Compiler Assisted Generation of Error Detecting Parallel Programs[1]

A. Roy-Chowdhury
459 CSRL 1308 W. Main
Urbana IL 61801
phone: 217 244 8473 FAX: 217 333 1910
email: amber@crhc.uiuc.edu

P. Banerjee
469 CSRL 1308 W. Main
Urbana IL 61801
phone: 217 333 6564 FAX: 217 333 1910
email: banerjee@crhc.uiuc.edu

[1]Acknowledgement: This research was supported by the Joint Services Electronics Programs under contract N00014-90-J-1270

Abstract

We have developed an automated, compile time approach to generating error-detecting parallel programs. The compiler is used to identify statements implementing affine transformations within the program and automatically insert code for computing, manipulating, and comparing checksums in order to check the correctness of the code implementing affine transformations. Statements which do not implement affine transformations are checked by duplication. Checksums are reused from one loop to the next if this is possible, rather than recomputing checksums for every statement. A global dataflow analysis is performed in order to determine points at which checksums need to be recomputed. We also use a novel method of specifying the data distributions of the check data using directives provided by the High Performance Fortran (HPF) standard so that the computations on the original data and the corresponding check computations are performed on different processors. Results are presented on an Intel Paragon distributed memory multicomputer.

Keywords: Algorithm-Based Fault Tolerance, Checksum Encoding, Parallelizing Compilers, Compiler Assisted Fault Tolerance.

1 Introduction

Numerical programs for parallel computers operate on enormous data sets and often take hours of computer time to finish executing. Due to the large amounts of hardware involved and the long execution times of these programs, it may become necessary to detect and possibly tolerate errors which occur during the execution of the program. Algorithm-based fault tolerance is a methodology of exploiting numerical properties of algorithms to devise fault-tolerant versions of numerical programs. The basic approach is to apply some encoding to the data being operated on by the algorithm, modify the encoded data concurrently with the original data, and check that the encoding is preserved at various points during the execution of the algorithm. Fault-tolerant algorithms using this approach have been devised for several numerical algorithms ([1], [2], [3], [4], [5], [6], [7]).

For a class of algorithms performing linear transformations on the data, a natural encoding to choose is the checksum encoding [1], where a checksum is computed of the data being operated on by the algorithm. The checksum is then transformed concurrently with the computations on the data elements, and at suitable points during the execution, the data elements are summed and compared with the transformed checksum.

As an example, consider the problem of matrix multiplication $C = AB$. The simple algorithm may be made error-detecting by the addition of an extra row to A which is computed by taking the sum of all other rows of A. The product of the extra row of A with B yields an extra row in the product matrix C which should equal the sum of all the other rows of C in the absence of errors (due to roundoff errors, a small tolerance has to be allowed in the comparison). This is illustrated in Fig. 1. This idea can be extended in an obvious manner in a multiprocessor environment, with each processor checking the data on another processor. This is illustrated in Fig. 2. Note that for $n \times n$ matrices, only $O(n^2)$ operations are required for creating, manipulating, and comparing checksums, while $O(n^3)$ operations are used to compute the matrix multiplication.

We have developed an automated, compile time approach to generating error-detecting parallel programs based on the above idea. The compiler is used to identify statements implementing affine transformations

