbols that are used to represent nonlinear terms. For example, suppose the definition of `JPLUS` shown in Figure 3 were followed by the uses shown in Figure 4. There would be three flow dependences to the use of `JPLUS(J)` on the third line of Figure 4: one from the definition in the loop of Figure 3, and one from each of the subsequent conditional definitions. We would describe the flow of values from the loop with the relation

$$\{\ [J] \rightarrow [N, J] \mid 1, JLOW \leq J \leq JUP, JMAX - 1 \wedge 2 \leq N \leq 4\ \}$$

```
      DO 300 N = 2,4
         DO 210 J = JLOW,JUP
            JP1 = JPLUS(J)
            JP2 = JPLUS(JP1)
            DO 212 K = KLOW,KUP
C              ...
               C4 = COEF4(J,K)*DTD
C              ...
               WORK(J,K,4) = -(C2 + 3.*C4 + C4M)*XYJ(JP1,K)
               WORK(J,K,5) = XYJ(JP2,K)*C4
212         CONTINUE
210      CONTINUE
C
       IF(.NOT.PERIDC  )THEN
         J = JLOW
         JP1 = JPLUS(J)
         JP2 = JPLUS(JP1)
         DO 220 K = KLOW,KUP
C           ...
            WORK(J,K,4) = -(C2 + 3.*C4 + C4M)*XYJ(JP1,K)
            WORK(J,K,5) = XYJ(JP2,K)*C4
220      CONTINUE
C      ...
C      (uses of WORK array follow)
```

Figure 4: Uses of `JPLUS` in `300.stepfx` in `ARC2D` (Simplified)

and the dependence from the first conditional assignment with

$$\{\ [] \rightarrow [N, JMAX] \mid PERIDC \wedge JLOW \leq JMAX \leq JUP \wedge 2 \leq N \leq 4\ \}.$$

In the above example, the relations describing memory aliasing and flow of values are identical for the dependence from the conditional assignment. For the dependence from the loop, the value-based flow dependence relation does not include a dependence from iteration $[JMAX]$, as `JPLUS(JMAX)` must be overwritten in the conditional. In contrast, the memory-based dependence can include a dependence from iteration $[JMAX]$.

Value-based dependence analysis is important for a number of transformations that expose parallelism, such as privatization [TP92]: If no variable carries values between loop iterations, the compiler can privatize the variables (give a separate copy to each processor) and run the loop in parallel. In the code above, value-based array dependence analysis shows that the `WORK` array does not carry any values between iterations.

We can apply a variety of operations such as union and join (composition) to our dependence relations. These operations often provide simple ways to describe program properties. For example, the composition of the value-based flow dependence from statement S_1 to another statements S_2 with the value-based flow dependence from S_2 to a third statement S_3 describes the flow of information from S_1 to S_3 via S_2.

Our use of affine constraints controls the domain of programs over which our analysis is exact. We can produce a description of intra-procedural data flow that is exact and free of function symbols for any procedure in which control flow is built only from properly nested loops, if's, and break statements, and in which all array subscripts and loop bounds and conditionals are affine functions of the enclosing loop indices and a set of symbolic constants. For such relations, the relational operations mentioned in this paper are exact unless otherwise noted (the complexity of some operations is extremely high in the worst case, but this has not been a problem for our program analysis in practice).

We can also perform analysis of programs outside of this domain as long as they have reducible control flow graphs. In this case, our memory-based dependence relations contain function symbols, and our value-based dependences may be approximate. Many of our relational operations may be forced to produce approximate results when operating on relations with function symbols. For details, refer to [PW93, PW94, Won95].

3 Dead Code Elimination

We generalize the technique of dead code elimination [Muc97] by searching not for statements that produce a value that is never used, but for *iterations* that do so. We can describe a set using the notation of Section 2, as a relation with no output variables. For example, iteration `JMAX` of the loop in Figure 3 defines of `JPLUS(JMAX)`, which is immediately killed. Thus, the set of live iterations is $\{\ [J] : 1 \leq J \leq JMAX - 1\ \}$, and the set of dead iterations is $\{\ [JMAX] : JMAX \geq 1\ \}$. We can eliminate such dead code by simply using the set of live iterations, rather than the set of all iterations, with the code generation system described in [KPR95].

```
      A(1) = 15
      DO 10 J = 2, JMAX
         A(J) = (A(J-1)*A(J-1))/10
10    CONTINUE
C     ... no further uses of A ...
```

Figure 5: Dead Code

Unfortunately, applying this definition in general is quite difficult. Scalar dead code detection can be applied iteratively, as there are a finite number of statements to be examined. In contrast, the compiler may not be able to place an upper bound on the number of data flow arcs within a loop with an upper bound, such as that shown in Figure 5. To prove that such a loop can be entirely eliminated, we must show that the transitive closure of the value-based flow dependences does not reach any live statements. Since the flow dependence graph is potentially infinite, this problem is undecidable in general [KPRS96].

