Scalar Replacement in the Presence of Conditional Control Flow

Steve Carr
Ken Kennedy

CRPC-TR92283
November 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Scalar Replacement in the Presence of Conditional Control Flow

Steve Carr
Ken Kennedy

SUMMARY

Most conventional compilers fail to allocate array elements to registers because standard data-flow analysis treats arrays like scalars, making it impossible to analyze the definitions and uses of individual array elements. This deficiency is particularly troublesome for floating-point registers, which are most often used as temporary repositories for subscripted variables.

In this paper, we present a source-to-source transformation, called *scalar replacement*, that finds opportunities for reuse of subscripted variables and replaces the references involved by references to temporary scalar variables. The objective is to increase the likelihood that these elements will be assigned to registers by the coloring-based register allocators found in most compilers. We extend our previous technique for scalar replacement to allow the presence of forward conditional control flow within loop bodies by mapping partial redundancy elimination to scalar replacement. Finally, we present experimental results showing that these techniques are extremely effective—capable of achieving integer factor speedups over code generated by good optimizing compilers of conventional design.

Keywords: Compilers, Optimization, Register Allocation

INTRODUCTION

Although conventional compilation systems do a good job of allocating scalar variables to registers, their handling of subscripted variables leaves much to be desired. Most compilers fail to recognize even the simplest opportunities for reuse of subscripted variables. For example, in the code shown below,

```
      DO 10 I = 1, N
         DO 10 J = 1, M
10          A(I) = A(I) + B(J)
```

most compilers will not keep `A(I)` in a register in the inner loop. This happens in spite of the fact that standard optimization techniques are able to determine that the address of the subscripted variable is invariant in the inner loop. On the other hand, if the loop is rewritten as

```
      DO 20 I = 1, N
         T = A(I)
         DO 10 J = 1, M
10          T = T + B(J)
20       A(I) = T
```

even the most naive compilers allocate `T` to a register in the inner loop.

The principal reason for the problem is that the data-flow analysis used by standard compilers is not powerful enough to recognize most opportunities for reuse of subscripted variables. Subscripted variables are treated in a particularly naive fashion, if at all, making it impossible to determine when a specific element might be reused. This is particularly problematic for floating-point register allocation because most of the computational quantities held in such registers originate in subscripted arrays.

Research supported by NSF Grant CCR-9120008 and by DARPA through ONR Grant N00014-91-J-1989.

For the past decade, the PFC and ParaScope Projects at Rice University have been using the theory of data dependence to recognize parallelism in Fortran programs.[1,5] Since data dependences arise from the reuse of memory cells, it is natural to speculate that dependence might be used for register allocation. This speculation led to methods for the allocation of vector registers, which led to scalar register allocation via the observation that scalar registers are vector registers of length 1.[2,4,3] In this paper, we extend the theory developed in our earlier works and report on experiments with a transformation system, based on ParaScope, that improves register allocation by rewriting subscripted variables as scalars, as illustrated by the example above.[5]

BACKGROUND

In this section, we will describe a transformation, called *scalar replacement* that can be applied to program source to improve the performance of compiled code. We assume that the target machine has a typical optimizing program compiler, one that performs scalar optimizations only. In particular, we assume that it performs strength reduction, allocates registers globally (via some coloring scheme) and schedules the instruction pipelines. This makes it possible for our preprocessor to restructure the loop nests, while leaving the details of optimizing the loop code to the compiler. These assumptions proved to be valid for the our experiments on the IBM RS/6000.

We begin the description of our method in the first section with a discussion of the special form of dependence graph that we use in subsequent transformations, paying special attention to the differences between this form and the standard dependence graph. Next, we show why previous algorithms for scalar replacement are inadequate when conditional control flow is present in inner loops. Finally, we show that partial redundancy elimination has properties that address the deficiencies in scalar replacement.

Dependence Graph

We say that a *dependence* exists between two references if there exists a control-flow path from the first reference to the second and both references access the same memory location.[9] The dependence is

- a *true dependence* or *flow dependence* if the first reference writes to the location and the second reads from it,
- an *antidependence* if the first reference reads from the location and the second writes to it,
- an *output dependence* if both references write to the location, and
- an *input dependence* if both references read from the location.

For the purposes of improving register allocation, we concentrate on the true and input dependences, each of which represents an opportunity to eliminate a load. In addition, output dependences can be helpful in determining when it is possible to eliminate a store.

A dependence is said to be *carried* by a particular loop if the references at the source and sink of the dependence are on different iterations of the loop. If both the source and sink occur in the same iteration, the dependence is *loop independent*. The *threshold* of a loop-carried dependence, τ_e, is the number of loop iterations between the source and sink. In determining which dependences represent reuse, we consider only those that have a *consistent threshold* — that is, those dependences for which the threshold is constant throughout the execution of the loop.[7,4] For a dependence to have a consistent threshold, it must be the case that the location accessed by the dependence source on iteration i is accessed by the sink on iteration $i + c$, where c does not vary with i.

Motivation

Although previous algorithms for scalar replacement have been shown to be effective, they have only handled loops without conditional-control flow.[3] The principle reason for past deficiencies is the reliance solely upon dependence information. A dependence contains little information concerning control flow between its source and sink. It only reveals that both statements *may* be executed. In the loop,

```
      DO 10 I = 1,N
 5      IF (M(I) .LT. 0) A(I) = B(I) + C(I)
10      D(I) = A(I) + E(I)
```

the true dependence from `A(I)` in statement `5` to `A(I)` in statement `10` does not reveal that the definition of `A(I)` is conditional. Using only dependence information, previous scalar replacement algorithms would produce the following incorrect code.

```
      DO 10 I = 1,N
         IF (M(I) .LT. 0) THEN
 5          A0 = B(I) + C(I)
            A(I) = A0
         ENDIF
10       D(I) = A0 + E(I)
```

If the result of the predicate were false, no definition of `A0` would occur, resulting in an incorrect value for `A0` at statement `10`. To ensure `A0` has the proper value, we can insert a load of `A0` from `A(I)` on the false branch, as shown below.

```
      DO 10 I = 1,N
         IF (M(I) .LT. 0) THEN
 5          A0 = B(I) + C(I)
            A(I) = A0
         ELSE
            A0 = A(I)
         ENDIF
10       D(I) = A0 + E(I)
```

The hazard with inserting instructions is the potential to increase run-time costs. In the previous example, we have avoided the hazard. If the true branch is taken, one load of `A(I)` is removed. If the false branch is taken, one load of `A(I)` is inserted and one load is removed. It is a requirement of our scalar replacement algorithm to prevent an increase in run-time accesses to memory.