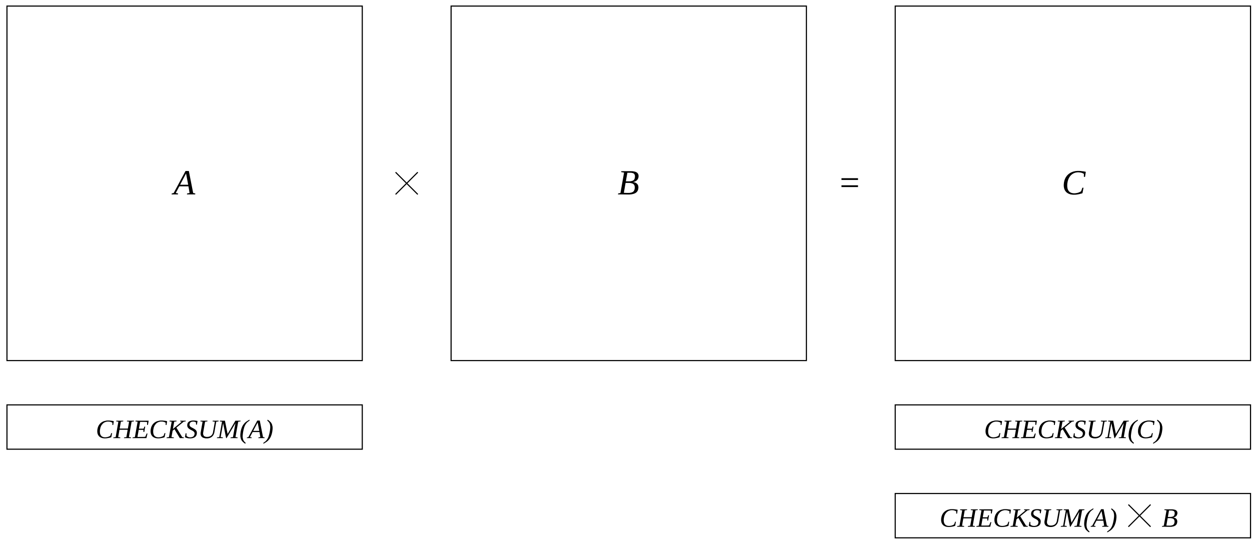

Figure 1: Illustration of matrix multiplication with checksums for error detection

within the program and automatically insert code for computing, manipulating, and comparing checksums in order to check the correctness of the code implementing affine transformations. Statements which do not implement affine transformations are checked by duplication. Checksums are reused from one loop to the next if this is possible, rather than recomputing checksums for every statement. A global dataflow analysis is performed in order to determine points at which checksums need to be recomputed. We also use a novel method of specifying the data distributions of the check data using directives provided by the High Performance Fortran (HPF) [14] standard so that the computations on the original data and the corresponding check computations are performed on different processors.

The organization of this report is as follows. We discuss some related work in Section 2. We give some motivation for our work by introducing an example code fragment in Section 3 and describing how it would be transformed to generate an error-detecting version of the code fragment. We give an overview of the entire system implementation in Section 4. We then describe the algorithms used in performing the transformations needed to generate the error-detecting version in Section 5. We present some results on the overheads of the error-detecting version over the original code on a real parallel machine in Section 6. Finally, we summarize and conclude in Section 7.

