to appear in the International Conference on Parallel Processing, Aug. 1996

COMPILER SUPPORT FOR PRIVATIZATION ON DISTRIBUTED-MEMORY MACHINES

Daniel J. Palermo, Ernesto Su, Eugene W. Hodges IV, and Prithviraj Banerjee
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{palermo, ernesto, ewhodges, banerjee}@crhc.uiuc.edu

ABSTRACT

The practice of using temporary scalar or array variables to store the results of common subexpressions presents several challenges to a parallelizing compiler. Not only does dependence analysis and, as a result, parallelization suffer, but existing techniques used for partitioning programs and generating communication for parallel execution on distributed-memory multicomputers also tend to break down. Techniques that have been developed over the years to compensate for this programming practice include scalar expansion, global forward substitution, and privatization, each of which has its own strengths and weaknesses. Compared to scalar expansion and global forward substitution, privatization has the advantage of not causing an increase in memory requirements or operation counts, but when compiling for distributed-memory machines it causes several new problems to arise. In this paper, we present a simple extension to a uniform array-region analysis framework that utilizes privatization information to partition loops and generate efficient communication, using the owner-computes rule, in the presence of temporary variables.

1. INTRODUCTION

Programmers often reduce the complexity of computations by using temporary scalar or array variables to store the values of common subexpressions. Not only does dependence analysis and, as a result, parallelization suffer, but existing techniques used for partitioning programs and generating communication for parallel execution on distributed-memory multicomputers also tend to break down. In order to improve the parallelization of programs which are written using this practice, several techniques have been developed over the years to help cope with these problems. These techniques include global forward substitution [1, 2], scalar expansion [1, 2, 3], and scalar [4] or array [5, 6] privatization.

Both scalar expansion and forward substitution have several drawbacks in terms of memory requirements as well as operation count while privatization suffers from none of these problems. When compiling for distributed-memory machines scalar expansion and forward substitution can be easily handled by existing techniques for partitioning programs and generating communication, but privatization causes several new problems to arise. The main difference between privatization and the other techniques is that privatization is implicit (in the sense that the privatizable variables are merely indicated as such) whereas scalar expansion and forward substitution are explicit. Scalar expansion replaces temporary variables with expanded temporary arrays (and can be viewed as a form of explicit privatization) while forward substitution completely removes the references to the temporary variables.

For distributed-memory environments, knowledge of which iterations each processor will execute a given statement is necessary in order to partition loops and statically generate any required communication. Since a reference to an implicitly privatized scalar or array variable will not contain subscripts corresponding to the loops over which it is privatized, this ownership information is absent and the variable would normally be considered to be replicated across the processors if any of these loops were partitioned.

If a loop is partitioned by directly distributing iterations across the available processors without regards to the data distribution of the references contained within it, communication is determined by comparing the ownership of a reference directly against the ownership of the iterations of the partitioned loop bounds. Even though private variables can normally be handled in this case by just ignoring them during communication analysis for the loops over which they are privatized, this form of computation partitioning can potentially result in both a pre- and post-loop communication phase (to obtain non-local data required for the computation and then to write back non-local results that were computed locally).

The alternative is to partition a loop by distributing the iterations such that the processor which performs a given iteration actually owns the data items computed within that iteration. This approach has become known as the *owner-computes* rule. The advantage of the owner-computes rule is that it allows all required communication to be performed once (to only obtain non-local data required for the computation), before a parallel computation begins. Communication is now determined by comparing the iteration ownership of references read against the references that are written for a given statement. Unfortunately, private variables can no longer be just ignored as the iterations for which they are owned by a processor must be known in order to determine the communication.

In this paper, we present a simple extension to a uniform array-region analysis framework that utilizes privatization information to partition loops and generate efficient communication, using the owner-computes rule, in the presence of temporary variables. This framework has been developed as part of the PARADIGM compiler project [7] at the University of Illinois.

The remainder of this paper is organized as follows: Section 2 describes the different approaches that are available for handling temporary variables which are used to store the values of common subexpressions. The techniques we have developed to support privatization using the owner-computes rule are described in Section 3. Experimental results obtained using an Intel Paragon to compare the different techniques are presented and evaluated in

This research was supported in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613, in part by an Office of Naval Research Graduate Fellowship, and in part by the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research office.

```
DO I = 2, IL
   XX        = X(I,1,1) - X(I-1,1,1)
   YX        = X(I,1,2) - X(I-1,1,2)
   PA        = P(I,2) + P(I,1)
   FS(I,1,1) = 0.
   FS(I,1,2) = - YX * PA
   FS(I,1,3) =   XX * PA
   FS(I,1,4) = 0.
ENDDO
```

Figure 1: Example Loop Utilizing Temporaries

Section 4. Related work in this area is then discussed in Section 5 and our conclusions are presented in Section 6.

2. APPROACHES

To illustrate each of the different approaches, each will be applied to an example loop which uses temporary variables to store intermediate computations. In Figure 1 such a loop, which appears in the EFLUX subroutine in FLO52 from the Perfect Club [8] suite, is shown. In this example, all three scalars (XX, YX, and PA) are temporary variables used only to store partial subexpressions that are immediately used in the computation.

