Program Optimization and Parallelization Using Idioms

SHLOMIT S. PINTER
Technion — Israel Institute of Technology
and
RON Y. PINTER
IBM Israel Science and Technology

Programs in languages such as Fortran, Pascal, and C were designed and written for a sequential machine model. During the last decade, several methods to vectorize such programs and recover other forms of parallelism that apply to more advanced machine architectures have been developed (particularly for Fortran, due to its pointer-free semantics). We propose and demonstrate a more powerful translation technique for making such programs run efficiently on parallel machines which support facilities such as parallel prefix operations as well as parallel and vector capabilities. This technique, which is global in nature and involves a modification of the traditional definition of the program dependence graph (PDG), is based on the extraction of parallelizable program structures ("idioms") from the given (sequential) program. The benefits of our technique extend beyond the above-mentioned architectures and can be viewed as a general program optimization method, applicable in many other situations. We show a few examples in which our method indeed outperforms existing analysis techniques.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processor—*compilers; optimization*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Array data flow analysis, computational idioms, dependence analysis, graph rewriting, intermediate program representation, parallelism, parallel prefix, reduction, scan operations

This research was performed in part while on sabbatical leave at the Department of Computer Science, Yale University, and was supported in part by NSF grant number DCR-8405478 and by ONR grant number N00014-89-J-1906. A preliminary version of this article appeared in the Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages (Jan. 1991).
Author's addresses: S. Pinter, Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa 32000, Israel; email: shlomit@ee.technion.ac.il; R. Pinter, IBM Science and Technology, MATAM Advanced Technology Center, Haifa 31905, Israel; email: pinter@haifasc3.vnet.ibm.com.

1. INTRODUCTION

Many of the classical compiler optimization techniques [Aho et al. 1986] comprise the application of local transformations to (intermediate) code, replacing suboptimal fragments with better ones. One hopes, as is often the case, that repeating this process (until a fixed point is found or some other criterion is met) will result in an overall better program. To a large extent, attempts to vectorize—and otherwise parallelize—sequential code [Wolfe 1989] are also based on the detection of local relationships among data items and the control structures that enclose them. These approaches, however, are local in nature and do not recognize the structure of the computation that is being carried out.

A computation structure sometimes spans a fragment of the program which may include some irrelevant details (that might obscure the picture both to the human eye and to automatic parallelism detection algorithms). Such "idioms" [Perlis and Rugaber 1979], which are not expressible directly as constructs in conventional high-level languages, include innerproduct calculations and recurrence equations in numeric code, data structure traversal in a symbolic context, selective processing of vector elements, and patterns of updating shared values in a distributed application. Recognizing them, which amounts to "lifting" the level of the program, is far beyond the potential of the classical, local methods.

Once recognized, such structures can be replaced lock, stock, and barrel by a new piece of code that has been highly optimized for the task at hand. This pertains to sequential target machines, but is most appealing in view of the potential gains in parallel computing. For example, a loop computing the convolution of two vectors, which uses array references, can be replaced by an equivalent fragment using pointers which was tailored by an expert assembler programmer. The same code can be replaced by a call to a BLAS [Dongarra et al. 1988; Lawson et al. 1979] routine that does the same job on the appropriate target machine. Better yet, if the machine at hand—for example, TMC's CM-2 [Thinking Machines Corp. 1987]—supports summary operators, such as reduction and scan which work in time $O(\log n)$ using a parallel prefix implementation [Blelloch 1989; Ladner and Fischer 1980] rather than $O(n)$ on a sequential machine (where n is the length of the vectors involved), then the gains could be even more dramatic.

In this article we propose a method for extracting parallelizable idioms from scientific programs. We cast our techniques in terms of the construction and analysis of the *computation graph*, which is a modified extension of the program dependence graph (PDG) [Ferrante et al. 1987], since PDG-based analysis methods seem to be most flexible and amenable for extensions. Forming this new type of graph from a source program—as is necessary for the required analyses—and using it for optimization transformations involves three major steps:

—When necessary, loops are unrolled so as to reach a *normal form*[1], whose

[1] In the sense of Munshi and Simons [1987], not that of Allen and Kennedy [1987].

precise definition and the algorithms for its formation are an additional contribution of this article.

—For each innermost loop in the program, we construct the computation graph for the basic block constituting its body. Certain conditionals (which are classified as "data filters") are handled gracefully, whereas others we disallow.

—The graph of each loop is replicated three times, for the initial, "middle," and final iterations, and together with an additional control node we obtain the computation graph of the whole loop. This process is repeated as we go up the nesting structure as far as possible (until we hit a problematic branching structure).

Once the graph is built, we apply optimization transformations to it using appropriate algorithms that we provide for this purpose.

Using this method, we were able to extract structures that other methods fail to recognize, as reported in the literature and as verified by experiments with a variety of existing tools, thereby providing a larger potential for speedup. Its power can be demonstrated by a small example, taken from Allen et al. [1987], who gave up on trying to parallelize the following loop (which indeed cannot be vectorized):

```
Example 1:      DO 100 I=2, N-1
                C(I)=A(I)+B(I)
                B(I+1)=C(I-1)*A(I)
          100   CONTINUE
```

Known methods (we examined KAP [Huson et al. 1986] Version 6 which can recognize second- and third-order linear recurrences, VAST-2 [Brode 1981], and other noncommercial systems), and even a human observer, may not realize that this loop hides two independent recurrences that can be implemented [Kogge and Stone 1973] in time $O(\log n)$ (n being the value of N) rather than $O(n)$. This kind of transformation can, however, be effected using our computation graph. We believe that our techniques can also be fitted to other program analysis frameworks, such as symbolic evaluation and plan analysis.

Another advantage of our approach is that it handles data and control uniformly: our method encapsulates the effect of inner loops so that idioms spanning deep nesting can be identified. Moreover, by incorporating data filters that abstract certain forms of conditionals, idioms for the selective manipulation of vector elements (which are useful on vector and SIMD machines) can be extracted. Finally, due to the careful definition of the framework and by the appropriate selection of transformation rules, our method can uniformly serve a wide variety of target architectures, ranging from vector to distributed machines.

In what follows we first review other work on compiler transformations and evaluate its limitations for purposes of idiom extraction. Then, in Section 3, we provide a formal presentation of our technique, including the definition of loop normalization, the construction of computation graphs, and the neces-

sary transformation algorithms. In Section 4 we exemplify our techniques, stressing the advantages over existing methods. Section 5 tries to put our work in perspective, discusses applications other than parallelism, and proposes further research.

2. EXISTING PARALLELIZATION METHODS

Much of the traditional program analysis work culminated in the definition and use of the above-mentioned PDG or its constituents, the control and data dependence graphs. Such a graph defines the flow of control in the program as well as the dependence between the variables being used in the program's statements; similar graphs were presented in Kuck et al. [1981] and Wolfe [1989] together with many compiler transformations mainly for recovering and improving vectorization. Analyzing the graph enables the detection of loop-invariant assignments for purposes of vectorization, classifies the type of other dependences as being loop carried and internal ones, and when applied to programs in single static assignment (SSA) form [Cytron et al. 1989] it can be quite effective.