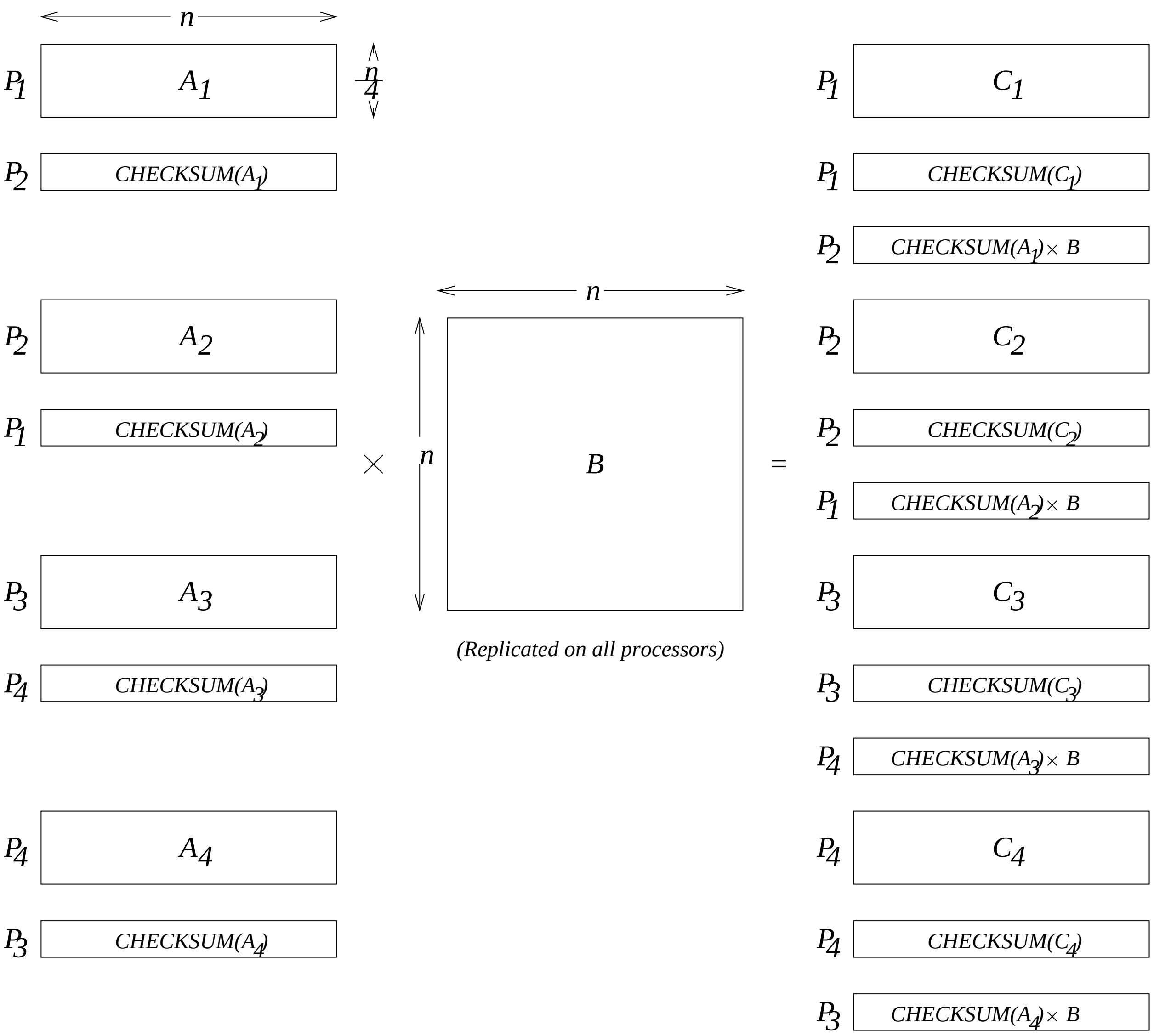

Figure 2: Parallel implementation of matrix multiplication with checksums for error detection

2 Related Work

Other researchers have also looked at the problem of automatic generation of error-detecting code at compile time. The approach of [8] and [9] is to utilize the VLIW compiler to insert redundant operations into functional units that would otherwise be idle. Fault diagnosis may also be done by analyzing functional unit mismatches. This approach requires hardware modification in the form of comparators to compare the outputs of the functional units. Also, it is tied to a particular kind of processor architecture, viz., VLIW processors. This technique could be used in conjunction with ours, since duplicated code is produced by our compiler for portions of code which do not implement affine transformations. The duplicated instructions could then be scheduled to utilize idle slots in the functional units of the VLIW processor.

Another approach to compiler assisted fault-detection for parallel programs is discussed in [10], [11]. Here, statements are duplicated in Single Program Multiple Data (SPMD) parallel programs and executed on processors which would otherwise be idle. However, in cases where the overhead for duplication would be too great, for example in loops which can be executed in parallel keeping each processor busy, only the last statement executed by each processor is duplicated and compared on another processor. While this may suffice in detecting permanent faults, it is not adequate for transient fault detection. Again, this technique could be used in conjunction with ours for portions of code which would be checked by duplication using our approach.

An automated approach for identifying linear transformations in a program and generating the code for computing and transforming checksums at compile time was first proposed in [12], [13]. Our approach builds on this idea, while improving on it in several ways to make it viable. The approach analyzed one statement at a time instead of the entire program, leading to potential inefficiencies in checksum computation. Also a full-fledged implementation based on their ideas was not performed, leaving the feasibility and usefulness of their approach unresolved. In the work reported in this report, we improve on the ideas suggested in [12] in several ways and implement them by augmenting a state-of-the-art parallelizing compiler.

The major improvements of our work over [12] are as follows. We are able to generate checksum-based checks for code which performs affine transformations, which are more general than linear transformations. Apart from the statement possessing a suitable syntactic structure, we identify additional conditions which the dependences the statement is involved in must satisfy in order to be able to generate a checksum-based check for it. We attempt to reuse checksums from one loop to the next instead of recomputing checksums for every statement. In order to determine if this is possible, we perform a dataflow analysis on the entire program. Finally, we use data distribution information provided by the original program through HPF directives to specify data distributions for the checksums (or any other extra data which may be needed to check the original computation) in such a manner that a checksum and the portion of data being checked reside on different processors. This, together with the owner-computes rule [15], ensures that the check for the data owned by one processor is performed on a different processor, thus increasing the likelihood of detecting single processor failures.

```
      PROGRAM jacobi
      INTEGER p(4,4)
      REAL a(1000,1000)
      REAL b(1000,1000)
      INTEGER k, j, i

!HPF$ PROCESSORS :: p(4,4)
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO p :: a, b

      DO k = 1,100
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1      ) / 4
          END DO
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      END
```