Partial Redundancy Elimination

In the elimination of partial redundancies, the goal is to remove the latter of two identical computations that are performed on a given execution path. A computation is partially redundant when there may be paths on which both computations are performed and paths on which only the latter computation is performed. In Figure 1, the expression `A+B` is redundant along one branch of the `IF` and not redundant along the other. Partial redundancy elimination will remove the computation `C=A+B`, replacing it with an assignment, and insert a computation of `A+B` on the path where the expression does not appear (see Figure 2). Because there may be no basic block in which new computations can be inserted on a particular path, insertion can be done on flow-graph edges and new basic blocks are created when necessary.[6]

The essential property of this transformation is that it is guaranteed not to increase the number of computations performed along any path.[10] In mapping partial redundancy elimination to scalar replacement, references to array expression can be seen as the computations. A load or a store followed by another load

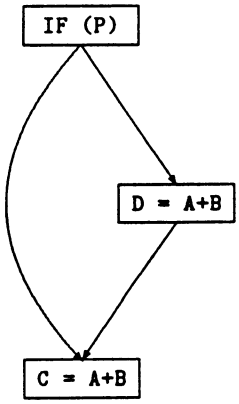

Figure 1 Partially Redundant Computation

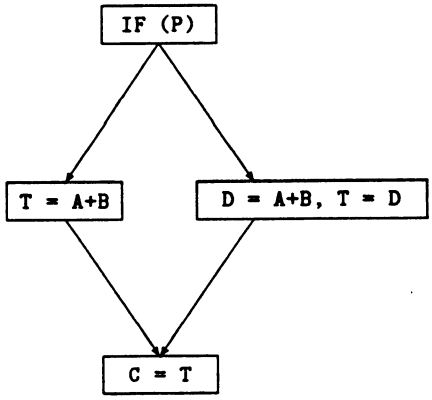

Figure 2 After Partial Redundancy Elimination

from the same location represents a redundant load that can be removed. Thus, using this mapping will guarantee that the number of memory accesses in a loop will not increase. However, we do not guarantee that a minimal number of loads will be inserted.

ALGORITHM

In this section, we present an algorithm for scalar replacement in the presence of forward conditional-control flow. We begin by determining which array accesses provide values for a particular array reference. Next, we link together references that share values by having them share temporary names. Finally, we generate scalar replaced code. For partially redundant array accesses, partial redundancy elimination is used to ensure that loads are inserted on the proper paths to make the array accesses fully redundant.

Control-Flow Analysis

Our optimization strategy focuses on loops; therefore, we can restrict control-flow analysis to loop nests only. Furthermore, it usually makes no sense to hold values across iterations of outer loops for two reasons.

1. There may be no way to determine the number of registers needed to hold all the values accessed in the innermost loop because of symbolic loop bounds.
2. Even if we know the register requirement, it is doubtful that the target machine will have enough registers.

Hence, we need only perform control-flow analysis on each innermost loop body.

To simplify our analysis, we impose a few restrictions on the control-flow graph. First, the flow graph of the innermost loop must be reducible. Second, backward jumps are not allowed within the innermost DO-loop because they potentially create loops. Finally, multiple loop exits are prohibited. This final restriction is for simplicity and can be removed with slight modification to the algorithm.

Availability Analysis

The first step in performing scalar replacement is to calculate *available array expressions*. Here, we will determine if the value provided by the source of a dependence is generated on every path to the sink of the dependence. Because values that cross iterations of the innermost loop between reuse points will not be available upon entry to the loop body, we assume that enough iterations of the loop have been peeled to make loop-carried values available upon entry to the loop. Since each lexically identical array reference accesses the same memory location on a given loop iteration, we do not treat each lexically identical array reference as a separate array expression. Rather, we consider them in concert.

Array expressions most often contain references to induction variables. Therefore, their naive treatment in availability analysis is inadequate. To illustrate this, in the loop,

```
      DO 30 I = 1,N
        IF (B(I) .LT. 0.0) THEN
10        C(I) = A(I) + D(I)
        ELSE
20        C(I) = A(I-1) + D(I)
        ENDIF
30      E(I) = C(I) + A(I)
```

the value accessed by the array expression `A(I)` is fully available at the reference to `A(I-1)` in statement 20, but it is not available at statement 10 and is only partially available at statement 30. Using a completely syntactic notion of array expressions, essentially treating each lexically identical expression as a scalar, `A(I)`

will be incorrectly reported as available at statements 10 and 30. Thus, more information is required. We must account for the fact that the value of an induction variable contained in an array expression changes on each iteration of the loop.

A convenient solution is to split the problem into loop-independent availability, denoted LIAV, where the back edge of the loop is ignored, and loop-carried availability, LCAV, where the back edge is included. Thus, an array expression is only available if it is in LIAV and there is a consistent incoming loop-independent dependence, or if it is in LCAV and there is a matching consistent incoming loop-carried dependence. The data-flow equations for availability analysis are shown below.

$$\begin{aligned} \text{LIAVIN(B)} &= \bigcap_{p \in preds\,(\text{B})} \text{LIAVOUT}(p) \\ \text{LIAVOUT(B)} &= (\text{LIAVIN(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)} \\ \text{LCAVIN(B)} &= \bigcap_{p \in preds\,(\text{B})} \text{LCAVOUT}(p) \\ \text{LCAVOUT(B)} &= \text{LCAVIN(B)} \bigcup \text{LCGEN(B)} \end{aligned}$$

For LIAV, an array expression is added to GEN when it is encountered whether it is a load or a store. At each store, the sources of all incoming inconsistent dependences are added to KILL and removed from GEN. At loads, nothing is done for KILL because a previously generated value cannot be killed. We call these sets LIGEN and LIKILL.

For LCAV, we must consider the fact that the flow of control from the source to the sink of a dependence will include at least the next iteration of the loop. Subsequent loop iterations can effect whether a value has truly been generated and not killed by the time the sink of the dependence is reached. In the loop,

```
      DO 10 I = 1,N
        IF (A(I) .GT. 0.0) THEN
          C(I) = B(I-2) + D(I)
        ELSE
          B(K) = C(I) + D(I)
        ENDIF
10      B(I) = E(I) + C(I)
```

the value generated by B(I) on iteration I=1 will be available at the reference to B(I-2) on iteration I=3 only if the false branch of the IF statement is not taken on iteration I=2. Since determining the direction of a branch at compile time is undecidable in general, we must assume that the value generated by B(I) will be killed by the definition of B(K). In general, any definition that is the source of an inconsistent output dependence can never be in LCGEN(B), $\forall$ B. It will always be killed on the current or next iteration of the loop. Therefore, we need only compute the LCGEN and not LCKILL.