We can perform approximate dead code elimination using the definition above and a safe approximation of transitive closure given in [KPRS96]: For each statement, only those iterations with a transitive value-based flow dependence to the exit of the procedure are live. This is simply the domain of the union of all transitive value-based flow dependences originating from each statement.

4 Constant Propagation

We generalize the notion of constant, as we generalized the idea of dead code, by applying it to an iteration of a statement rather than a whole statement. We say that an iteration of a statement is a constant-valued iteration if its value is a constant function of the loop index and other constants. We first give an algorithm for determining which iterations are constant-valued, and then discuss the determination of the value of each iteration.

4.1 Identification of Constant-Valued Iterations Within a Procedure

If there is no cycle among statements in the dataflow graph, we can identify the constant-valued iterations as follows: We generate a DAG in which each node is a statement, and each arc represents an inter-statement flow of values, and then perform a topological sort on this DAG. We then visit each statement, moving from defs to uses, and generate a description of the sets of iterations that are constant-valued. Those iterations that read input or have incoming data flow from a non-constant-valued iteration, are not themselves constant-valued; iterations that combine only constant values are constant-valued. For a statement S, the set of non-constant-valued iterations is defined as

$$nc(S) := \left(\bigcup_{S' \text{ s.t. } \delta^v(S',S)} range(restrictDomain(\delta^v(S',S), nc(S')) \bigcup input(S) \right.$$

```
      REAL*8 A(1:8), B(1:8)

      DO 10 I=1,6
        A(I) = 5*I+3
10    CONTINUE

      DO 20 I=1,4
        READ A(I+4)
20    CONTINUE

      DO 30 I=1,6
        B(I) = 2*A(I+1)-5
30    CONTINUE

      RETURN B(3)+B(4)
```

Figure 6: Intra-Procedural Constant Propagation Example

where $\delta^v(S', S)$ is the value-based flow dependence from S' to S and $input(S)$ is the set of all iterations that read input (typically this is all or none of the loop iterations, unless a read is guarded by an if). In other words, the set of non-constant-valued statement iterations is the set of statement iterations that can be reached by following any transitive value-based dependence from any statement iteration that reads input.

When our dependence relations are exact and free of function symbols, the calculation above produces an exact result. When a procedure contains non-affine subscripts or control flow, we can still apply the above definition, though when the result is flagged as an approximation, we must be sure to interpret it conservatively. When the inter-statement data flow graph of a procedure is cyclic, we fall back on the techniques of [KPRS96], which may also produce approximate results.

Example

Consider the application of our algorithm to the code in Figure 6. There are dependences from the definition of `B` in loop 30 to the return statement, and from each of the definitions of `A` to the definition of `B`. We therefore identify sets of constant-valued iterations for the definitions of `A` (in any order), then for the definition of `B`, and then for the returned value.

All iterations of the first definition of `A` meet our definition of constant-valued, so we identify the set of constant-valued iterations as $\{\ [I] \mid 1 \leq I \leq 6\ \}$ (or, equivalently, the set of non-constant-valued iterations is empty). No iterations of the second definition meet our definition, so the set of non-constant-valued iterations is $\{\ [I] \mid 1 \leq I \leq 4\ \}$.

No iterations of the definition of `B` read input, so our analysis is entirely focused on the incoming value-based dependences, which are

$$\{\ [I] \rightarrow [I-1] \mid 2 \leq I \leq 4\ \}$$

for the dependence from first statement and

$$\{\ [I] \rightarrow [I+3] \mid 1 \leq I \leq 3\ \}$$

for the dependence from the second. Note that our value-based dependence analysis has already eliminated the dependences from the first write that are killed by the second. When we restrict the domain of each of these dependences to the sets of non-constant-valued iterations, the first dependence becomes empty and the second remains unchanged. The ranges of each of these restricted dependences are the empty set and $\{ [I+3] \mid 1 \leq I \leq 3 \}$. The union of these two ranges, $\{ [I] \mid 4 \leq I \leq 6 \}$, gives the set of non-constant-valued iterations of loop 30.