2.1 Direct Application of the Owner-Computes Rule

It is possible to perform a direct application of the owner-computes rule to the code shown in Figure 1 (with the first dimension of the arrays partitioned across a linear array of processors). Due to the fact that the scalar variables create a single convergence point for the values of the computation, several problems will arise.

The first, and most significant, problem is that it is not possible to partition the loop across the processors. Since scalars are normally replicated on all processors, every processor must execute every iteration of the loop in order to satisfy the owner-computes rule. If the ownership of the scalars were assigned to a specific processor, the situation would actually become worse as this would only serve to serialize the computation on that processor resulting in every other processor waiting to receive the scalar values as they are computed (for this reason, single ownership of scalars will not be considered any further).

The second, less significant, problem which arises with replicated scalars is that to compute their values all processors must broadcast their portions of the partitioned arrays in order make the required data available on all processors. If the owner-computes rule were instead relaxed to an owner-stores rule, the communication is reduced somewhat as the actual computation could be carried out where the data existed and only the result of the computation would be broadcast to all processors. Either way, there will be a lot of communication occurring simultaneously among the processors, which can potentially degrade performance due to network contention.

2.2 Global Forward Substitution

In cases where there is only one control path between the last write of a temporary variable and its use, it is also possible to perform global forward substitution [1, 2] to replace all uses with the actual expression used to compute the temporary. Since each use of a temporary is replaced with the temporary's expression, the major drawback of forward substitution is the potential for increasing the operation count and degrading the performance of a program through redundant computation. This will occur whenever a temporary is used in multiple references.

```
   DO I = 2, IL
      FS(I,1,1) = 0.
      FS(I,1,2) = - (X(I,1,2) - X(I-1,1,2))
&                   * (P(I,2) + P(I,1))
      FS(I,1,3) =   (X(I,1,1) - X(I-1,1,1))
&                   * (P(I,2) + P(I,1))
      FS(I,1,4) = 0.
   ENDDO
```

Figure 2: After Performing Global Forward Substitution

The code in Figure 2 is the result of performing global forward substitution on the example in Figure 1. As the temporary PA was used twice in the original code, after the transformation the operation count of the loop body has been increased by one which, overall, has added $IL - 2$ operations to the loop. In general the reuse of temporaries that were computed from complex expressions can have a large impact on the overall operation count of a program.

The resulting code in Figure 2 can now be easily analyzed to statically partition the loop and generate communication according to the owner-computes rule. Once the code has been partitioned, it is possible that the adverse effects of forward substitution could be reduced or eliminated if the target machine's node compiler was able to optimize the computation through common subexpression elimination (which is the inverse of global forward substitution). It is debatable, however, to what extent most commercial optimizing compilers would actually perform common subexpression elimination for floating point expressions (this question will be examined in more detail in Section 4).

```
DO I = 2, IL
   XX(I)     = X(I,1,1) - X(I-1,1,1)
   YX(I)     = X(I,1,2) - X(I-1,1,2)
   PA(I)     = P(I,2) + P(I,1)
   FS(I,1,1) = 0.
   FS(I,1,2) = - YX(I) * PA(I)
   FS(I,1,3) =   XX(I) * PA(I)
   FS(I,1,4) = 0.
ENDDO
```

Figure 3: After Performing Scalar Expansion

2.3 Scalar Expansion

Scalar expansion is a technique that can be used to promote scalar temporaries into arrays of temporary values [1, 2, 3]. Since a scalar is replaced with an array (with dimensions related to the extent of the loops bounds in which the scalar was contained), scalar expansion can greatly increase the memory requirements of a program. The benefit, however, is fairly obvious when examining the resulting code in Figure 3. The convergence points due to the scalar temporaries have been effectively eliminated as arrays of temporaries are now used instead. The loops can now be partitioned across the processors and communication requirements have dropped significantly as the broadcast operations are no longer necessary as only a single element shift of data is required to obtain non-local values.

```
!HPF$ INDEPENDENT, NEW(XX, YX, PA)
      DO I = 2, IL
         XX        = X(I,1,1) - X(I-1,1,1)
         YX        = X(I,1,2) - X(I-1,1,2)
         PA        = P(I,2) + P(I,1)
         FS(I,1,1) = 0.
         FS(I,1,2) = - YX * PA
         FS(I,1,3) =   XX * PA
         FS(I,1,4) = 0.
      ENDDO
```

Figure 4: Specification of Privatization

2.4 Privatization

Other research in the area has resulted in several techniques which can be used to determine when variables appearing within are loop are simply used as scratch space for the current iteration and can therefore be treated as a private reference for each iteration of the loop. A privatizable reference can be either a scalar variable [4] or an array [5, 6].

In Figure 4, the example is shown along with the High Performance Fortran (HPF) [9] `NEW` directive which is provided for specifying that a variable can be privatized with respect to the associated loop. This allows privatization to be specified either manually by a programmer or automatically through a separate compiler pass.