There is one special case where this definition of LCGEN will unnecessarily limit the effectiveness of scalar replacement. When a dependence threshold is 1, it may happen that the sink of the generating dependence edge occurs before the killing definition. Consider the following loop.

```
      DO 10 I = 1,N
        B(K) = B(I-1) + D(I)
10      B(I) = E(I) + C(I)
```

the value used by B(I-1) that is generated by B(I) will never be killed by B(K). The solution to this limitation is to create a new set called LCAVIF1 that contains availability information only for loop-carried dependences with a threshold of 1. Control flow through the current and next iterations of the loop is included when computing this set. The data-flow equations for LCAVIF1 are identical to those of LIAV. This is because we consider control flow on the next iteration of the loop, unlike LCAV. Below are the data-flow equations for LCAVIF1.

$$\begin{aligned} \text{LCAVIF1IN(B)} &= \bigcap_{p \in preds\,(\text{B})} \text{LCAVIF1OUT}(p) \\ \text{LCAVIF1OUT(B)} &= (\text{LCAVIF1IN(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)} \end{aligned}$$

Because we consider both fully and partially redundant array accesses, we need to compute partial availability in order to scalar replace references whose load is only partially redundant. As in full-availability analysis, we partition the problem into loop-independent, loop-carried and loop-carried-if-1 sets. Computation of KILL and GEN corresponds to that of availability analysis. Below are the data-flow equations for partially available array expression analysis.

$$\begin{aligned}
\text{LIPAVIN}(B) &= \bigcup_{p \in preds\,(B)} \text{LIPAVOUT}(p) \\
\text{LIPAVOUT}(B) &= (\text{LIPAVIN}(B) - \text{LIKILL}(B)) \bigcup \text{LIGEN}(B) \\
\text{LCPAVIN}(B) &= \bigcup_{p \in preds\,(B)} \text{LCPAVOUT}(p) \\
\text{LCPAVOUT}(B) &= \text{LCPAVIN}(B) \bigcup \text{LCGEN}(B) \\
\text{LCPAVIF1IN}(B) &= \bigcup_{p \in preds\,(B)} \text{LCPAVIF1OUT}(p) \\
\text{LCPAVIF1OUT}(B) &= (\text{LCPAVIF1IN}(B) - \text{LIKILL}(B)) \bigcup \text{LIGEN}(B)
\end{aligned}$$

Reachability Analysis

Since there may be multiple lexically identical array references within a loop, we want to determine which references actually supply values that reach a sink of a dependence and which supply values that are killed before reaching such a sink. In other words, which values *reach* their potential reuses. In computing reachability, we do not treat each lexically identical array expression in concert. Rather, each reference is considered independently. Reachability information along with availability is used to select which array references provide values for scalar replacement. While reachability information is not required for correctness, it can prevent the marking of references as providing a value for scalar replacement when that value is redefined by a later identical reference. This improves the readability of the transformed code.

We partition the reachability information into three sets: one for loop-independent dependences (LIRG), one for loop-carried dependences with a threshold of 1 (LCRGIF1) and one for other loop-carried dependences (LCRG). Calculation of LIGEN and LIKILL is the same as that for availability analysis except in one respect. The source of any incoming loop-independent dependence whose sink is a definition is killed whether the threshold is consistent or inconsistent. Additionally, LIKILL is subtracted from LCRGOUT to account for consistent references that redefine a value on the current iteration of a loop. For example, in the following loop, the definition of `B(I)` kills the load from `B(I)`, therefore, only the definition reaches the reference to `B(I-1)`.

```
      DO 10 I = 1,N
        A(I) = B(I-1) + B(I)
10      B(I) = E(I) + C(I)
```

Using reachability information, the most recent access to a value can be determined. Even using LIKILL when computing LCRG will not eliminate all unreachable references. References with only outgoing consistent loop-carried output or antidependences will not be killed. This does not effect correctness, rather only readability, and can only happen when partially available references provide the value to be scalar replaced. Below are the data-flow equations used in computing array-reference reachability.

$$\begin{aligned}
\text{LIRGIN}(B) &= \bigcup_{p \in preds\,(B)} \text{LIRGOUT}(p) \\
\text{LIRGOUT}(B) &= (\text{LIRGIN}(B) - \text{LIKILL}(B)) \bigcup \text{LIGEN}(B) \\
\text{LCRGOUT}(B) &= \bigcup_{p \in preds\,(B)} \text{LCRGOUT}(P) \\
\text{LCRGOUT}(B) &= (\text{LCRGIN}(B) - \text{LIKILL}(B)) \bigcup \text{LCGEN}(B) \\
\text{LCRGIF1IN}(B) &= \bigcup_{p \in \text{PREDS}\,(B)} \text{LCRGIF1OUT}(P) \\
\text{LCRGIF1OUT}(B) &= (\text{LCRGIF1IN}(B) - \text{LIKILL}(B)) \bigcup \text{LIGEN}(B)
\end{aligned}$$

Potential-Generator Selection

At this point, we have enough information to determine which array references potentially provide values to the sinks of their outgoing dependence edges. We call these references *potential generators* because they can be seen as generating the value used at the sink of some outgoing dependence. The dependences leaving a generator are called *generating dependences*. Generators are only potential at this point because we need more information to determine if scalar replacement will even be profitable.

We have two goals in choosing potential generators. The first is to insert the fewest number of loads, correlating to the maximum number of memory accesses removed. The second is to minimize register pressure, or the number of registers required to eliminate the loads. To meet the first objective, fully available expressions are given the highest priority in generator selection. To meet the second, loop-independent fully available generators are preferred because they require the fewest number of registers. If no loop-independent generator exists, loop-carried fully available generators are considered next. If there are multiple such generators, the ones that require the fewest registers (having the smallest threshold) are chosen.

If there are no fully available generators, partially available array expressions are next considered as generators. Partially available generators do not guarantee a reduction in the number of memory accesses because memory loads need to be inserted on paths along which a value is needed but not generated. However, we can guarantee that there will not be an increase in the number of memory loads by only inserting load instructions if they are guaranteed to have a corresponding load removal on any execution path.[10] Without this guarantee, we may increase the number of memory accesses at execution time, resulting in a performance degradation.

The best choice for a partially available generator would be one that is loop-independent. Although there may be a "more available" loop-carried generator, register pressure is kept to a minimum and scalar replacement will be applied if we chose a loop-independent generator. If there are no loop-independent partially available array expressions, then the next choice would be a loop-carried partially available array expression with a generating dependence having the largest threshold of any incoming potential generating dependence. Although choosing the largest threshold contradicts the goal of keeping the register pressure to a minimum, we increase the probability that there will be a use of the value on every path by increasing the window size for potential uses of the generated value. We have chosen to sacrifice register pressure for potential savings in memory accesses.

Finally, when propagating data-flow sets through a basic block to determine availability or reachability at a particular point, information is not always incrementally updated. For loop-independent information, we update a data-flow set with GEN and KILL information as we encounter statements. However, the same is not true for loop-carried and loop-carried-if-1 information. GEN information is not used to update any loop-carried data-flow set. Loop-carried information must propagate around the loop to be valid and a loop-carried data-flow set at the entry to a basic block already contains this information. KILL information is only incrementally updated for loop-carried-if-1 sets since flow through the second iteration of a loop after a value is generated is considered.

In the next portion of the scalar replacement algorithm, we will ensure that the value needed at a reference to remove its load is fully available. This will involve insertion of memory loads for references whose generator is partially available. We guarantee through our partial redundancy elimination mapping that we will not increase the number of memory accesses at run time. However, we do not guarantee a minimal insertion of memory accesses.