The dependences from loop 30 to the two reads in the return are $\{ [3] \rightarrow [] \}$ and $\{ [4] \rightarrow [] \}$, so the first of these expressions is considered constant, but the second is not. Thus, the entire expression is considered non-constant, though we will be able to insert a value for `B(3)` at compile time, as we will see in Section 4.3. Note also that our dead code elimination algorithm, if applied after the insertion of the constant value for `B(3)`, will identify as dead code all iterations but number 3 of loop 30, all iterations but number 1 of loop 20, and all iterations of loop 10. (However, we cannot eliminate any iterations of loop 20, as reading of input has a side effect.)

4.2 Interprocedural Identification of Constant-Valued Iterations

We identify sets of constant-valued iterations on a procedure-by-procedure basis. We start with a pass of inter-procedural data flow analysis in which no attempt is made to distinguish among array elements (and thus standard algorithms for scalars [Muc97] may be employed). We generate a graph in which each procedure is a node, and there is an arc from $P1$ to $P2$ if $P2$ may read a value defined in $P1$. We break this graph into strongly connected components, and topologically sort the components. We process these components in order, starting with those that receive no data from any other component. Within one component, we process procedures in an arbitrary order.

Note that the above ordering may have little effect other than to identify one "initialization" procedure, and place it before all others in our analysis. We believe this will be sufficient for our purposes, however, as we expect to find many constants in such routines.

To process a procedure, we must start with information about its parameters and any global variables it uses. We provide such information as a set of array subscript tuples for which constant values have been stored. This representation can be used interchangeably with our iteration-space based representations by treating procedure entry as a sequence of loop nests, each of which defines one value per iteration for a given procedure parameter.

We collect information about parameters from all call sites, reshaping parameters as necessary (we require that all dimensions are known at compile time, in which case reshaping is a simple application of our relational operations [Won95]). If all call sites have been processed and agree on the sets of elements that are constant (and their values), we use this information. In other situations, we can either perform analysis in cases (if the results allow for different optimizations, procedure cloning [Muc97] can be used to produce optimized versions for some call sites), or safely assume all array elements are non-constant.

When analysis of a procedure is complete, we produce summary information about constant-valued elements of each array that is modified by the procedure. This information can then be propagated into procedures that have not been analyzed. It is possible to invent cases in which our analysis could be made more exact through iterative application to each procedure within a strongly connected component of the interprocedural dataflow graph, but we have not investigated the possibility of doing this in practice.

Example

Consider the application of our algorithm to subroutine `STEPFX` of the `ARC2D` (shown in Figure 4). This routine will be analyzed after subroutine `INITIA`, as data flows from `INITIA` to `STEPFX` but not the other way. `STEPFX` makes use of the `JPLUS` array, which is defined in `INITIA`, as shown in Figure 3. This definition produces summary information showing that elements $1..JMAX$ of the array are constant, though the last element's value depends on the symbolic constant $PERIDC$ (as is permitted in our definition). This information is described as the union of three sets (one for each of the definitions):

$$\begin{array}{ll} & \{\ [J] \mid 1, JLOW \leq J \leq JUP, JMAX-1\ \} \\ \cup & \{\ [JMAX] \mid PERIDC\ \} \\ \cup & \{\ [JMAX] \mid \neg PERIDC\ \} \end{array}$$

As `INITIA` is always called before `STEPFX`, and no other procedure writes to `JPLUS`, we can use the above summaries at entry to `STEPFX`.

Analysis of the `STEPFX` procedure itself includes the following steps:

The definition of `JP1` at the top of loop 210 reads a value from `JPLUS(J)`. `JPLUS` is not written in `STEPFX`, so this is treated as a data flow from the initial value found in the previous paragraph. Array elements identified as constant provide values for all iterations of this statement, so all iterations of the definition of `JP1` are identified as constant-valued.

The definition of `WORK(J,K,4)` in iteration $[N, J, K]$ receives a value from the procedure entry in `XYJ` and a value from iteration $[N, J]$ of the definition of `JP1`. It must therefore be analyzed after the definition of `JP1`. In the absence of other information, the `XYJ` array is presumed not to be constant, and all iterations of this definition of `WORK` are labelled non-constant-valued.