Privatization is similar to defining variables which are local to a loop nest, as is possible in Cedar Fortran [10]. Since the privatized variables will only be used within a single iteration before being overwritten, only one copy of each variable need be maintained for each processor as opposed to actually maintaining one for each iteration. For this reason, an efficient implementation of privatization will require the same amount of memory and perform the same number of operations as the original program.

2.5 Summary of the Approaches

The advantages of each of the different approaches for handling variables which are used to compute common subexpressions are summarized in Table 1. For a given technique, each category indicates: memory utilization is maintained, performance is maintained, or compilation for distributed memory is simplified.

Even though privatization is the most attractive of these techniques in terms of memory use and potential performance, it causes several new problems to arise when compiling for distributed-memory machines using the owner-computes rule. These problems along with the technique that we have developed to handle privatized variables within the PARADIGM compilation framework will be discussed in the next section.

Table 1: Advantages of Each of the Available Approaches

	Memory	Performance	Compilation
Direct partitioning	√		
Scalar expansion		√	√
Fwd. substitution	√		√
Privatization	√	√	

3. SUPPORT FOR PRIVATIZATION

In this section, we will first briefly describe PARADIGM's approach to parallelization, which involves the tasks of static loop partitioning and communication generation. We will then discuss how privatization is handled in the compilation framework.

3.1 Basic Compilation Strategy

PARADIGM applies the owner-computes rule to distribute computation across processors through loop partitioning. Given a program's data distribution information, which can be either manually specified with HPF directives or automatically generated by PARADIGM [11], the compiler first computes the ACCESS sets of all references enclosed in loops. The ACCESS set of a reference with respect to a particular processor is the set of loop iterations for which the reference is local to the processor. According to the owner-computes rule, a processor only executes loop iterations for which some reference on the left-hand side (*lhs*) of an assignment statement in the loop is local to that processor. In other words, the loop can be partitioned by having each processor execute only the loop iterations that are in the union of the *lhs* ACCESS sets in the loop. This union forms the *reduced iteration set* (RIS) of the loop [12]. For example, the loop:

DO $i = L, U$
  $A(i) = \cdots$

after loop partitioning becomes:

DO $i \in \{i \mid A(i) \text{ is stored locally}\}$
  A(i) = $\cdots$

In addition, if the RIS causes an access to a non-local right-hand side (*rhs*) reference in the loop, it must be received from the processor that owns it. To determine which iterations require interprocessor communication, PARADIGM computes the set difference between the *lhs* and *rhs* ACCESS sets [13, 14].

It is worth mentioning that PARADIGM also supports the FORALL construct, providing a pass that rewrites FORALL into DOALL loops while maintaining the original semantics [15]. This process occurs before the computation of ACCESS sets.

Since scalar variables are normally replicated on all processors (as discussed in Section 2.1), partitioning a loop containing scalars by directly applying the owner-computes rule will result in code with very poor performance. If, for example in Figure 1, `XX` is replicated, its ACCESS set is the entire iteration space. Since the RIS is the union of the ACCESS sets of the enclosed *lhs* references, the `I` loop must therefore also traverse the entire iteration space regardless of the distribution of the other *lhs* references in the loop. In other words, this loop is not partitioned, and hence no useful speedup can be expected. Worse yet, because the *lhs* is replicated, any *rhs* array that is partitioned will cause an all-to-all broadcast which effectively replicates the *rhs* arrays on all processors as well, tremendously wasting both time and memory. As discussed in Section 2.4, these problems can be overcome by privatizing the temporary variables.

3.2 Privatization Techniques in PARADIGM

In PARADIGM, privatization information is assumed to be either provided by a separate pass or specified through the use of the HPF `INDEPENDENT` directive with a `NEW` clause, as shown in Figure 4. Given a variable in the `NEW` clause, a reference appearing inside the associated loop is considered private. This implies that a private copy "exists" for each iteration of that loop (as well as the iterations of any outer loops). References appearing outside of the loop are not private.

```
PARTITION(loop_nest)
for each ref in loop_nest
    do Compute ACCESS set for ref
for each loop in loop_nest
    do Compute new loop bounds for loop,
        UNION of lhs ACCESS sets

▷ Extension to update ACCESS sets
▷ for privatized references
for each priv_ref in loop_nest
    do if priv_ref = LHS(STMT(priv_ref))
          then add set dimensions collected from
                 outer loops to ACCESS(priv_ref)
          else  add private portion of
                 ACCESS(LHS(STMT(priv_ref)))
                 to ACCESS(priv_ref)

for each stmt in loop_nest
    do Compute communication sets for stmt
```

Figure 5: Loop Partitioning Algorithm
Macros accessing fields in structures are shown in CAPS
Functions are indicated by SMALLCAPS

In order to deal with privatized references, the loop partitioning and communication generation algorithm is performed with the following extension. Let T be a reference to a temporary variable that has been declared as private. During the computation of the RIS all *lhs* references that have been specified to be private for the loop are ignored. After all of the loops have been partitioned according to their RIS, the ACCESS set of all private references, T, appearing on the *lhs* are augmented with the set collected from the enclosing loops for which T is private.[a]

This means that the value of T will be computed for only those iterations of the (partitioned) loop nest executed by a specific processor to carry out the computation of the non-privatized references. For a privatized reference that is on the *rhs*, its ACCESS is augmented with the portion of the *lhs* ACCESS set of the statement in which it is contained, thus encompassing iterations over which it is private.