Anticipability Analysis

After determining potential generators, we need to locate the paths along which loads need to be inserted to make partially available generators fully available. Loads need to be inserted on paths along which a value is needed but not generated. We have already encapsulated value generation in availability information. Now, we encapsulate value need with *anticipability*. The value generated by an array expression, v, is anticipated by an array expression w if there is a true or input edge $v \rightarrow w$ and v is w's potential generator.

```
      DO 6 I = 1,N
        IF (A(I) .GT. 0.0)
5         B(I) = C(I) + D(I)
        ELSE
          F(I) = C(I) + D(I)
        ENDIF
6       C(I) = E(I) + B(I)
```

In the above example, the value generated by the definition of `B(I)` in statement 5 is anticipated at the use of `B(I)` in statement 6.

As in availability analysis, we consider each lexically identical array expression in concert. We split the problem, but this time into only two partitions: one for loop-independent generators, LIAN, and one for loop-carried generators, LCAN. We do not consider the back edge of the loop during analysis for either partition. For LIAN, the reason is obvious. For LCAN, we only want to know that a value is anticipated on all paths through the loop. This is to ensure that we do not increase the number of memory accesses in the loop. In each partition, an array expression is added to GEN at an array reference if it is a potential generator for that reference. For members of LIAN, array expressions are killed at the point where they are defined by a consistent or inconsistent reference. For LCAN, only inconsistent definitions kill anticipation because consistent definitions do not define the value being anticipated on the current iteration. For example, in the loop

```
    DO 1 I = 1,N
      A(I) = B(I) + D(I)
1     B(I) = A(I-1) + C(I)
```

the definition of `A(I)` does not redefine a particular value anticipated by `A(I-1)`. The value is generated by `A(I)` and then never redefined because of the iteration change.

$$\begin{aligned}
\text{LIANOUT(B)} &= \bigcap_{s \in succs\,(\text{B})} \text{LIANIN}(s) \\
\text{LIANIN(B)} &= (\text{LIANOUT(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)} \\
\text{LCANOUT(B)} &= \bigcap_{s \in succs\,(\text{B})} \text{LCANIN}(s) \\
\text{LCANIN(B)} &= \text{LCANOUT(B)} - \text{LCKILL(B)} \bigcup \text{LCGEN(B)}
\end{aligned}$$

Dependence-Graph Marking

Once anticipability information has been computed, the dependence graph can be marked so that only dependences to be scalar replaced are left unmarked. The other edges no longer matter because their participation in value flow has already been considered. Figure 3 shows the algorithm *MarkDependenceGraph*.

At a given array reference, we mark any incoming true or input edge that is inconsistent and any incoming true or input edge that has a symbolic threshold or has a threshold greater than that of the dependence edge from the potential generator. Inconsistent and symbolic edges are not amenable to scalar replacement because it is impossible to determine the number of registers needed to expose potential reuse at compile time. When a reference has a consistent generator, all edges with threshold less than or equal to the generating

```
Procedure MarkDependenceGraph(G)

Input: G = (V, E), the dependence graph

for each v ∈ V
  if v has no generator then
    mark v's incoming true and input edges
  else if v's generator is inconsistent or symbolic d_n then
      mark v's incoming true and input edges
      v no longer has a generator
  else if v's generator is LCPAV and v ∉ LCANTIN(ENTRY)
      mark v's incoming true and input edges
      v no longer has a generator
  else
    τ_v = threshold of edge from v's generator
    mark v's incoming true and input edges with d_n > τ_v
    mark v's incoming edges whose source does not reach v or
      whose source is not partially available at v
  mark v's incoming inconsistent and symbolic edges
```

Figure 3 MarkDependenceGraph

threshold are left unmarked in the graph. This is to facilitate the consistent register naming discussed in subsequent sections. It ensures that any reference occurring between the source and sink of an unmarked dependence that can provide the value at the sink will be connected to the dependence sink. Finally, any edge from a loop-carried partially available generator that is not anticipated at the entry to the loop is removed because there will not be a dependence sink on every path.

Name Partitioning

At this point, the unmarked dependence graph represents the flow of values for references to be scalar replaced. We have determined which references provide values that can be scalar replaced. Now, we move on to linking together the references that share values. Consider the following loop.

```
      DO 6 I = 1,N
5         B(I) = B(I-1) + D(I)
6         C(I) = B(I-1) + E(K)
```

After performing the analysis discussed so far, the generator for the reference to `B(I-1)` in statement 6 would be the load of `B(I-1)` in statement 5. However, the generator for this reference is the definition of `B(I)` in statement 5 making `B(I-1)` in statement 5 an intermediate point in the flow of the value rather than the actual generator for `B(I-1)` in statement 6. These references need to be considered together when generating temporary names because they address the same memory locations.

The nodes of the dependence graph can be partitioned into groups by dependence edges (see Figure 4) to make sure that temporary names for all references that participate in the flow of values through a memory location are consistent. Any two nodes connected by an unmarked dependence edge after graph marking belong in the same partition. Partitioning is accomplished by performing a traversal of the dependence

graph, following unmarked true and input dependences only since these dependences represent value flow. Partitioning will tie together all references that access a particular memory location and represent reuse of a value in that location.

After name partitioning is completed, we can determine the number of temporary variables (or registers) that are necessary to perform scalar replacement on each of the partitions. To calculate register requirements, we split the references (or partitions) into two groups: variant and invariant. Variant references contain the innermost-loop induction variable within their subscript expression. Invariant references do not contain the innermost-loop induction variable. In the previous example, the reference to `E(K)` is invariant with respect to the `I`-loop while all other array references are variant.

For variant references, we begin by finding all references within each partition that are first to access a value on a given path from the loop entry. We call these references *partition generators*. A variant partition generator will already have been selected as a potential generator and will not have incoming unmarked dependences from another reference within its partition. The set of partition generators within one partition is called the generator set, γ_p. Next, the maximum over all dependence distances from a potential generator to a reference within this partition is calculated. Even if dependence edges between the generator and a reference have been removed from the graph, we can compute the distance by finding the length of a chain, excluding cycles, to that reference from the generator. Letting τ_p be the maximum distance, each partition requires $r_p = \tau_p + 1$ registers or temporary variables: one register for the value generated on the current iteration and one register for each of the τ_p values that were generated previously and need to flow to the last sink. In the previous example, `B(I)` is the oldest reference and is the generator of the value used at both references to `B(I-1)`. The threshold of the partition is 1. This requires 2 registers because there are two values generated by `B(I)` that are live after executing statement 5 and before executing statement 6.

For invariant references, we can use one register for scalar replacement because each reference accesses the same value on every iteration. In order for an invariant potential generator to be a partition generator, it must be a definition or the first load of a value along a path through the loop.

Register-Pressure Moderation

Unfortunately, experiments have shown that exposing all of the reuse possible with scalar replacement may result in a performance degradation.[3] Register spilling can completely counteract any savings from scalar replacement. The problem is that specialized information is necessary to recover the original code to prevent excessive spilling. As a result, we need to generate scalar temporaries in such a way as to minimize register pressure, in effect doing part of the register allocator's job.