However, the analysis methods allowed by this framework, e.g., as embodied in the PTRAN system [Allen et al. 1988], are very much tied to the original structure of the program. Moreover, the focus is on following dependences as they are reflected at the syntactic level (variable names and statements) rather than tracking the flow of data (among values) using deeper semantic analysis. Thus, this approach is not quite strong enough to allow us to extract patterns of data modification that are not readily apparent from the source program.

For example, the PDG of the loop shown in Section 1 (Example 1) contains two nodes on a directed cycle, implying a circular dependency. This representation is not accurate since the two reductions hidden in this computation do not depend on each other. Thus, the PDG is more conservative than our graph which will expose the potential for parallelization in this loop.

Partial success in recovering reduction-type operators is reported in the work on KAP [Huson et al. 1986], Parafrase [Lee et al. 1985; Polychronopoulos et al. 1986], and VAST [Brode 1981; Pacific-Sierra 1990] (all these tools also do a very good job in vectorization and loop parallelization). The restructuring methods that were used are in the spirit of the PDG-based work, but the techniques for detecting reduction operators are different from ours and somewhat ad hoc. The KAP system (the strongest in recovering these types of operations) can recognize second- and third-order linear recurrences, yet when such computations are somewhat intermixed as in our previous example it fails to do so.

Another transformation-based approach is that of capturing the data dependence among array references by means of dependence vectors [Chen 1986; Karp et al. 1967; Lamport 1974]. Then methods from linear algebra and number theory can be brought to bear to extract wavefronts of the computation. The problem with this framework is that only certain types of dependences are modeled; there is no way to talk about specific operations (and

their algebraic properties), and in general it is hard to extend. We note that the work of Banerjee [1988] on data dependence analysis and that of Wolfe [1989] can be viewed as extending the above-mentioned dependence graph framework by using direction and distance vectors.

More recently, Callahan [1991] provided a combination of algebraic and graph-based methods for recognizing and optimizing a generalized class of recurrences. This method is indeed effective in exploiting this type of computation, but it is strongly tied to a specific framework and does not lend itself to generalizations to nonrecurrence-type transformations.

Finally, various symbolic-evaluation methods have been proposed [Jouvelot and Dehbonei 1989; Letovsky 1988; Rich and Waters 1988], mostly for plan analysis of programs. These methods all follow the structure of the program rigorously, and even though the program is transformed into some normal form up front and all transformations preserve this property, still these methods are highly sensitive to "noise" in the source program. More severely, the reasoning about array references (which are the mainstay of scientific code) is quite limited for purposes of finding reduction operations on arrays.

3. COMPUTATION GRAPHS AND ALGORITHMS FOR IDIOM EXTRACTION

In this section we propose a new graph-theoretic model to represent the flow of data and the dependences among values in a program, namely, the *computation graph*. To allow effective usage of this model, programs must be preprocessed, and most importantly, loops must be normalized; this technique will be presented in the beginning of this section. Next we define the computation graph formally and list some of its important properties. Finally we provide an algorithmic framework that uses this new abstraction in order to analyze the structure of programs and identify computations that can be parallelized or otherwise optimized; specific patterns and their replacements are provided.

To guide the reader through this section we use the sample program of Figure 1, which is written in Fortran. This example is merely meant to illustrate the definitions and algorithms, not to demonstrate the advantages of our approach over other work; this will be done in Section 4.

3.1 Preprocessing and Loop Normalization

Before our analysis can be applied to a given program we must transform it so as to allow the computation graph to effectively capture the program's structure for purposes of idiom extraction. Some of these "preprocessing" transformations are straightforward and well-known optimizations, as we shall point out in passing, but others require some new techniques that we shall explain.

At this stage we assume that programs contain only IF statements that serve as *data filters*, as in `IF(A(I).EQ.0) B(I) = I`. Some of the other conditionals can be pulled out of loops or converted to filters using standard techniques, but those that cannot are outside the scope of our techniques. Finally, we assume that all array references depend linearly (with integral coefficients) on loop variables.

```
      DO 10 I=1,N
      T = I+4
C     assume N is even
      DO 10 J=3,N
      A(I,J) = A(I,J) + T*A(I,J-2)
10 CONTINUE
```

Fig. 1. A sample Fortran program.

The preprocessing transformations are as follows:

—Standard basic block optimizations (such as dead-code and dead-store elimination, common subexpression detection, etc.) are performed per Aho et al. [Section 9.4, 1986], so that the resulting set of assignments to values does not contain redundancies. This implies that the expression DAG that is obtained represents only the def-use relationship between variables that are live upon entry to each block and those that are live at the exit. We further assume that the relation among values is "simple," meaning that each defining expression is of some predefined form. Typical restrictions could be that it contains at most two values and one operator or that it be a linear form; which restriction is imposed depends on the type of recognition algorithm we want to apply later (in Section 3.3).

—To extend the scope of the technique we assume that loops are recognized wherever possible.

—Loops are *normalized* with respect to their index, as defined below. Munshi and Simons [1987] have observed that by sufficient unrolling all loop-carried dependences can be made to occur only between consecutive iterations. The exact conditions stating how much unrolling is required and the technique for transforming a loop to normal form have never been spelled out; thus—in what follows—we provide the necessary details.

Normalization comprises two steps: determination of *loop-carried dependences* (specifically among different references of arrays) and *unrolling*. If data dependence analysis does not provide the exact dependences we may unroll the loop more than is necessary without affecting the correctness of the process.

Customarily, e.g., Allen et al. [1987], Ferrante et al. [1987], and Wolfe [1989], data dependence is defined between statements, rather than scalar values and array references. We provide the following definition for data dependence among values in a program (and from now on we use only this notion of data dependence):

Definition 1. Value A is *data dependent* on value B if they both refer to the same memory location and either (i) a use of A depends on the definition of B (flow dependence), (ii) the definition of A redefines B which was used in

an expression (antidependence), or (iii) the definition of A redefines the previous definition of B (output dependence).

A dependence is *loop carried* if it occurs between values that appear in different iterations of a loop.

In the following example the array A has two references (and so does B). Since $i + m \neq j$ for all $2 \leq i, j \leq m$, where m is the value of M, there is no loop-carried dependence between the references of A.

```
Example 2:      DO 20 I=2, M
                B(I)=2*A(I)
             20 A(I+M)=B(I-1)
```

Conceptually one can view the array A as if it were partitioned into two (pairwise disjoint) subarrays, one for each reference. Between the two references of B there is a loop-carried flow dependence which involves only consecutive iterations. Now we are ready to say when a loop is in normal form:

Definition 2. A loop is in *normal form* if each of its loop-carried dependences is between two consecutive iterations.

Any given loop can be normalized as follows. Find all loop-carried dependences, and let the *span* of the original loop be the largest integer k such that some value set in iteration i depends directly on a value set in iteration $i - k$. Then, to normalize a loop with span $k > 1$, the loop's body is copied (unrolled) $k - 1$ times. The loop's index is adjusted so it starts at 1 and is incremented appropriately at each iteration. Loops are normalized from the most deeply nested outward.