This technique applies to both privatized scalars and arrays in a straightforward manner. The only difference is that privatized arrays also have their own ACCESS set dimensions (corresponding to their array dimensions) which must be included in addition to the privatized set dimensions obtained as described previously.

Once all of the ACCESS sets have been computed, communication generation is performed without modification [13, 14]. Communication may be generated for a statement that computes a privatized temporary T, but it will be limited to only those iterations that each processor will be executing according to the partitioned RIS for the loop nest. For the uses of privatized temporaries, the values of the private references will be available for every iteration of the loops over which they have been privatized. By assigning private *rhs* ACCESS sets in the manner described previously, their set differences with the corresponding *lhs* will be empty, and therefore no communication will be generated for these references over loops for which they are private.

The overall partitioning technique with support for privatization (summarized in Figure 5), can now be applied to the example in Figure 4. The RIS of the `I` loop will be computed as the union of the ACCESS sets of `FS(I,1,1)`, `FS(I,1,2)`, `FS(I,1,3)`, and `FS(I,1,4)`. The ACCESS sets of `XX`, `YX` and `PA` in the first three assignments are then collected from the enclosing loops for which they are private. For this example, this is just RIS of the `I` loop. Communication would be generated for `X` for the first two statements based on the ACCESS sets of `XX`, `YX`, and `X`. No communication is generated for the last four statements, since all of their *rhs* references are privatized (or constant).

[a] ACCESS sets of *lhs* private references could also be obtained from the union of the ACCESS sets from the *lhs* of the assignments containing their uses. This requires tracing the flow for each private reference and will only show a benefit when the private variables are used in statements in which the union of their *lhs* is much smaller than the union of all the *lhs* within the loop nest.

```
      PROGRAM STENCIL
      INTEGER I, J
      REAL T1, T2
      REAL A(1000,1000), B(1000,1000)
      REAL C(1000,1000), D(1000,1000)
!HPF$ INDEPENDENT
      DO 10 J = 2, 999
!HPF$ INDEPENDENT, NEW(T1, T2)
      DO 10 I = 2, 999
        T1 = (A(I,J) + A(I-1,J-1) + A(I,J-1) +
     &    A(I+1,J-1) + A(I-1,J) + A(I+1,J+1) +
     &    A(I-1,J+1) + A(I,J+1) + A(I+1,J)) / 9.
        T2 = (B(I,J) + B(I-1,J-1) + B(I,J-1) +
     &    B(I+1,J-1) + B(I-1,J) + B(I+1,J+1) +
     &    B(I-1,J+1) + B(I,J+1) + B(I+1,J)) / 9.
        C(I,J) = T1 + T2
        D(I,J) = T1 - T2
   10 CONTINUE
      END
```

Figure 6: Serial Code of STENCIL

4. EVALUATION

Two benchmarks, STENCIL and EFLUX, are used to evaluate the effectiveness of the different approaches. STENCIL, shown in Figure 6, is a synthetic example involving 9-point stencil computations on several 1000×1000 arrays. EFLUX is a subroutine from the Perfect Club Benchmark FLO52 [8], with the first dimension of each array increased to 5000 (was originally 194). Its code, listed in Figure 7, also includes additional HPF directives that help the compiler identify privatizable variables. For clarity, initialization loops were excluded from both of these listings. The arrays in STENCIL were 2-D BLOCK distributed, whereas those in EFLUX were distributed by BLOCK along the first dimension. Both of these data distribution schemes were automatically selected by PARADIGM.