Our goal is to allocate temporary variables so that we can eliminate the most memory accesses given the available number of machine registers. This can be approximated using a greedy algorithm. In this approximation scheme, the partitions are ordered in decreasing order based upon the ratio of benefit to cost, or in this case, the ratio of memory accesses saved to registers required. At each step, the algorithm chooses the first partition that fits into the remaining registers. This method requires $O(n \log n)$ time to sort the ratios and $O(n)$ time to select the partitions. Hence, the total running time is $O(n \log n)$.

To compute the benefit of scalar replacing a partition, we begin by computing the probability that each basic block will be executed. The first basic block in the loop is assigned a probability of execution of 1. Each outgoing edge from the basic block is given a proportional probability of execution. In the case of two outgoing edges, each edge is given a 50% probability of execution. For the remaining blocks, this procedure is repeated, except that the probability of execution of a remaining block is the sum of the probabilities of its incoming edges. Next, we compute the probability that the generator for a partition is available at the entrance to and exit from a basic block. The probability upon entry is the sum of the probabilities at the exit

```
Procedure GenerateNamePartitions(G, P)

Input: G = (V, E), the dependence graph
       P = the set of name partitions

  ∀v ∈ V mark v as unvisited
  i = 0
  for each v ∈ V
    if v is unvisited then
      put v in P_i
      Partition(v, P_i)
      i = i + 1
  for each p ∈ P do
    FindGenerator(p, γ_p)
    if p is invariant then
      r_p = 1
    else
      r_p = max_{g∈γ_p}(CalcNumberOfRegisters(g) + 1)
  enddo
end

Procedure Partition(v, p)
  mark v as visited
  add v to p
  for each (e = (v, w) ∨ (w, v)) ∈ E
    if e is true or input and unmarked and w not visited then
      Partition(w, p)
end

Procedure FindGenerator(p, γ)
  for each v ∈ p that is a potential generator
    if (v is invariant ∧ (v is a store ∨
        v has no incoming unmarked loop-independent edge)) ∨
        (v is variant ∧ v has no unmarked incoming true or
        input dependences from another w ∈ p) then
      γ = γ ∪ v
end

Procedure CalcNumberOfRegisters(v)
  dist = 0
  for each e = (v, w) ∈ E
    if e is true or input and unmarked ∧ w not visited ∧
        P(w) = P(v) then
      dist = max(dist, d_n(e)+ CalcNumberOfRegisters(w))
  return(dist)
end
```

Figure 4 GenerateNamePartitons

of a block's predecessors weighted by the number of incoming edges. Upon exit from a block, the probability is 1 if the generator is available, 0 if it is not available and the entry probability otherwise. After computing each of these probabilities, the benefit for each reference within a partition can be computed by multiplying the execution probability for the basic block that contains the reference by the availability probability of the references generator. Figure 5 gives the complete algorithm for computing benefit. As an example of benefit, in the loop,

```
   DO 1 I = 1,N
      IF (M(I) .LT. 0) THEN
         A(I) = B(I) + C(I)
      ENDIF
1     D(I) = A(I) + E(I)
```

the load of `A(I)` in statement 1 has a benefit of 0.5.

Unfortunately, the greedy approximation can produce suboptimal results. To see this, consider the following example, in which each partition is represented by a pair (n, m), where n is the number of registers needed and m is the number of memory accesses eliminated. Assume that we have a machine with 6 registers and that we are generating temporaries for a loop that has the following generators: $(4,8), (3,5), (3,5), (2,1)$. The greedy algorithm would first choose $(4,8)$ and then $(2,1)$, resulting in the elimination of nine accesses instead of the optimal ten.

To get a possibly better allocation, we can model register-pressure moderation as a knapsack problem, where the number of scalar temporaries required for a partition is the size of the object to be put in the knapsack, and the size of the register file is the knapsack size. Using dynamic programming, an optimal solution to the knapsack problem can be found in $O(kn)$ time, where k is the number of registers available for allocation and n is the number of generators. Hence, for a specific machine, we get a linear time bound. However, with the greedy method, we have a running time that is independent of machine architecture, making it more practical for use in a general tool. The greedy approximation of the knapsack problem is also provably no more than two times worse than the optimal solution.[8] Additionally, our experiments suggest that in practice the greedy algorithm performs as well as the knapsack algorithm.

After determining which generators will be fully scalar replaced, there may still be a few registers available. Those partitions that were eliminated from consideration can be examined to see if partial allocation is possible. In each eliminated partition whose generator is not LCPAV, we allocate references whose distance from γ_p is less than the number of remaining registers. All references within p that do not fit this criterium are removed from p. This step is performed on each partition, if possible, while registers remain unused. Finally, since we have possibly removed references from a partition, anticipability analysis for potential generators must be redone.

To illustrate partial allocation, assume that in the following loop there is one register available.

```
     DO 10 I = 1,N
        A(I) = ...
        IF (B(I) .GT. 0.0) THEN
           ... = A(I)
        ENDIF
10      ... = A(I-1)
```

Here, full allocation is not possible, but there is a loop-independent dependence between the `A(I)`'s. In partial allocation, `A(I-1)` is removed from the partition allowing scalar replacement to be performed. The algorithm in Figure 6 gives the complete register-pressure moderation algorithm, including partial allocation.

Procedure CalculateBenefit(P, FG)

Input: P = set of reference partitions
FG = flow graph
Defs: $A_e^p(\text{B})$ = probability γ_p is available on entry to B
$A_x^p(\text{B})$ = probability γ_p is available on exit from B
b_p = benefit for partition p
E_{FG} = edges in flow graph
$\mathcal{P}(\text{B})$ = probability block B will be executed

$\mathcal{P}(\text{ENTRY}) = 1$
n =# outgoing forward edges of ENTRY
weight each outgoing forward edge of ENTRY at $\frac{\mathcal{P}(\text{ENTRY})}{n}$
for the remaining basic blocks B$\in FG$ in reverse
depth-first order
let $\mathcal{P}(\text{B})$= sum of all incoming edge weights
n =# outgoing forward edges of B
weight each outgoing forward edge at $\frac{\mathcal{P}(\text{B})}{n}$
for each $p \in P$
for each basic block B$\in FG$ in reverse depth-first order
n =# incoming forward edges of B
for each incoming forward edge of B, e = (C,B)
$A_e^p(\text{B}) = A_e^p(\text{B}) + \frac{A_x^p(\text{C})}{n}$
if $\gamma_p \in$ LIAVOUT(B) $\bigcup$ LCAVOUT(B) then
$A_x^p(\text{B}) = 1$
else if $\gamma_p \in$ LIPAVOUT(B) $\bigcup$ LCPAVOUT(B) then
$A_x^p(\text{B}) = A_e^p(\text{B})$
else
$A_x^p(\text{B}) = 0$
for each $v \in p$
if γ_p is LCPAV or LCPAVIF1 then
$b_p = b_p + A_x^p(\text{EXIT}) \times \mathcal{P}(\text{B}_v)$
else if γ_p is LIPAV
$b_p = b_p + A_e^p(\text{B}) \times \mathcal{P}(\text{B}_v)$
else
$b_p = b_p + \mathcal{P}(\text{B}_v)$
end