In the code of Figure 1, if $m > 4$ (where m denotes the value of M) then the inner loop needs to be normalized by unrolling it once, whereas the outer loop is already in normal form (it is vacuously so, since there are no reference loop-carried dependences). We also assume that the temporary scalar T has been expanded, thus obtaining the program in Figure 2.

In order to compute the span it is not enough to find the distance (in the loop iteration space) of a dependence. In the following example the span is 1 although the distance in the first assignment is 2:

```
Example 3:      DO 10 I=1, N
                A(I+2)=A(I)+X
                A(I+1)=A(I+2)+Y
             10 CONTINUE
```

Here there is only one loop-carried flow dependence from the value set in the second statement to its use by the first statement in the following iteration.

At the current stage the scope of our work includes only loops for which the span is a constant. Note that the loop can still include array references which use its bound in some cases. For example the span of the following loop is 2:

```
Example 4:      DO 10 I=2, N-1
                A(N-I)=A(N-I)+A(N+2-I)
             10 CONTINUE
```

```
      DO 10 I=1,M
      T(I) = I+4
C     assume M is even
      DO 10 J=1,M-2,2
      A(I,J+2) = A(I,J+2) + T(I)*A(I,J)
      A(I,J+3) = A(I,J+3) + T(I)*A(I,J+1)
10 CONTINUE
```

Fig. 2. The program of Figure 1 in normal form.

An immediate result of normalization is that loop-carried dependences become uniform. This observation is, later on, being used in the computation graph to explicitly reveal all the possible interaction patterns among iterations of the loop and its surrounding context. Thus, we can match graph patterns in order to enable the transformations. Finally, note that a similar notion to normalization appears in Aiken and Nicolau [1988] where the computation of a loop is carried out until a steady state is reached.

Comments on Normalization. We cannot handle indirect array references, such as `A(X(I))`, without employing partial evaluation techniques. Indirect referencing is mostly used to generate access patterns which are irregular and data dependent; since current target machines can take advantage of such patterns only at run-time, this limitation is inherent to all compiler-based approaches.

There is a relation between the span of a loop and Banerjee's [1988] loop-carried dependence distance, which defines for a given dependence the distance between the two points of the iteration space that cause the dependence. In particular, if a loop is being normalized then there is at least one dependence distance equal to the span. In current transformations, distance values are mainly used for introducing the proper synchronization delay (of some sort). This delay must cause a wait which lasts through the duration of the execution of as many operations as may be generated by unrolling using the span. Both methods can handle (at the moment) only constant span or distances which may not be the same for all the statements in the loop.

At first it may seem that unrolling a loop for normalization results in extraneous space utilization. However, we point out that unrolling loop bodies is done by many optimizing compilers in order to better utilize pipeline-based machines; thus space usage and other resource issues are similar for both cases.

3.2 The Computation Graph

Given a program satisfying the above assumptions, we define its computation graph (CG) in two stages: first we handle basic blocks; then we show how to

represent normalized loops; and finally we show how to incorporate data filters.

The CG for a basic block is a directed acyclic graph $G = (V, E)$, defined as follows:

—There is a node $v \in V$ representing each value that is defined in the block; there is also a node in V for each value that is live [Aho et al. 1986] at the entry of the block and that is used in it (initial value). Each node is uniquely identified by a tag; nodes that correspond to values set in assignments are tagged by the corresponding statement number, and for nodes that represent initial values additional (new) tags are generated.

—If the loop control variable is used in the computations within its body we assume it has a proper initial value.

—The meaning of an edge (u, v) in the graph is that the computation of u must end before the computation of v starts. There are three types of edges drawn between nodes: one represents the classical def-use relationship [Aho et al. 1986] (which includes flow dependence), and the other two represent two kinds of data dependences between values (per Definition 1). We draw a *def-use* edge from u to v if the value u is used in the definition of v. We draw an *output* data dependence edge from u to v if v is output dependent on u as per Definition 1. Lastly, we draw an *antidependence* edge from u to v if the definition of u uses a value on which v is antidependent (unless this edge is a self-loop which is redundant since the fetch is done before the assignment).

—Each node v is labeled by the expression according to which the value is being computed. The labeling uses a numbering that is imposed on the def-use edges entering the node (i.e., the arguments). The label $\perp$ is used for initial values.

Using the array references verbatim as atomic variables, the basic block constituting the body of the inner loop of Figure 2 gives rise to the computation graph shown in Figure 3. Since we shall be looking for potential reduction and scan operations, which apply to linear forms, we allow the functions at nodes to be ternary multiply-add combinations. Note also that the nodes denoting the initial values of the variables could be replaced when the graph is embedded in a larger context, as we shall see.

Next we define computation graphs for normalized loops. Such graphs are obtained by replicating the DAGs representing the enclosed body[2] as follows:

—Three copies of the graph representing the body are generated: one copy represents the initial iteration; one is a typical middle iteration; and one stands for the final iteration. Each node is uniquely identified by extending the tag of the replicated node with the unrolled copy it belongs to (initial, middle, or final).

[2] At the innermost level these are basic blocks, but as we go up the structure these are general computation graphs.

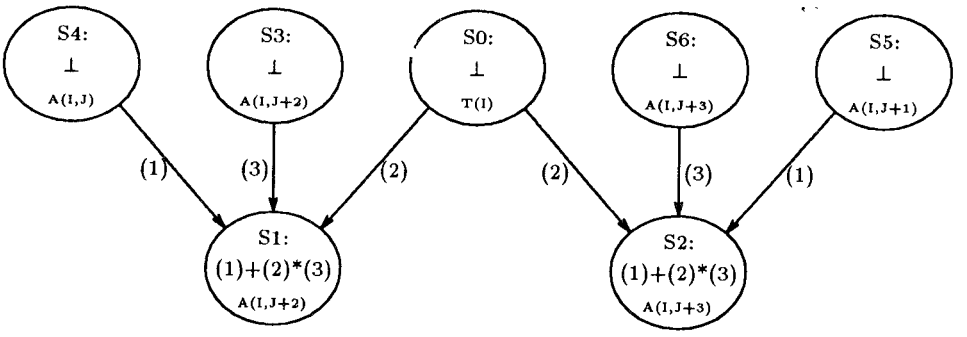

Fig. 3. The computation graph of the basic block constituting the inner loop of the program in Figure 2. In case there are both a flow and an output dependence we draw only the flow edge.

—Loop-carried data dependence edges (cross iteration edges) are drawn between the different copies.

—Each instance of the loop variable (like in expressions of array references) is replaced with `init`, `mid`, and `fin` depending on its appearance in the initial, middle, or final copy, respectively.

—Two nodes in different copies that represent a value in the same memory location (like `A(init+2)` and `A(mid)` when the stride of the loop is 2) are coalesced, unless there exists a flow or an output dependence edge between them (in which case there are possibly two different values; see also note below on antidependences).

—The graph is linked to its surroundings by data dependence edges where necessary (i.e., from initializations outside the loop and to subsequent uses).

The graph is constructed on a loop-nest by loop-nest basis, i.e., there is no need to build it for the whole program. In fact, one might decide to limit the extent of the graph and its application in order to reduce compile time or curb the potential space complexity.