The parallel versions of these benchmarks using privatization are generated directly by the compiler from the sequential code plus HPF directives. Figure 8 shows the output of PARADIGM for EFLUX using privatization support. To test a direct application

```
      PROGRAM EFLUX
      INTEGER IL, JL, I2, J2
      PARAMETER(I2=5000, J2=34, IL=I2-1, JL=J2-1)
C     EULER FLUXES
      DIMENSION DW(I2,J2,4), FS(I2,J2,4)
      DIMENSION W(I2,J2,4), P(I2,J2), X(I2,J2,2)
!HPF$ INDEPENDENT
      DO 10 J = 2, JL
!HPF$ INDEPENDENT, NEW(XY, YY, PA, QSP, QSM)
      DO 10 I = 1, IL
      XY        = X(I,J,1) - X(I,J-1,1)
      YY        = X(I,J,2) - X(I,J-1,2)
      PA        = P(I+1,J) + P(I,J)
      QSP       = (YY*W(I+1,J,2)
     &            - XY*W(I+1,J,3))/W(I+1,J,1)
      QSM       = (YY*W(I,J,2) - XY*W(I,J,3))/W(I,J,1)
      FS(I,J,1) = QSP*W(I+1,J,1) + QSM*W(I,J,1)
      FS(I,J,2) = QSP*W(I+1,J,2) + QSM*W(I,J,2) + YY*PA
      FS(I,J,3) = QSP*W(I+1,J,3) + QSM*W(I,J,3) - XY*PA
      FS(I,J,4) = QSP*(W(I+1,J,4) + P(I+1,J))
     &            + QSM*(W(I,J,4) + P(I,J))
   10 CONTINUE
      DO 20 N = 1, 4
      DO 20 J = 2, JL
      DO 20 I = 2, IL
      DW(I,J,N) = FS(I,J,N) - FS(I-1,J,N)
   20 CONTINUE
!HPF$ INDEPENDENT, NEW(XX, YX, PA)
      DO 25 I = 2, IL
      XX        = X(I,1,1) - X(I-1,1,1)
      YX        = X(I,1,2) - X(I-1,1,2)
      PA        = P(I,2)   + P(I,1)
      FS(I,1,1) = 0.
      FS(I,1,2) = -YX*PA
      FS(I,1,3) =  XX*PA
      FS(I,1,4) = 0.
   25 CONTINUE
!HPF$ INDEPENDENT
      DO 30 J = 2, JL
!HPF$ INDEPENDENT, NEW(XX, YX, PA, QSP, QSM)
      DO 30 I = 2, IL
      XX        = X(I,J,1)  -X(I-1,J,1)
      YX        = X(I,J,2)  -X(I-1,J,2)
      PA        = P(I,J+1)  +P(I,J)
      QSP       = (XX*W(I,J+1,3)
     &            - YX*W(I,J+1,2))/W(I,J+1,1)
      QSM       = (XX*W(I,J,3) - YX*W(I,J,2))/W(I,J,1)
      FS(I,J,1) = QSP*W(I,J+1,1) + QSM*W(I,J,1)
      FS(I,J,2) = QSP*W(I,J+1,2) + QSM*W(I,J,2) - YX*PA
      FS(I,J,3) = QSP*W(I,J+1,3) + QSM*W(I,J,3) + XX*PA
      FS(I,J,4) = QSP*(W(I,J+1,4) + P(I,J+1))
     &            + QSM*(W(I,J,4)  +P(I,J))
   30 CONTINUE
      DO 40 N = 1, 4
      DO 40 J = 2, JL
      DO 40 I = 2, IL
      DW(I,J,N) = DW(I,J,N) + FS(I,J,N) - FS(I,J-1,N)
   40 CONTINUE
      END
```

Figure 7: EFLUX Taken from the Perfect Club Benchmark FLO52 [8] Augmented with HPF Directives

of the owner-computes rule without privatization, the HPF `NEW()` clauses are removed before compiling with PARADIGM. Without these clauses, previously privatized variables are treated as regular scalars that are replicated on all processors. For scalar expansion and forward substitution, the corresponding code transformations were first performed manually to the sequential code, and were then compiled by PARADIGM to obtain the parallel version.

For timing purposes, both benchmarks were iterated 100 times on an Intel Paragon for each approach. Table 2 shows EFLUX's execution times and speedups, for up to 16 processors, for the four approaches. Similarly, Table 3 shows the results for STENCIL. Notice that the serial times for privatization and the direct application of the owner-computes rule are the same, since they do not restructure the original sequential program as do scalar expansion and forward substitution. This sequential time is used as the baseline to compute speedup for all four approaches.

As expected, a direct application of the owner-computes rule does not provide any significant speedup regardless of the number of processors involved, because loops containing private variables are not partitioned. For EFLUX, only two small loop nests (20 and 40) are parallelizable under this approach, while none of the loops in STENCIL are parallelizable. The failure to partition such loops leads to redundant computations as well as increased network traffic to replicate arrays on all processors. In contrast, privatization shows the best results, especially in STENCIL, with near-ideal speedups.

Although scalar expansion comes close in terms of speedup, it has the disadvantage of using significantly more memory than the other three approaches. Table 4 lists the memory usage of uniprocessor runs, and shows that scalar expansion requires 47% more memory in EFLUX and 50% more in STENCIL. This not only limits the problem size allowed, but also can give rise to adverse cache effects due to a larger working set.

Forward substitution performed two to three times slower than privatization because the expressions after substitution resulted in a substantially increased operation count, and the increased complexity of these expressions made it difficult for the sequential compiler to perform any significant common subexpression elimination.

5. RELATED WORK

Both scalar [4] and array [5, 6] privatization techniques have been the focus of much research over the past several years. As most of these techniques have been used in relation to shared-memory multiprocessors to enable parallelization, not much work has been done to also allow distributed-memory compilation to benefit from these efforts .

Privatization was not supported in either the Fortran D [12] or Vienna Fortran [16] but has since been added to the HPF language definition [9]. Whenever situations arose that could have been handled by privatization, techniques such as global forward substitution [17] were commonly applied instead.