Figure 5 CalculateBenefit

Procedure ModerateRegisterPressure(P, G)

Input: P = set of reference partitions
$G = (V, E)$, the dependence graph

Defs: b_p = benefit for a partition p
r_p = register required by a partition p

```
CalculateBenefit(P, G)
R = registers available
H = sort of P on ratio of b_p / r_p in decreasing order
for i = 1 to |H| do
  if r_{H_i} < R then
    allocate(H_i)
    R = R - r_{H_i}
  else
    S = S ∪ H_i
    P = P - H_i
  endif
enddo
if R > 0 then
  i = 1
  while i < |S| and R > 0 do
    while |S_i| > 0 and R > 0 do
      S_i = {v | v ∈ S_i, d_n(ê) < R, ê = (γ_{S_i}, v)}
      allocate(S_i)
      R = R - r_{S_i}
      P = P ∪ S_i
    enddo
  enddo
  redo anticipability analysis
 endif
end
```

Figure 6 ModerateRegisterPressure

Although this method of partial allocation may still leave possible reuses not scalar replaced, experience suggests this rarely, if ever, happens. One possible solution is to consider dependences from intermediate points within a partition when looking for potential reuse.

Reference Replacement

At this point, we have determined which references will be scalar replaced. We now move into the code generation phase of the algorithm. Here, we will replace array references with temporary variables and ensure that the temporaries contain the proper value at a given point in the loop.

After we have determined which partitions will be scalar replaced, we replace the array references within each partition. First, for each variant partition p, we create the temporary variables $T_p^0, T_p^1, \ldots, T_p^{r_p-1}$, where T_p^i represents the value generated by $g \in \gamma_p$ i iterations earlier, where g is the first generator to access the value used throughout the partition. Each reference within the partition is replaced with the temporary that coincides with its distance from g. For invariant partitions, each reference is replaced with T_p^0. If a replaced reference $v \in \gamma_p$ is a memory load, then a statement of the form $T_p^i = v$ is inserted before the generating statement. Requiring that the load must be in γ_p for load insertion ensures that a load that is a potential generator but also has a potential generator itself will not have a load inserted. The value for the potential generator not in γ_p will already be provided by its potential generator. If the v is a store, then a statement of the form $v = T_p^i$ is inserted after the generating statement. The latter assignment is unnecessary if it has an outgoing loop-independent edge to definition that is always executed and it has no outgoing inconsistent true dependences. We could get better results by performing availability and anticipability analysis exclusively for definitions to determine if a value is always redefined.

The effect of reference replacement will be illustrated on the following loop nest.

```
      DO 3 I = 1, 100
1       IF (M(I) .LT. 0) E(I) = C(I)
2       A(I) = C(I) + D(I)
3       B(K) = B(K) + A(I-1)
```

The reuse-generating dependences are:

1. A loop-independent input dependence from `C(I)` in statement 1 to `C(I)` in statement 2 (threshold 0), and
2. A true dependence from `A(I)` to `A(I-1)` (threshold 1).
3. A true dependence from `B(K)` to `B(K)` in statement 2 (threshold 1).

By our method, the generator `C(I)` in statement 1 needs only one temporary, `T10`. Here, we are using the first numeric digit to indicate the number of the partition and the second to represent the distance from the generator. The generator `B(K)` in statement 1 needs one temporary, `T20`, since it is invariant, and the generator `A(I)` needs two temporaries, `T30` and `T31`. When we apply the reference replacement procedure to the example loop, we generate the following code.

```
      DO 3 I = 1, 100
1       IF (M(I) .LT. 0) THEN
           T10  = C(I)
           E(I) = T10
        ENDIF
        T30  = T10 + D(I)
2       A(I) = T30
        T20  = T20 + T31
3       B(K) = T20
```

The value for `T31` is not generated in this example. We will discuss its generation in later sections.

Statement-Insertion Analysis

After we replace the array reference with scalar temporaries, we need to insert loads for partially available generators. Given a reference that has a partially available potential generator, we need to insert a statement at the highest point on a path from the loop entrance to the reference that anticipates the generator where the generator is not partially available and is anticipated. By performing statement-insertion analysis on potential generators, we guarantee that every reference's anticipated value will be fully available. Here, we handle each individual reference, whereas name partitioning linked together those references that share values. This philosophy will not necessarily introduce a minimum number of newly inserted loads, but there will not be an increase in the number of run-time loads. The place for insertion of loads for partially available generators can be determined using Drechsler and Stadel's formulation for partial redundancy elimination, as shown below.[6]

$$\text{PPIN(B)} = \text{ANTIN(B)} \bigcap \text{PAVIN(B)} \bigcap (\text{ANTLOC(B)} \bigcup (\text{TRANSP(B)} \bigcap \text{PPOUT(B)}))$$

$$\text{PPOUT(B)} = \begin{cases} \text{FALSE if B is the loop exit} \\ \bigcap_{s \in \text{SUCC(B)}} \text{PPIN}(s) \end{cases}$$

$$\text{INSERT(B)} = \text{PPOUT(B)} \bigcap \neg\text{AVOUT(B)} \bigcap (\neg\text{PPIN(B)} \bigcup \neg\text{TRANSP(B)})$$

$$\text{INSERT}_{\text{(A,B)}} = \text{PPIN(B)} \bigcap \neg\text{AVOUT(A)} \bigcap \neg\text{PPOUT(A)}$$

Here, PPIN(B) denotes placement of a statement is possible at the entry to a block and PPOUT(B) denotes placement of a statement is possible at the exit from a block. INSERT(B) determines which loads need to be inserted at the bottom of block B. $\text{INSERT}_{\text{(A,B)}}$ is defined for each edge in the control-flow graph and determines which loads are inserted on the edge from A to B. TRANSP(B) is true for some array expression if it is not defined by a consistent or inconsistent definition in the block B. ANTLOC(B) is the same as GEN(B) for anticipability information. Three problems of the above form are solved: one for LIPAV generators, one for LCPAV generator and one for LCPAVIF1 generators. Additionally, any reference to loop-carried ANTIN information refers to the entry block.

If $\text{INSERT}_{\text{(A,B)}}$ is true for some potential generator g, then we insert a load on the edge (A,B) of the form $T_p^d = g$, where T_p^d is the temporary name associated with g. If INSERT(B) is true for some potential generator g, then a statement of identical form is inserted at the end of block B. Finally, if $\text{INSERT}_{\text{(A,B)}}$ is true $\forall \text{A} \in \text{PRED(B)}$, then loads can be collapsed into the beginning of block B.