Note. If a node v is a target of an antidependence edge then v is also a target of an output dependence edge. Consider the existence of an antidependence edge from u to v, i.e., the value that was stored in the same memory location as v and that was used in the definition of u is now being destroyed. This means that there is a node w which constitutes the old value that was used in the definition of u; thus there is an output dependence edge between w and v in addition to the def-use edge from w to u. In general, if the meaning of an output dependence edge, say (u, v), is that all the uses of u—on def-use edges leaving u—must be executed before v, then the antidependence edges are redundant.

Figure 4 shows the computation graph of the inner loop (`J`) of the example in Figure 2. The graph comprises three copies of the graph from Figure 3. Node S4 of the middle copy is coalesced with node S1 of the initial copy since

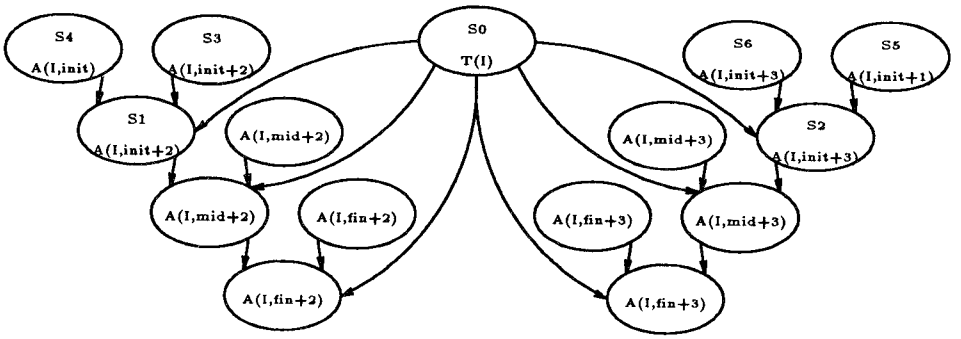

Fig. 4. The computation graph of the inner loop of the program in Figure 2.

they both represent the same value (the stride of `J` is 2, thus, `init + 2 = mid`), and there are no loop-carried output dependences between these nodes. Similarly, node S5 of the middle copy is coalesced with node S2 of the initial copy. The same pattern is repeated between the final and the middle copies. Node S0 represents the value of `T(I)` which is the same value for all three copies. The graph for the whole program (including the outer `I` loop) would consist of three copies of what is shown in Figure 4, with appropriate annotation of the nodes.

A more general example that includes anti, output, and flow loop-carried dependences is presented in Figure 5.

The main contribution of the triple unfolding of loops is that when combined with normalization the computation graph represents explicitly all possible dependences. The computation graph spans a "signature" of the whole structure, and we call it the *summary form* of the loop. Once inner loops are transformed into their summary form, they are treated as part of the body of the enclosing loops and may—in turn—be replicated similarly to yield another summary form.

The following lemma summarizes this observation, and it will serve as the foundation for the applicability of the transformations described in the next Section.

LEMMA 1. *Three iterations of a normalized loop capture its entire data dependence structure.*

PROOF. Recall that all loop-carried dependences are only between consecutive iterations. Thus, having three nodes to represent a value in a normalized loop assures the property that dependences involving the second copy are the same for all the middle iterations, i.e., those that are not the first or the last one. □

Note, that when embedding a loop in a program fragment we need to cover the cases in which the loop's body is executed fewer than three times. Since we are interested in parallelizing loops that must be iterated more than twice we omit the description of such an embedding.

```
    DO I = 1,N
S1:   B(I) = A(I) * 5
S2:   D(I) = D(I-1) + C(I+1)
S3:   C(I) = B(I-1)
    ENDDO
```

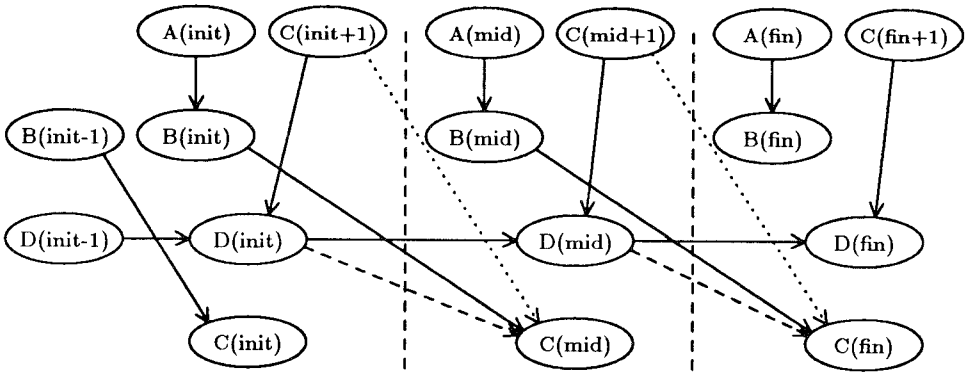

Fig. 5. A program and its CG. Dashed and dotted arrows are anti and output dependences, respectively; the vertical dashed lines separate the three copies.

The computation graph can also represent some types of conditional constructs like data filters.

Definition 3. A *data filter* is a conditional expression—none of whose constituent values is involved in any loop-carried dependence—controlling an assignment.

A data filter is represented in the graph by a special *filter node*, denoted by a double circle. The immediate predecessors of this node are the arguments of the Boolean expression constituting the predicate of the conditional, the previous definition of the variable to which the assignment is being made, and the values involved in the expression on the right-hand side of the assignment. The outgoing edge from the filter node goes to the node corresponding to the value whose assignment is being controlled. The filter node is labeled by a conditional expression whose *then* value is the right-hand side of the original assignment statement, and the *else* value is the previous (old) value of the variable to which the assignment is being made. This expression is written in terms of the incoming edges, as Figure 6 demonstrates: in statement 20, the assignment to S is controlled by a predicate on the value of

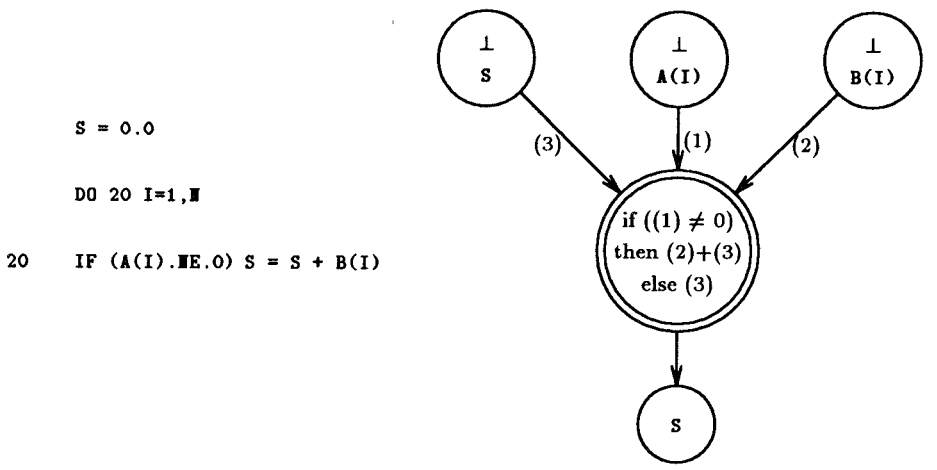

Fig. 6. A program with a data filter and the computation graph of its loop's body.