The definition of `JP2` reads the value of `JP1` defined in the previous statement, and uses it as an index into `JPLUS` array. In such cases, the initial execution of our value-based dependence analysis system treated the read from `JPLUS` as a read with a non-affine subscript (because it lacked information about the value of the variable `JP1`), and we can not be sure the subscript expression has a value in the range $1..JMAX$. However, we can now perform a more accurate analysis, based on the value we will determine for `JP1` with the techniques from the next section.

4.3 Expressing Constant Values

The algorithm given above identifies sets of iterations that meet our definition of constant-valued iteration. To perform constant propagation, we must also record what the values of these constants are. Recall that a constant-valued iteration has a value that is a constant function of the loop index and other constants. In principle, we could choose any representation for this function,

such as the compiler's intermediate representation for the expression. We must be able to perform several operations on our representation, including composition (as we propagate values forward), transitive closure (as we propagate forward around a dependence cycle), evaluation (to produce constants to insert in the code, such as 41 for `B(3)` in Figure 6), and extraction of information for other compile-time analysis routines.

We represent the values of constant-valued iterations with the same constraint-based relations we use for dependences. This provides a representation on which the required operations have already been implemented, and has the advantage that our information about values is already in a form that can be used in our analysis system. When the constant functions involve nonlinear operations, our relational operations produce approximate results, and we conservatively mark all iterations of the offending statement as non-constant-valued.

Example

For Figure 6, we represent the values of the iterations of loop 10 as

$$\{\ [I] \rightarrow [5 * I + 3] \mid 1 \leq I \leq 6\ \}.$$

To find values for the constant-valued iterations of loop 30 (iterations 1..3), we first compose the above relation with the inverse of the flow dependence from loop 10 to loop 30, producing a relation from iteration of loop 30 to the value read from `A`:

$$\{\ [I] \rightarrow [5 * (I + 1) + 3] \mid 1 \leq I \leq 3\ \}.$$

Finally, we compose the above with the relation for the expression giving the value produced by loop 30 in terms of the value read from the above relation, $\{\ [X] \rightarrow [2X - 5]\ \}$, producing a relation mapping iteration of loop 30 to the value of that iteration:

$$\{\ [I] \rightarrow [10 * I + 11] \mid 1 \leq I \leq 3\ \}.$$

For the `ARC2D` benchmark, our analysis of `INITIA` produces the following descriptions mapping ranges of subscripts to values of `JPLUS`:

$$\begin{array}{ll} & \{\ [J] \rightarrow [J+1] \mid 1, JLOW \leq J \leq JUP, JMAX - 1\ \} \\ \cup & \{\ [JMAX] \rightarrow [JMAX] \mid PERIDC\ \} \\ \cup & \{\ [JMAX] \rightarrow [1] \mid \neg PERIDC\ \}. \end{array}$$

This allows us to describe the value stored in `JP1` in iteration $[N, J]$ of loop 210:

$$\begin{array}{ll} & \{\ [N,J] \rightarrow [J+1] \mid 1, JLOW \leq J \leq JUP, JMAX - 1 \wedge 2 \leq N \leq 4\ \} \\ \cup & \{\ [N, JMAX] \rightarrow [JMAX] \mid PERIDC \wedge JLOW \leq JMAX \leq JUP \\ & \qquad \wedge\ 2 \leq N \leq 4\ \} \\ \cup & \{\ [N, JMAX] \rightarrow [1] \mid \neg PERIDC \wedge JLOW \leq JMAX \leq JUP \\ & \qquad \wedge\ 2 \leq N \leq 4\ \}. \end{array}$$

When analyzing the conditional definition of `JP1` just above loop 220, we make use of the fact that `PERIDC` must be false if this code is executed, to eliminate the middle one of these three relations.

This information about the value of `JP1` cannot be used for constant folding, as we do not make use of a value from a known constant iteration (as we did

in Figure 6). However, we can use the information about values during other compile-time analysis. For example, we can use the information above in proving that run-time bounds checks are not needed for the subscripts of `XYJ`, if we are using a system that detects illegal subscripts. Additionally, we can determine that the inner loop steps through this array with unit stride, when optimizing this routine for cache performance.

Finally, we note that the routines `STEPFX` and `FILERX` both contain many uses of constants derived from `JPLUS` in inner loops. These are two of the three most time-consuming routines in the `ARC2D` benchmark, taking about one third of the total run time [EHLP91].

5 Forward Substitution

As our algorithm propagates constant functions, rather than constant scalars, we cannot always use our results for constant folding (as in the case of `JP1` in `ARC2D`). However, this does not prevent us from substituting an expression to compute the constant function. We believe our basic approach could be extended in a straightforward way to perform more general forward substitution. In this section, we give speculate about the implementation and utility of this transformation.