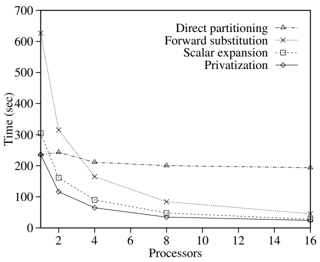
More recently, a prototype HPF compiler being developed as part of the SP2 project [18] has addressed this issue in a different manner. In this project, scalars are assumed to be replicated by default, but for each definition of the scalar that can be privatized, decision to use the privatization are determined in a separate pass by examining different alignments for each use of the private variable. Privatization is used if there is a lower estimated cost resulting from communication being moved outside of the loop. As mentioned earlier in Section 3, this requires tracing the flow for

```
      PROGRAM eflux
      include 'fnx.h'
      CHARACTER m$buf(680)
      INTEGER my$proc, m$numproc, m$status
      INTEGER my$pid, my$p(2), m$num(2), m$numdim
      INTEGER m$to(-1:1,-1:1)
      INTEGER m$bx1, m$bp1, m$bw1, m$bfs1, m$bdw1
      INTEGER f$pack2, f$unpack2, f$pack3, f$unpack3
      INTEGER m$inc, m$off
      INTEGER il, jl, i2, j2
      PARAMETER(i2 = 5000, j2 = 34, il = 4999, jl = 33)
      REAL dw(313,34,4), fs(0:313,34,4), w(314,34,4)
      REAL p(314,34), x(0:313,34,2)
      INTEGER j, i

      my$proc = mynode()
      my$pid = mypid()
      m$numproc = numnodes()
      CALL gsync()
c     *** Minimum mesh configuration
      m$numdim = 2
      m$num(1) = 16
      m$num(2) = 1
c     *** Reconfigure for available processors
      CALL m$gridinit(m$numdim, m$num, m$numproc)

      my$p(1) = mod(my$proc,m$num(1))
      my$p(2) = my$proc / m$num(1)
      m$to( -1,0) = m$num(1) * my$p(2) + my$p(1) - 1
      m$to(1,0) = m$num(1) * my$p(2) + my$p(1) + 1

      m$bdw1 = ceil(float(5000) / m$num(1))
      m$bfs1 = ceil(float(5000) / m$num(1))
      m$bw1 = ceil(float(5000) / m$num(1))
      m$bp1 = ceil(float(5000) / m$num(1))
      m$bx1 = ceil(float(5000) / m$num(1))
!HPF$ INDEPENDENT
      IF (my$p(1) .GE. 1) THEN
        m$off = 1
        m$inc = f$pack3(w,4,1,314,1,34,1,4,1,1,1,1,
     1  34,1,1,4,1,m$buf)
        m$off = m$off + m$inc
        m$inc = f$pack2(p,4,1,314,1,34,1,1,1,1,34,1,
     1  m$buf(m$off))
        m$off = m$off + m$inc
        CALL csend(0,m$buf,m$off - 1,m$to( -1,0),my$pid)
      END IF
      IF (my$p(1) .LE. (m$num(1) - 2)) THEN
        CALL crecv(0,m$buf,680)
        m$off = 1
        m$inc = f$unpack3(w,4,1,314,1,34,1,4,m$bw1
     1  + 1,m$bfs1 + 1,1,1,34,1,1,4,1,m$buf)
        m$off = m$off + m$inc
        m$inc = f$unpack2(p,4,1,314,1,34,m$bp1
     1  + 1,m$bfs1 + 1,1,1,34,1,m$buf(m$off))
        m$off = m$off + m$inc
      END IF
      DO 10 j = 2,33
!HPF$ INDEPENDENT, NEW (xy, yy, pa, qsp, qsm)
        DO 10 i = 1,MIN(-m$bfs1 * my$p(1) + 4999,m$bfs1)
          xy = x(i,j,1) - x(i,j - 1,1)
          yy = x(i,j,2) - x(i,j - 1,2)
          pa = p(i + 1,j) + p(i,j)
          qsp = (yy * w(i + 1,j,2) - xy * w(i + 1,j,3))
     1    / w(i + 1,j,1)
          qsm = (yy * w(i,j,2) - xy * w(i,j,3)) / w(i,j,1)
          fs(i,j,1) = qsp * w(i + 1,j,1) + qsm * w(i,j,1)
          fs(i,j,2) = qsp * w(i + 1,j,2) + qsm * w(i,j,2)
     1    + yy * pa
          fs(i,j,3) = qsp * w(i + 1,j,3) + qsm * w(i,j,3)
     1    - xy * pa
          fs(i,j,4) = qsp * (w(i + 1,j,4) + p(i + 1,j))
     1    + qsm * (w(i,j,4) + p(i,j))
10        CONTINUE
      IF (my$p(1) .LE. (m$num(1) - 2)) THEN
        m$inc = f$pack3(fs,4,0,313,1,34,1,4,m$bdw1,
     1  m$bfs1,1,1,34,1,1,4,1,m$buf)
        CALL csend(1,m$buf,m$inc,m$to(1,0),my$pid)
      END IF
      IF (my$p(1) .GE. 1) THEN
        CALL crecv(1,m$buf,544)
        m$inc = f$unpack3(fs,4,0,313,1,34,1,4,0,0,
     1  1,1,34,1,1,4,1,m$buf)
      END IF
      DO 20 n = 1,4
        DO 20 j = 2,33
          DO 20 i = MAX(-m$bdw1 * my$p(1) + 2,1),
     1      MIN(-m$bdw1 * my$p(1) + 4999,m$bdw1)
            dw(i,j,n) = fs(i,j,n) - fs(i - 1,j,n)
20          CONTINUE
!HPF$ INDEPENDENT, NEW (xx, yx, pa)
      IF (my$p(1) .LE. (m$num(1) - 2)) THEN
        m$inc = f$pack3(x,4,0,313,1,34,1,2,m$bfs1,
     1  m$bx1,1,1,34,1,1,2,1,m$buf)
        CALL csend(2,m$buf,m$inc,m$to(1,0),my$pid)
      END IF
      IF (my$p(1) .GE. 1) THEN
        CALL crecv(2,m$buf,272)
        m$inc = f$unpack3(x,4,0,313,1,34,1,2,0,0,
     1  1,1,34,1,1,2,1,m$buf)
      END IF
      DO 25 i = MAX(-m$bfs1 * my$p(1) + 2,1),
     1 MIN(-m$bfs1 * my$p(1) + 4999,m$bfs1)
        xx = x(i,1,1) - x(i - 1,1,1)
        yx = x(i,1,2) - x(i - 1,1,2)
        pa = p(i,2) + p(i,1)
        fs(i,1,1) = 0.
        fs(i,1,2) =  -yx * pa
        fs(i,1,3) = xx * pa
        fs(i,1,4) = 0.
25      CONTINUE
!HPF$ INDEPENDENT
      IF (my$p(1) .LE. (m$num(1) - 2)) THEN
        m$inc = f$pack3(x,4,0,313,1,34,1,2,m$bfs1,
     1  m$bx1,1,1,34,1,1,2,1,m$buf)
        CALL csend(3,m$buf,m$inc,m$to(1,0),my$pid)
      END IF
      IF (my$p(1) .GE. 1) THEN
        CALL crecv(3,m$buf,272)
        m$inc = f$unpack3(x,4,0,313,1,34,1,2,0,0,
     1  1,1,34,1,1,2,1,m$buf)
      END IF
      DO 30 j = 2,33
!HPF$ INDEPENDENT, NEW (xx, yx, pa, pa, qsp, qsm)
        DO 30 i = MAX(-m$bfs1 * my$p(1) + 2,1),
     1    MIN(-m$bfs1 * my$p(1) + 4999,m$bfs1)
          xx = x(i,j,1) - x(i - 1,j,1)
          yx = x(i,j,2) - x(i - 1,j,2)
          pa = p(i,j + 1) + p(i,j)
          qsp = (xx * w(i,j + 1,3) - yx * w(i,j + 1,2))
     1    / w(i,j + 1,1)
          qsm = (xx * w(i,j,3) - yx * w(i,j,2)) / w(i,j,1)
          fs(i,j,1) = qsp * w(i,j + 1,1) + qsm * w(i,j,1)
          fs(i,j,2) = qsp * w(i,j + 1,2) + qsm * w(i,j,2)
     1    - yx * pa
          fs(i,j,3) = qsp * w(i,j + 1,3) + qsm * w(i,j,3)
     1    + xx * pa
          fs(i,j,4) = qsp * (w(i,j + 1,4) + p(i,j + 1))
     1    + qsm * (w(i,j,4) + p(i,j))
30        CONTINUE
      DO 40 n = 1,4
        DO 40 j = 2,33
          DO 40 i = MAX(-m$bdw1 * my$p(1) + 2,1),
     1      MIN(-m$bdw1 * my$p(1) + 4999,m$bdw1)
            dw(i,j,n) = dw(i,j,n) + fs(i,j,n)
     1      - fs(i,j - 1,n)
40          CONTINUE
      END
```