A(I); hence the computation graph of the loop's body contains a filter node. Figure 7 shows the computation graph of the program of Figure 6.

Finally, note that when the CG represents a program without parallel constructs there cannot be edges from the middle copy to the initial copy of the body, nor can there be edges from the final copy to the middle copy. Additionally, due to normalization, there are no cross iteration edges between the initial and final copies. The following lemma summarizes the main structural properties of the CG; these properties are used in designing transformation patterns and assuring the correctness of their application.

LEMMA 2. *A CG representing a normalized loop of a sequential program is acyclic, and the only existing cross iteration edges are from the initial to the middle copy and from the middle to the final copy.*

Computation graphs are acyclic by construction. This is obvious for basic blocks and is not hard to show both for loops as well as for data filters.

3.3 Algorithms

Having constructed the computation graph, the task of finding computational idioms in the program amounts to recognizing certain patterns in the graph. These patterns comprise graph structures such as paths or other particular subgraphs, depending on the idiom, and some additional information pertaining to the labeling of the nodes. The idiom recognition algorithm constitutes both a technique for identifying the subgraph patterns in the given graph as well as the conditions for when they apply, i.e., checking whether the context in which they are found is one where the idiom can indeed be used.

Overall, the optimization procedure consists of the following algorithmic ingredients:

—Matching and replacement of individual patterns is achieved by using graph grammars to describe the rewrite rules. While rewriting, we also

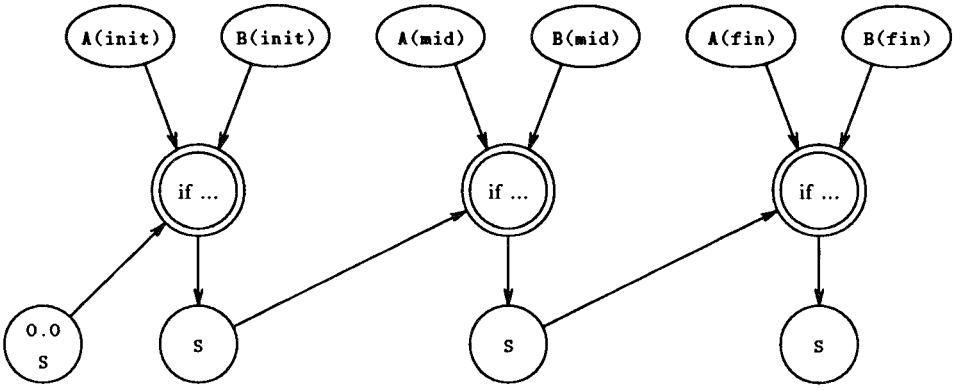

Fig. 7 The computation graph of the program of Figure 6.

transform the labels (including the operators, of course), thus generating the target idioms. We make sure no side effects are lost by denoting forbidden entries and exits per Rosendahl and Mankwald [1979].

—We need to provide a list of idioms and the graph-rewriting rules that replace them. These include structures such as reduction, scan, recurrence equations, transposition, reflection, and FFT butterflies. Compositions thereof, such as inner product, convolution, and other permutations, can be generated as a preprocessing stage. Notice that data filters can be part of a basic rule.

—At the top level, we (repeatedly) match patterns from the given list according to a predetermined application schedule until no more changes are applicable (or some other termination condition is met). This means that we need to establish a precedence relation among rules that will govern the order in which they are applied. This greedy tactic, which is similar to the conventional application of optimization transformations [Aho et al. 1986], is just one alternative. One could assign costs that reflect the merits of transformations and find a minimum cost cover of the whole graph at each stage, and then iterate.

We next elaborate on each of the above items, filling in the necessary details. First we discuss graph-rewriting rules. Since we are trying to summarize information, these will be mostly reduction rules, i.e., shrinking subgraphs into smaller ones. More importantly, there are three characteristics that must be matched besides the skeletal graph structure: the operators (functions), the array references, and context, i.e., relationship to the enclosing structure.

The first two items can be handled by looking at the labels of the vertices. The third involves finding a particular subgraph that can be replaced by a new structure and making sure it does not interfere with the rest of the computation. For example, to identify a reduction operation on an array, we need to find a path of nodes all having the same associative-operator label (e.g., multiplication, addition, or an appropriate linear combination thereof)

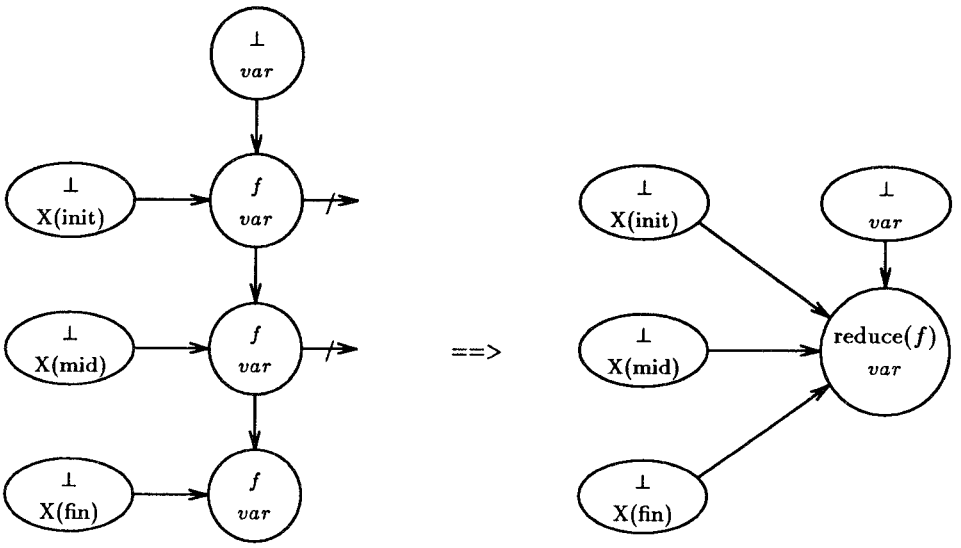

Fig. 8. Matching and replacement rule for reduction.

and using consecutive (i.e., initial, middle, and final) entries in the array to update the same summary variable; we also need to ascertain that no intervening computation is going on (writing to or reading from the summary variable).

Both the annotations of the nodes as well as the guards against intervening computations are part of the graph grammar productions defining the replacement rules. Figures 8 and 9 provide two such rules to make this notion clear. Here we assume that vectorization transformations, including scalar expansion, have occurred, so we rely on their results when looking for patterns; the vectorization transformations themselves can be applied first by using a different tool or can be part of our rules base (they can be formulated similarly with appropriate expanding rewrite rules [Pinter and Mizrachi 1992]).

To accommodate data filters, additional rules are necessary. For example, the rule of Figure 10 takes care of a filtered reduction. Notice the similarity to the rule of Figure 8, if we splice out the filter nodes and redirect the edges; the scan rule of Figure 9 can be similarly used to derive a rule for a filtered scan.