If we perform statement insertion on our example loop, we get the following results.

```
      DO 3 I = 1, 100
1       IF (M(I) .LT. 0) THEN
           T10  = C(I)
           E(I) = T10
        ELSE
           T10  = C(I)
        ENDIF
        T30  = T10 + D(I)
2       A(I) = T30
        T20  = T20 + T31
3       B(K) = T20
```

Again, the generation of T31 is left for later sections.

Register Copying

Next, we need to ensure that the values held in the temporary variables are correct across loop iterations. The value held in T_p^i needs to move one iteration further away from its generator, T_p^0, on each subsequent

loop iteration. Since i is the number of loop iterations the value is from the generator, the variable T_p^{i+1} needs to take on the value of T_p^i at the end of the loop body in preparation for the iteration change. For each p, the following sequence of transfers is inserted at the end of the loop body.

$$T_p^{r_p-1} = T_p^{r_p-2}$$
$$T_p^{r_p-2} = T_p^{r_p-3}$$
$$\cdots$$
$$T_p^1 = T_p^0$$

After inserting register copies, our example code becomes:

```
      DO 3 I = 1, 100
1        IF (M(I) .LT. 0) THEN
            T10  = C(I)
            E(I) = T10
         ELSE
            T10  = C(I)
         ENDIF
         T30  = T10 + D(I)
2        A(I) = T30
         T20  = T20 + T31
         B(K) = T20
3        T31  = T30
```

Code Motion

It may be the case that the assignment to or a load from a generator may be moved entirely out of the innermost loop. This is possible when the reference to the generator is invariant with respect to the innermost loop. In the example above, `B(K)` does not change with each iteration of the `I`-loop; therefore, its value can be kept in a register during the entire execution of the loop and stored back into `B(K)` after the loop exit.

```
      DO 3 I = 1, 100
1        IF (M(I) .LT. 0) THEN
            T10  = C(I)
            E(I) = T10
         ELSE
            T10  = C(I)
         ENDIF
         T30  = T10 + D(I)
2        A(I) = T30
         T20  = T20 + T31
3        T31  = T30
      B(K) = T20
```

When inconsistent dependences leave an invariant array reference that is a store, the generating store for that variable cannot be moved outside of the innermost loop. Consider the following example.

```
      DO 10 J = 1, N
10       A(I) = A(I) + A(J)
```

The true dependence from `A(I)` to `A(J)` is not consistent. If the value of `A(I)` were stored into `A(I)` outside of the loop, then the value of `A(J)` would be wrong whenever `I=J` and `I > 1`.

Initialization

To ensure that the temporary variables contain the correct values upon entry to the loop, it is peeled using the algorithm in Figure 7. We peel $\max(r_{p_1}, \ldots, r_{p_n}) - 1$ iterations from the beginning of the loop, replacing the members of a variant partition p for peeled iteration k with their original array reference, substituting the iteration value for the induction variable, only if $j \geq k$ for a temporary T_p^j. For invariant partitions, we only replace non-partition generators' temporaries on the first peeled iteration. Additionally, we let $r_p = 2$ for each invariant partition when calculating the number of peeled iterations. This ensures that invariant partitions will be initialized correctly. Finally, at the end of each peeled iteration, the appropriate number of register transfers is inserted.

When this transformation is applied to our example, we get the following code.

```
IF (M(1) .LT. 0) THEN
   T10  = C(1)
   E(1) = T10
ELSE
   T10  = C(1)
ENDIF
T30  = T10 + D(1)
A(1) = T30
T20  = B(K) + A(0)
T31  = T30
    ...
LOOP BODY
```

Register Subsumption

In our example loop, we have eliminated three loads and one store from each iteration of the loop, at the cost of three register-to-register transfers in each iteration. Fortunately, inserted transfer instructions

---

Procedure Initialize(P,G)

Input: P = set of reference partitions
$G = (V, E)$, the dependence graph

$x = \max(r_{p_1}, \ldots, r_{p_2}) - 1$
for $k = 1$ to x
G' =peel of the kth iteration of G
for each $v \in V'$
if $v = T_p^j \wedge (v$ is variant $\wedge\, j \geq k) \vee$
(v is invariant $\wedge\, v \notin \gamma_{P(v)} \wedge j + 1 \geq k \wedge$
v's generator is loop carried) then
replace v with its original array reference
replace the inner loop induction variable with
its kth iteration
endif
InsertRegisterCopies(G', P, k)
end

Figure 7 Initialize

---

can be eliminated if we unroll the scalar replaced loop in the following manner. If we have the partitions $p_0, p_1, \ldots, p_n$, we can remove the transfers by unrolling $\text{lcm}(r_{p_0}, r_{p_1}, \ldots, r_{p_n}) - 1$ times. In the kth unrolled body, the temporary variable T_p^j is replaced with the variable $T_p^{\text{mod}(j-k, r_p)}$ where $\text{mod}(y, x) = y - \lfloor \frac{y}{x} \rfloor x$. Essentially we capture the permutation of values by circulating the register names within the unrolled iterations.

The final result of scalar replacement on our example is shown in Figure 8 (the pre-loop to capture the extra iterations and the initialization code are not shown).

EXPERIMENT

We have implemented a source-to-source translator in the ParaScope programming environment, a programming system for Fortran, that uses the dependence analyzer from PFC. The translator replaces subscripted variables with scalars using the described algorithm. The experimental design is illustrated in Figure 9. In this scheme, ParaScope serves as a preprocessor, rewriting Fortran programs to improve register allocation. Both the original and transformed versions of the program are then compiled and run using the standard product compiler for the target machine.

For our test machine, we chose the IBM RS/6000 model 540 because it had a good compiler and a large number of floating-point registers (32). In fact, the IBM XLF compiler performs scalar replacement for those references that do not require dependence analysis. Many fully available loop-independent and invariant cases are handled. Therefore, the results described here only reflect the cases that required dependence analysis. Essentially, we show the results of performing scalar replacement on loop-carried dependences and in the presence of inconsistent dependences.

```
      DO 3 I = 2, 100,2
1       IF (M(I) .LT. 0) THEN
           T10    = C(I)
           E(I)   = T10
        ELSE
           T10    = C(I)
        ENDIF
        T30  = T10 + D(I)
2       A(I) = T30
        T20    = T20  + T31
        IF (M(I+1) .LT. 0) THEN
           T10    = C(I+1)
           E(I+1)   = T10
        ELSE
           T20    = C(I+1)
        ENDIF
        T31  = T10 + D(I+1)
        A(I+1) = T31
3       T20    = T20  + T30
```

Figure 8 Example After Scalar Replacement

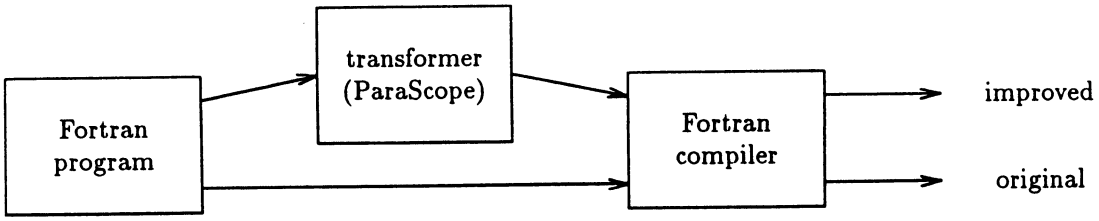

Figure 9 Experimental design.