For the constant functions described in the previous section, forward substitution is always legal, though it may not always be beneficial. Extending our system for more general forward substitution requires a system for determining whether the expression being substituted has the same value at the point of substitution. This test a simple application of memory-based dependence information to determine which of the relevant variables may have been overwritten.

Estimating the profitability of forward substitution in our system is somewhat harder. This requires a comparison of costs of storing the value and later fetching it from memory (which depends on cache hit ratio among other things) and evaluating the expression at the point of use. Note that a naive translation of the value relations used in the previous section could introduce a large number of conditionals that would add greatly to the cost of evaluating the expression.

We have not explored these issues in detail, in part because we are not aware of realistic cases where it might be beneficial to perform forward substitution of values that are more complex than those that come under our definition of constant.

6 Related Work

There are a number of systems for performing analysis of the flow of values in array variables [Fea91, MAL92, TP92, PW93, PW94, Tu95, BCF97, GLL97, PW98]. Most of these produce parameterized descriptions of either live array regions or the flow of values through the iteration spaces of the program. In principle, either kind of information could serve as a foundation for extending the scalar analysis algorithms discussed here, though most of the above work focuses on optimizations with larger potential payoff (such as automatic parallelization), rather than constant propagation. Noteworthy exceptions are the works of Sarkar and Knobe [SK98] and Padua and Tu [TP92, Tu95].

Sarkar and Knobe describe an extension of static single assignment form for array variables, and present variation of the conditional constant propagation algorithm given by Wegman and Zadeck [WZ91]. The Wegman and Zadeck algorithm operates on the SSA representation of the program; Sarkar and Knobe's variation uses their Array SSA form. As noted in Section 1, Sarkar and Knobe use a definition of constant that is much more limited than ours. However, their iterative algorithm should be better at handling cycles in which value flow depends on constant propagation and vice versa.

Padua and Tu have developed techniques for propagating symbolic expressions describing the values stored in arrays [Tu95, Section 2.4.2]. They emphasize the use of this information for analysis of array subscripts, rather than for general constant propagation.

7 Conclusions

The availability of information about the flow of values in array variables enables the use of a variety of analysis and optimization techniques that had previously been applied only to scalars, such as dead code elimination and constant propagation. These techniques are most effective when the fundamental definitions are reexamined and updated for use with arrays and loops: dead code elimination can eliminate dead iterations of statements rather than dead statements, and constant propagation can propagate constant functions rather than constant scalar values.

We have presented algorithms for dead code elimination and constant propagation based on the above definitions. We believe these algorithms will be effective for code in which data flow is primarily acyclic, such as the propagation of constants created in initialization routines and used throughout a program. This very situation occurs in some of the most expensive routines in the `ARC2D` benchmark.

References

[B+89] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.

[BCF97] Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[Cor97] The Standard Performance Evaluation Corporation. *SPEC CPU95 Benchmarks*. 1997. http://www.spec.org/osg/cpu95/.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of 4 Perfect benchmark programs. In *Proceedings of the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991. Also Technical Report 1193, CSRD, Univ. of Illinois.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[GLL97] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Experience with efficient array data flow analysis for array privatization. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 157–167, Las Vegas, Nevada, June 1997.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[KPRS96] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *International J. of Parallel Programming*, 24(6):579–598, December 1996.

[MAL92] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Data dependence and data-flow analysis of arrays. In *5th Workshop on Languages and Compilers for Parallel Computing (Yale University tech. report YALEU/DCS/RR-915)*, pages 283–292, August 1992.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, August 1993. Also available as Tech. Report CS-TR-3196, Dept. of Computer Science, University of Maryland, College Park.

[PW94] William Pugh and David Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, Dept. of Computer Science, University of Maryland, College Park, November 1994.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20(3):635–678, May 1998. http://www.acm.org/pubs/citations/journals/toplas/1998-20-3/p635-pugh/.

[SK98] Vivek Sarkar and Kathleen Knobe. Enabling sparse constant propagation of array element via array SSA form. In *Static Analysis Symposium (SAS'98)*, 1998.

[TP92] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, September 1992.

[Tu95] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[Won95] David G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, Dept. of Computer Science, The University of Maryland, August 1995. Available as ftp://ftp.cs.umd.edu/pub/omega/davewThesis/davewThesis.ps.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.