Once such rules are applied, the computation graph contains new types of nodes, namely, summary nodes representing idioms. These nodes can, of course, appear themselves as candidates for replacement (on the left-hand side of a rule), thereby enabling further optimization. Obviously, we would apply the rules for lower-level optimizations first and only then use the others, but one should not get the impression that this procedure necessarily follows the loop structure of the given program. On the contrary, the computation graph as constructed allows the detection of structures that might otherwise be obscured, as can be seen in Section 4.

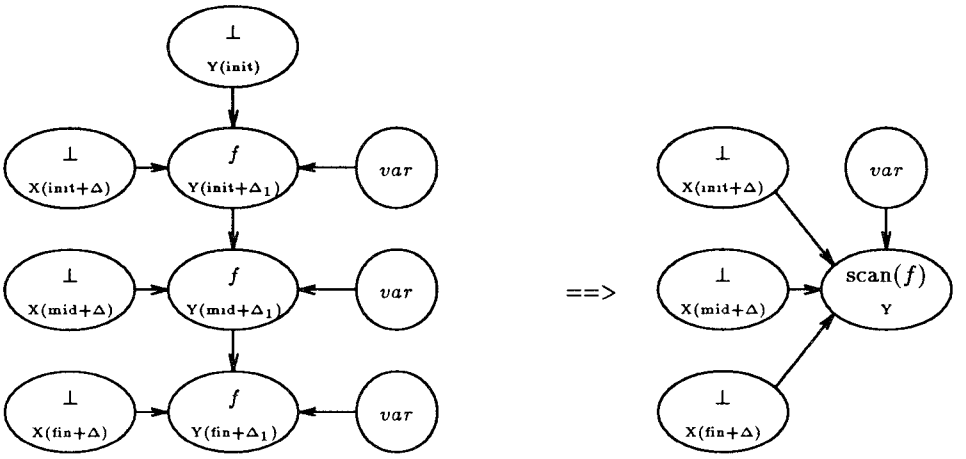

Fig. 9. Matching and replacement rule for scan.

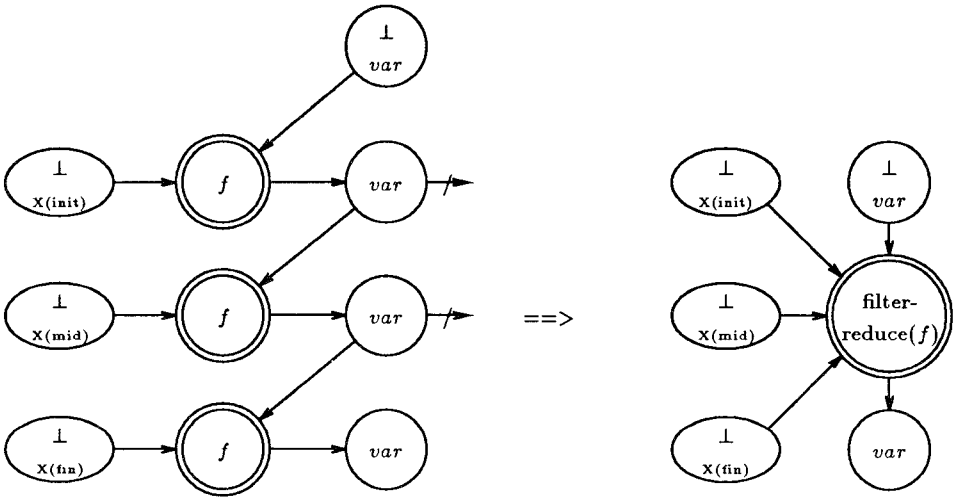

Fig. 10. A matching and replacement rule for a filtered reduction.

The result of applying the rule of Figure 9 to the graph of Figure 4 is shown in Figure 11.

If we had started with a graph for the entire program of Figure 2, not just the inner loop (as represented in Figure 4), then applying a vectorization rule to the result would have generated the following program:

```
      DOALL 10 I = 1, N
      T(I) = I + 4
      SCAN(A(I, *), 1, M - 1, 2, T(I), "*", "+")
      SCAN(A(I, *), 2, M, 2, T(I), "*", "+")
10    CONTINUE
```

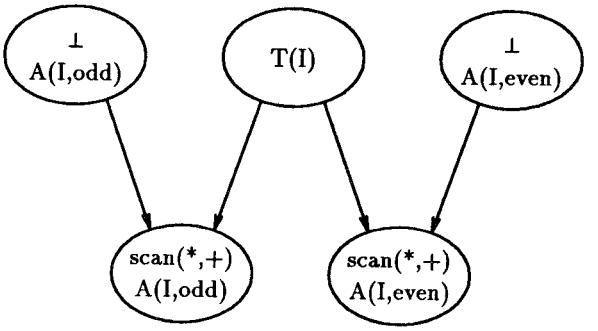

Fig. 11. The computation graph resulting from applying the transformations of Figure 9 to the graph in Figure 4.

We use an arbitrary template for SCAN which includes all the necessary information, including array bounds, strides inside vectors or minors, and the operations to be performed in reverse Polish notation.

In the case of Figures 6 and 7, the resulting problem would be

```
T(1:N) = 0.0
WHERE(A(1:N).NE.0) T(1:N) = B(1:N)
S = REDUCE(T, 1, N, 1, " + ")
```

where T is a newly defined array. (This somewhat awkward phrasing of the resultant program is due to the way most parallel dialects, such as Fortran 90 and others [Guzzi et al. 1990], are defined.)

In general, the order in which rules are applied and the termination condition depend on the rule set. If the rules are Church-Rosser [Church and Rosser 1936] then this is immaterial, but often they are competing (in the sense that the application of one would outrule the consequent application of the other) or are contradictory (creating potential oscillation). This issue is beyond the scope of this article, and we defer its discussion to general studies on properties of rewriting systems.

4. EXAMPLES

To exemplify the advantages of our idiom recovery method, we present here three cases in which other methods cannot speed the computation up but ours can (if combined properly with standard methods). We do not include the complete derivation in each case, but we outline the major steps and point out the key items.

The first example is one of the loops used in a Romberg integration routine. The original code (as released by the Computer Sciences Corporation of Hampton, VA) looks as follows:

```
Example 5:      DO 40 N=2, M+1
                KN2 = K + N - 2
                KNM2 = KN2 + M
                KNM1 = KNM2 + 1
                F = 1.0 / (4.0**(N - 1) - 1.0)
```

```
      TEMP1 = WK(KNM2) - WK(KN2)
      WK(KNM1) = WK(KNM2) + F*TEMP1
40    CONTINUE
```

After substitution of temporaries, scalar expansion of F, and renaming of loop bounds, which are all straightforward transformations, we obtain

```
      DO 40 I = K+M, K+2*M-1
      F(I) = 1.0 / (4.0**(I-K-M+1) - 1.0)
      WK(I+1) = WK(I) + F(I)*(WK(I) - WK(I-M))
40    CONTINUE
```

Notice that no normalization is necessary in this case since from data dependence analysis the only loop-carried data dependence between the references to WK is of distance 1 which implies a span of 1. Furthermore, we observe that the computation of F(I) can be performed in a separate loop, as can be deduced using the well-known technique of "loop distribution."