Figure 8: EFLUX Compiled Using Privatization (with a Minimum 16 × 1 Processor Configuration)

Table 2: Performance of EFLUX (100 Iterations)

Approach	Number of Processors 1	2	4	8	16
	Time (sec.)				
Direct partitioning	236	243	211	200	194
Forward substitution	626	315	165	84	45
Scalar expansion	305	162	90	48	28
Privatization	236	116	65	34	24
	Speedup				
Direct partitioning	1.00	0.97	1.12	1.18	1.22
Forward substitution	0.38	0.75	1.43	2.80	5.19
Scalar expansion	0.77	1.46	2.62	4.93	8.51
Privatization	1.00	2.03	3.65	6.83	9.84



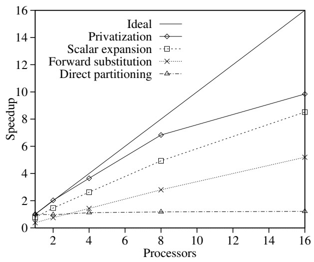
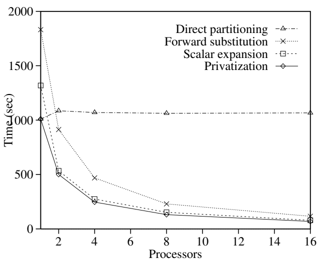


Table 3: Performance of STENCIL (100 Iterations)