Livermore Loops. We tested scalar replacement on a number of the Livermore Loops. Some of the kernels did not contain reuse detectable with dependence analysis; therefore, we do not show their results. In the table below, we show the performance gain attained by our transformation system.

Loop	Iterations	Original	Transformed	Speedup
1	10000	3.40s	2.54s	1.34
5	10000	3.05s	1.36s	2.24
6	10000	3.82s	2.82s	1.35
7	5000	3.94s	2.02s	1.95
8	10000	3.38s	3.07s	1.10
11	10000	4.52s	1.69s	2.67
12	10000	1.70s	1.42s	1.20
13	5000	3.25s	3.01s	1.08
18	1000	2.62s	2.54s	1.03
20	500	2.99s	2.90s	1.03
23	5000	2.68s	2.35s	1.14

Some of the interesting results include the performances of loops 5 and 11, which compute first order linear recurrences. Livermore Loop 5 is shown below.

```
      DO 5 I = 2,N
5       X(I) = Z(I) * (Y(I) - X(I-1))
```

Here, scalar replacement not only removes one memory access, but also improves pipeline performance. The store to `X(I)` will no longer cause the load of `X(I-1)` on the next iteration to block. Loop 11 presents a similar situation.

Loop 6, shown below, is also an interesting case because it involves an invariant array reference that requires dependence analysis to detect.

```
      DO 6 I= 2,N
        DO 6 K= 1,I-1
6         W(I)= W(I)+B(I,K)*W(I-K)
```

A compiler that recognizes loop invariant addresses to get this case, such as the IBM compiler, fails because of the load of `W(I-K)`. Through the use of dependence analysis, we are able to prove that there is no dependence between `W(I)` and `W(I-K)` that is carried by the innermost loop, allowing code motion. Because of this additional information, we are able to get a speedup of 1.35.

Linear Algebra Kernels. We also tested scalar replacement on both the point and block versions of LU decomposition with and without partial pivoting. In the table below, we show the results.

Kernel	Original	Transformed	Speedup
LU Decomp	6.76s	6.09s	1.11
Block LU	6.39s	4.40s	1.45
LU w/ Pivot	7.01s	6.35s	1.10
Block LU w/ Pivot	6.84s	4.81s	1.42

Each of these kernels contains invariant array references that require dependence analysis to detect. The speedup achieved on the block algorithms is higher because an invariant load and store are removed rather than just a load as in the point algorithms.

Applications. To complete our study we ran a number of Fortran applications through our translator. We chose programs from Perfect, RiCEPS and local sources. Of those programs that belong to the benchmark suites, but are not included in the experiment, 5 failed to be successfully analyzed by PFC, 1 failed to compile on the RS6000 and 10 contained no reuse opportunities for our algorithms. Most of those that contained no reuse opportunities for our algorithm had some loop-independent or loop-invariant reuse that was captured by the IBM XLF compiler. The table below contains a short description of each application.

Suite	Application	Description
Perfect	Adm	Pseudospectral Air Pollution
	Arc2d	2d Fluid-Flow Solver
	Flo52	Transonic Inviscid Flow
RiCEPS	Shal	Weather Prediction
	Simple	2d Hydrodynamics
	Sphot	Particle Transport
	Wave	Electromagnetic Particle Simulation
Local	CoOpt	Oil Exploration
	Seval	B-Spline Evaluation
	Sor	Successive Over-Relaxation

The results of performing scalar replacement on these applications is reported in the following table. Any application not listed observed a speedup of 1.00.

Suite	Program	Original	Transformed	Speedup
Perfect	Adm	236.84s	228.84s	1.03
	Arc2d	410.13s	407.57s	1.01
	Flo52	66.32s	63.83s	1.04
RiCEPS	Shal	302.03s	290.42s	1.04
	Simple	963.20s	934.13s	1.03
	Sphot	3.85s	3.78s	1.02
	Wave	445.94s	431.11s	1.03
Local	CoOpt	122.88s	120.44s	1.02
	Seval	0.62s	0.56s	1.11
	Sor	1.83s	1.26s	1.46

The applications Sor and Seval performed the best because we were able to optimize their respective computationally intensive loop. Each had one loop which comprised almost the entire running time of the program. For the program Simple, one loop comprised approximately 50% of the program execution time, but the carried dependence could not be exposed due to a lack of registers. In fact, without register-pressure minimization, program performance deteriorated. Sphot's improvement was gained by performing scalar replacement on one partition in one loop. This particular partition contained a loop-independent partially

available generator that required our extension to handle control flow. Although the codes in this study did not often contain inner-loop conditional control flow, other applications may be found that require the mapping of partial redundancy elimination.

The IBM RS/6000 has a load penalty of only 1 cycle. On processors with larger load penalties, such as the DEC Alpha, we would expect to see a larger performance gain through scalar replacement. Additionally, the problem sizes for benchmark are typically small. On larger problem sizes, we expect to see larger performance gains due to a higher percentage of time being spent inside of loops.

CONCLUSIONS

In this paper, we have presented a method to expose the reuse available in array expressions in innermost loops so that typical scalar optimizing compilers can allocate array values to registers. We have extended previous work by handling forward conditional-control flow within innermost loops. To handle values that are only partially available at their point of reuse, we have mapped partial redundancy elimination to scalar replacement to ensure that values are fully available.

We have implemented our algorithm for scalar replacement within the ParaScope programming environment. Experimentation with this implementation has shown that integer-factor speedups over code generated by quality optimizing compilers are possible. We believe that the effectiveness of scalar replacement, as demonstrated in our experiments, indicates that they should be included in every highly optimizing scalar compiler .

Given that future machine designs are certain to have increasingly complex memory hierarchies, compilers will need to adopt increasingly sophisticated strategies for managing the memory hierarchy so that programmers can remain free to concentrate on program logic. The transformations and experiments reported in this paper represent an encouraging first step in that direction. It is pleasing to note that the theory of dependence, originally developed for vectorizing and parallelizing compilers can be used to substantially improve the performance of modern scalar machines as well.

References

[1] J.R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In *Supercomputers: Design and Applications*, pages 186–205. IEEE Computer Society Press, Silver Spring, MD., 1984.

[2] J.R. Allen and K. Kennedy. Vector register allocation. Technical Report TR86-45, Department of Computer Science, Rice University, 1988.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5, 1988.

[5] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987.

[6] K.H. Drechsler and M.P. Stadel. A solution to a problem with morel and renvoise's "global optimization by suppression of partial redudancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[7] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, 1987.

[8] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[9] D. Kuck. *The Structure of Computers and Computations Volume 1.* John Wiley and Sons, New York, 1978.

[10] E. Morel and C. Revoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), February 1979.