The loop computing F can be easily vectorized, so all we are left with is a loop containing a single statement which computes WK(I+1). If we draw the computation graph for this loop and allow operators to be linear combinations with a coefficient (F(I) in this case), we can immediately recognize the scan operation that is taking place and thus produce the following:

```
WK1(K+M:K+2*M-1) = WK(K:K+M-1)
SCAN(WK, K+M, K+2*M-1, 1, WK1, "-", F, "*", "+")
```

The second example is the program appearing in the introduction (taken from Allen et al. [1987], who gave up on trying to parallelize it). Notice that the loop is already in normal form. Once the loop is unfolded three times, as required, we detect two independent chains in the computation graph, one including the computation of the even[3] entries in B and C, and the other computing the odd entries. If the data are arranged properly, the whole computation can be performed using the method for computing recurrence equations presented in Kogge and Stone [1973], taking time $O(\log n)$ (n being the value of N) rather than $O(n)$. The computation graph can be seen in Figure 12. Notice that it is completely different from the PDG which consists of two nodes on a directed cycle.

Finally, we parallelize a program computing the inner product of two vectors, which is a commonly occurring computation:

```
      P = 0
      DO 100 I = 1, N
      P = P + X(I)*Y(I)
100   CONTINUE
```

Traditional analysis of the program (in preparation for vectorization) replaces the loop's body by

```
T(I) = X(I)*Y(I)
P = P + T(I)
```

[3] We use "even" and "odd" here just to distinguish between the chains; the recognition procedure does not need to know or prove this fact at all

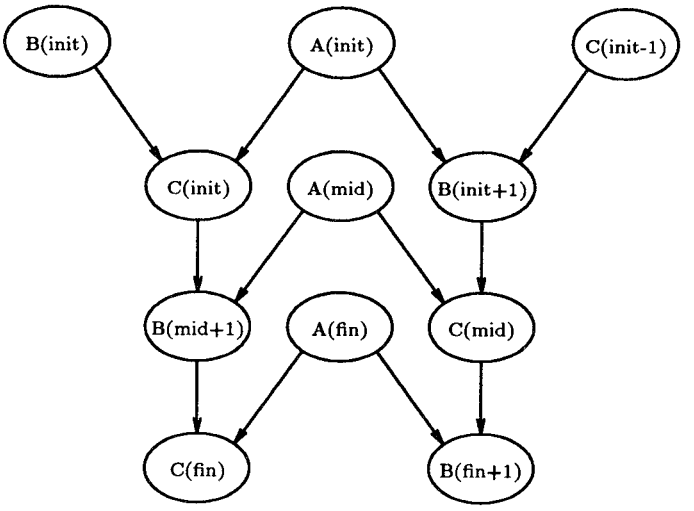

Fig. 12. The computation graph of example 1.

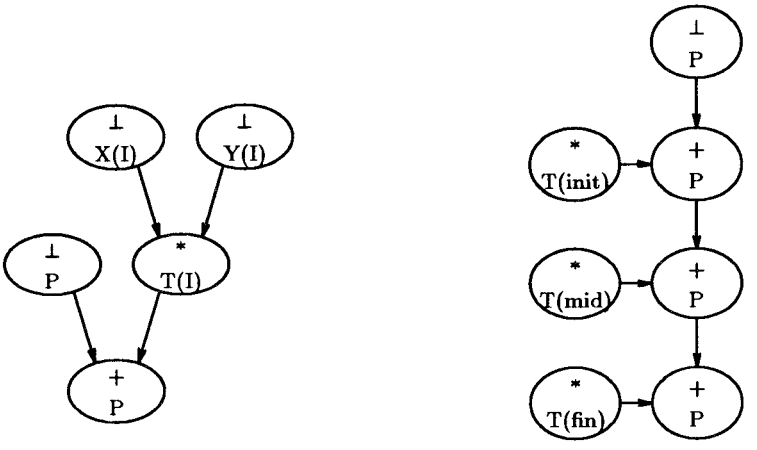

Fig. 13. Transforming an inner product to a reduction.

Figure 13 shows the computation graphs of this transformed basic block and that of the relevant part of the loop. The vectorizable portion of the loop can be transformed into a statement of the form `T = X*Y`, and what is left can be matched by the left-hand side of the rule of Figure 8. Applying the rule produces the graph corresponding to the statement `P = REDUCE(T, 1, N, 1, " + ")`.

The recognition of an inner product using our techniques will not be disturbed by enclosing contexts such as a matrix multiplication program. The framed part of the program in Figure 14 produces the same reduced graph that appears replicated after treating the two outermost loops.

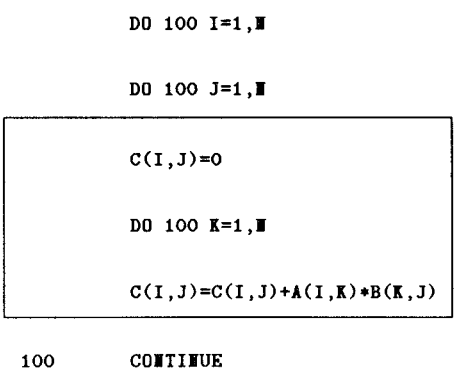

Fig. 14. The portion of a matrix multiplication program that is recognized by the transformation of Figure 12.

5. DISCUSSION

We have proposed a new program analysis method for purposes of optimization and parallelization. It extracts more information from the source programs, thereby adding to existing methods the ability to recognize idioms more globally. When comparing our method to the other most common methods we can say the following:

—Constructing and then using the program dependence graph (PDG) or a similar structure focuses on following dependences as they are reflected at the syntactic level alone. Thus, the PDG is more conservative than our computation graph (CG) which contains more information and exposes further potential for parallelization.

—There was partial success in recovering reduction operators (using PDG-like analysis) in the work on Parafrase [Lee et al. 1985; Polychronopoulos et al. 1986]. The techniques employed, however, are somewhat ad hoc compared to ours.

—The places where normalization is carried out are, in general, not applicable for known parallelizing transformations (thereby providing a proper extension to automatic restructuring theory).

—Capturing the data dependence between array references by means of dependence vectors [Chen 1986; Karp et al. 1967; Wolfe 1989] and then solving a system of linear equations to extract the wavefronts of the computation are limited to functional (side effect free) sets of expressions.

—Symbolic and partial evaluation methods [Jouvelot and Dehbonei 1989; Letovsky 1988; Rich and Waters 1988] all follow the structure of the program rigorously, and even when the program is transformed into some normal form these methods are still highly sensitive to "noise" in the source program. Their major drawback is the lack of specific semantics for array references, limiting their suitability for purposes of finding reduction operations on arrays.

Our primary objective is to cater for machines with effective support of data parallelism. Our techniques, however, are not predicated on any particu-

lar hardware, but can rather be targeted to an abstract architecture or to language constructs that reflect such features. Examples of such higher-level formalisms are APL functionals and idioms, the BLAS package (which has many efficient implementations on a variety of machines), vector-matrix primitives as suggested in Agrawal et al. [1989], and languages such as Crystal, C*, and *lisp which all support reductions and scans.