Approach	Number of Processors 1	2	4	8	16
	Time (sec.)				
Direct partitioning	1010	1086	1070	1063	1067
Forward substitution	1832	913	469	229	116
Scalar expansion	1318	534	274	152	79
Privatization	1010	501	246	130	69
	Speedup				
Direct partitioning	1.00	0.93	0.94	0.95	0.95
Forward substitution	0.55	1.11	2.15	4.40	8.70
Scalar expansion	0.77	1.89	3.68	6.64	12.83
Privatization	1.00	2.01	4.11	7.73	14.70



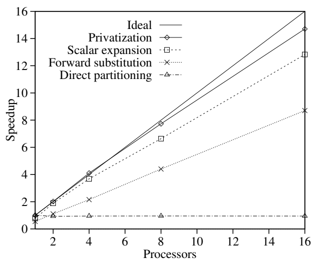


Table 4: Memory Usage

Approach	Application EFLUX	STENCIL
Scalar expansion	14.2 MB	22.9 MB
All others	9.7 MB	15.3 MB

each private reference and will only show a benefit when the private variables are used in statements in which the union of their *lhs* is much smaller than the union of all of the *lhs* within the loop nest. In our compiler, privatization is always used when provided since we assume that it is always beneficial.

Currently, commercial HPF compilers are available from Applied Parallel Research (APR), Digital Equipment Corporation (DEC), and The Portland Group (PGI), among others. APR's xHPF [19] has its own `DO PAR` directive instead of the standard `INDEPENDENT` directive. It was not clear from the documentation whether it supports private variables and, if so, how they are specified. On the other hand, HPF compilers from DEC [20] and PGI [21] do support the `INDEPENDENT` directive, but they do not elaborate on how they handle private variables.

6. CONCLUSIONS

In this paper, we examined several approaches to compile loops that use temporary variables to store intermediate data, presented a uniform approach for handling privatized variables during loop partitioning and communication generation for distributed-memory multicomputers, and experimentally obtained performance results for the different approaches for two benchmarks. For these benchmarks, the results have shown that:

- A direct application of the owner-computes rule results in little or no performance improvements when using privatized variables.
- Forward substitution provides modest speedups with no increase in memory requirements.
- Scalar expansion yields good speedups at the expense of substantial memory overhead.
- Privatization exhibits the best speedups, with no increase in memory usage, and does not require code restructuring.

As these privatization techniques are based only on how the ACCESS sets are assigned to the private references, they will also apply to CYCLIC and BLOCK-CYCLIC distributions [13].

REFERENCES

[1] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, vol. 29, pp. 1184–1201, Dec. 1986.

[2] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*. PhD thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Oct. 1982. CSRD-329.

[3] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, pp. 266–320. New York: ACM Press, 1990.

[4] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh, "Automatic Generation of Nested, Fork-Join Parallelism," *The Journal of Supercomputing*, vol. 3, pp. 71–88, July 1989.

[5] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, "Array Data Flow Analysis and its Use in Array Privatization," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles of Programming Languages*, (Charleston, SC), pp. 2–15, Jan. 1993.

[6] P. Tu and D. Padua, "Automatic Array Privatization," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, vol. 768 of *Lecture Notes in Computer Science*, (Portland, OR), pp. 500–521, Springer-Verlag, Aug. 1993.

[7] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Multicomputers," *IEEE Computer*, vol. 28, pp. 37–47, Oct. 1995.

[8] Perfect Club, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *The International Journal of Supercomputing Applications*, vol. 3, pp. 5–40, Fall 1989.

[9] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, *The High Performance Fortran Handbook*. Cambridge, MA: The MIT Press, 1994.

[10] J. Hoeflinger, "Cedar Fortran Programmer's Handbook," Tech. Rep. CSRD-1157, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Oct. 1991.

[11] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 179–193, Mar. 1992.

[12] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD Distributed Memory Machines," *Communications of the ACM*, vol. 35, pp. 66–80, Aug. 1992.

[13] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee, "Advanced Compilation Techniques in the PARADIGM compiler for Distributed Memory Multicomputers," in *Proceedings of the 9th ACM International Conference on Supercomputing*, (Barcelona, Spain), pp. 424–433, July 1995.

[14] E. Su, D. J. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers," in *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, (Montréal, Canada), pp. 123–132, Aug. 1994.

[15] E. W. Hodges IV, "High Performance Fortran Support for the PARADIGM Compiler," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, Oct. 1995.

[16] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, vol. 1, pp. 31–50, Aug. 1992.

[17] T. Fahringer, *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, University of Vienna, Vienna, Austria, Sept. 1993. TR93-3.

[18] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo, "An HPF Compiler for the IBM SP2," in *Proceedings of Supercomputing '95*, (San Diego, CA), Dec. 1995.

[19] Applied Parallel Research, Inc., Placerville, CA, *XHPF User's Guide, version 2.0*, 1995.

[20] D. B. Loveman, "The DEC High Performance Fortran 90 Compiler Front End," in *Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, (McLean, VA), pp. 46–53, 1995.

[21] The Portland Group, Inc., Wilsonville, OR, *PGHPF User's Guide*, 1995.