All in all, we feel that this article makes both methodological and algorithmic contributions that should be further investigated. In addition to the constructs mentioned above, many others can be expressed as computation graph patterns and be identified as idioms [Pinter and Mizrachi 1992]; such constructs can be used to eliminate communication and synchronization steps that might appear in conjunction with parallel computations. Also, further work on algorithmic methods other than repeated application of the rules (such as fixed-point computations) is also necessary. Finally, we are currently implementing the techniques mentioned here as part of an existing parallelization system [Lempel et al. 1992] and plan to obtain experimental results that would indicate how prevalent each of the transformations is.

ACKNOWLEDGMENTS

The authors would like to thank Ron Cytron, Ilan Efrat, and Liron Mizrachi for helpful discussions, as well as Jeanne Ferrante and David Bernstein for comments on an earlier draft of this article. We would also like to thank the anonymous referees for their insightful reviews.

REFERENCES

AGRAWAL, A., BLELLOCH, G. E., KRAWITZ, R. L., AND PHILLIPS, C. A. 1989. Four Vector-matrix primitives. In the *Symposium on Parallel Algorithms and Architectures*. ACM, New York, 292–302.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

AIKEN, A. AND NICOLAU, A. 1988. Optimal loop parallelization. In the *SIGPLAN'88 Conference on Programming Language Design and Implementation*. ACM, New York, 308–317.

ALLEN, R. AND KENNEDY K. 1987. Automatic translation of FORTRAN programs to vector form.

ALLEN, F. E., BURKE, M., CHARLES, P., CYTRON, R., AND FERRANTE, J. 1988. An overview of the PTRAN analysis system for multiprocessing. *J. Parall. Distrib. Comput. 5*, 5 (Oct.), 617–640.

ALLEN, R., CALLAHAN, D., AND KENNEDY, K. 1987. Automatic decomposition of scientific programs for parallel execution. In the *14th Annual Symposium on the Principles of Programming Languages*. ACM, New York, 63–76.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, New York.

BLELLOCH, G. E. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput. C-38*, 11 (Nov.), 1526–1539.

BRODE, M. 1981. Precompilation of FORTRAN programs to facilitate array processing. *Computer 14*, 9 (Sept.), 46–51.

CALLAHAN, D. 1991. Recognizing and parallelizing bounded recurrences. In the *4th Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 589. Springer-Verlag, New York, 169–185.

CHEN, M. C. 1986. A parallel language and its compilation to multiprocessor machines or VLSI. In the *13th Annual Symposium on the Principles of Programming Languages*. ACM, New York, 131–139.

CHURCH, A. AND ROSSER, J. B. 1936 Some properties of conversion. *Trans. Am. Math Soc. 39*, 472–482.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M N., AND ZADECK, F K 1989. An efficient method of computing static single assignment form. In the *16th Annual Symposium on the Principles of Programming Languages*. ACM, New York, 25–35.

DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., AND HANSON, R. J. 1988 An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14*, 1 (Mar.), 1–17.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July), 319–349.

GUZZI, M. D., PADUA, D. A., HOEFLINGER, J. P., AND LAWRIE, D. H. 1990. Cedar FORTRAN and other vector and parallel FORTRAN dialects. *J Supercomput. 4*, 1 (Mar.), 37–62.

HUSON, C., MACKE, T., DAVIS, J. R., WOLFE, M. J., AND LEASURE, B. 1986. The KAP/205: An advanced source-to-source vectorizer for the Cyber 205 supercomputer. In the *International Conference on Parallel Processing*. CRC Press, Inc., 827–832.

JOUVELOT, P., AND DEHBONEI, B. 1989. A unified semantic approach for the vectorization and parallelization of generalized reductions. In the *International Conference on Supercomputing*. ACM, New York, 186–194.

KARP. R. M., MILLER, R. E., AND WINOGRAD, S. 1967. The organization of computations for uniform recurrence equations. *J. ACM 14*, 3 (July), 563–590.

KOGGE, P. M. AND STONE, H. S. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput. C-22*, 8 (Aug.), 786–793.

KUCK, D., KUHN, R. H., PADUA, D. A., LEASURE, B., AND WOLFE, M. 1981. Dependence graphs and compiler optimizations In the *18th Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York, 207–218.

LADNER, R. E. AND FISCHER, M. J. 1980. Parallel prefix computation. *J. ACM 27*, 4 (Oct.), 831–838

LAMPORT, L. 1974. The parallel execution of do loops. *Commun. ACM 17*, 2 (Feb.), 89–93.

LAWSON, C. L., HANSON, R. J., KINCAID, D R., AND KROGH, F. T. 1979. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft. 5*, 3 (Sept.), 308–323.

LEE, G., KRUSKAL, C. P., AND KUCK, D. J. 1985. An empirical study of automatic restructuring of nonnumerical programs for parallel processors. *IEEE Trans. Comput. C-34*, 10 (Oct), 927–933.

LEMPEL, O., PINTER, S. S., AND TURIEL, E. 1992. Parallelizing a C dialect for distributed memory MIMD machines. In the *5th Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 757. Springer-Verlag, New York, 369–390

LETOVSKY, S. I. 1988 Plan analysis of programs. Ph.D. thesis, YALEU/CSD/TR-662, Dept. of Computer Science, Yale Univ., New Haven, Conn.

MUNSHI, A. AND SIMONS, B. 1987. Scheduling sequential loops on parallel processors. Tech. Rep RJ 5546, IBM Almaden Research Center, San Jose, Calif.

PACIFIC-SIERRA RESEARCH. 1990. *VAST for RS / 6000*. Pacific-Sierra Research, Los Angeles, Calif.

PERLIS, A. J. AND RUGABER, S. 1979. Programming with idioms in APL. In *APL '79* ACM, New York, 232–235. Also, *APL Quote Quad*, vol. 9, no. 4.

PINTER, S. S. AND MIZRACHI, L. 1992. Using the computation dependence graph for compile-time optimization and parallelization. In *Proceedings of the Workshop on Advanced Compilation Techniques for Novel Architectures*. Springer-Verlag, New York.

POLYCHRONOPOULOS, C. D., KUCK, D. J., AND PADUA, D. A. 1986. Execution of parallel loops on parallel processor systems. In the *International Conference on Parallel Processing*. IEEE, New York, 519–527.

RICH, C. AND WATERS, R. C. 1988. The programmer's apprentice: A research overview *Computer 21*, 11 (Nov.), 10–25.

ROSENDAHL, M. AND MANKWALD, K. P. 1979. Analysis of programs by reduction of their

structure. In *Graph-Grammars and Their Applications to Computer Science and Biology*. Lecture Notes in Computer Science, vol. 73, Springer-Verlag, New York, 409–417.

THINKING MACHINES CORPORATION. 1987. Connection Machine Model CM-2 technical summary. Tech. Rep. HA87-4. Thinking Machines Corp., Cambridge, Mass.

WOLFE, M. J. 1989. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, Mass.

Received May 1990; revised October 1993; accepted October 1993