Figure 3: Code fragment implementing Jacobi's iterative technique

3 A Motivational Example

We will use the code fragment in Fig. 3 to illustrate the development of an error-detecting parallel program. The code fragment solves a system of equations using Jacobi's iterative technique. This and other similar code fragments occur in numerical routines designed to solve partial differential equations used in modeling physical phenomena. Our eventual output is designed to be an error-detecting parallel program computing the same results as the serial program. The serial program is augmented with HPF data distribution directives to aid in the generation of the parallel program; this information is also used in generating the error-detecting version. Note that in the absence of the data distribution annotations, our compiler would be able to generate a serial version of the program useful for detecting transient errors.

Before the actual generation of the checksum based checks is performed, a version of the original program is created by duplicating all array assignments in the program. For the Jacobi example, this results in the

```
      DO k = 1,100
        DO j = 2,999
          DO i = 2,999
            $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j + 1)
     1      ) / 4
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1      ) / 4
          END DO
        END DO
        DO j = 2,999
          DO i = 2,999
            $b(i,j) = $a(i,j)
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      END
```

Figure 4: Jacobi kernel with duplicate array assignments

code fragment shown in Fig. 4.

The next step in the process is to determine duplicate assignment statements in the code which implement affine transformations. These can be identified by examining the syntactic structure of the statement and by verifying that the dependences the statement is involved in satisfy some additional conditions. Both the assignment statements which were newly introduced into the program satisfy the criteria for being affine transformations. These statements are then replaced by statements which transform checksums on array elements rather than the array elements themselves. One of the array dimensions is chosen to compute the checksums over, and the loop traversing this dimension in the original code is deleted. The code after introducing checksum transformations is shown in Fig. 5.

Information about available checksums is then propagated across statements in the program. This information is used to recompute checksums at points where checksum values are required but are not available. For the code shown in Fig. 5, the second checksum statement requires the values of `$cs2_a(i)` for $2 \leq i \leq 999$, but information propagation across statements is able to determine that these values are already available when the second assignment statement is encountered. However, checksums `$cs2_a` and `$cs2_b` need to be computed prior to the start of the `k` loop, since these are required within the loop body.

```
      DO k = 1,100
        DO i = 2,999
          $cs2_a(i) = ($cs2_b(i - 1) + $cs2_b(i + 1) + ($cs2_b(i) - b(i
     1     ,999) + b(i,1)) + ($cs2_b(i) + b(i,1000) - b(i,2))) / 4
        END DO
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1       ) / 4
          END DO
        END DO
        DO i = 2,999
          $cs2_b(i) = $cs2_a(i)
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      END
```

Figure 5: Jacobi kernel after introduction of checksum statements

Checks are generated at intermediate points in the program only where necessary (this will be elaborated on in Section 5) and also at the end of the program. The program after information propagation and checksum and check generation have been performed is shown in Fig. 6.

Next, data distribution information about the original arrays is used to choose a suitable data distribution for the extra data which was introduced in the form of checksums and possibly some extra arrays. This may necessitate introducing an extra dimension for a checksum variable so that each new checksum now covers a smaller portion of the array than the old checksum. This may also require the checksum transformation statements to be modified accordingly. Data distribution information is then specified for checksums and any extra arrays which may have been introduced so that computations on the original data and the corresponding check data are performed on different processors. The code after data distribution information has been introduced for checksums is shown in Fig. 7.

Finally, a parallelizing compiler for a distributed memory machine (in our case, Paradigm [18]) is used to generate an error-detecting parallel program based on the code incorporating checks and data distribution

```
  DO $i1 = 2,999
    DO $i2 = 2,999
      $a($i1,$i2) = a($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    DO $i2 = 1,1000
      $b($i1,$i2) = b($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    $cs2_a($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $cs2_a($i1) = $cs2_a($i1) + a($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    $cs2_b($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $cs2_b($i1) = $cs2_b($i1) + b($i1,$i2)
    END DO
  END DO
  DO k = 1,100
    DO i = 2,999
      $cs2_a(i) = ($cs2_b(i - 1) + $cs2_b(i + 1)
1+ ($cs2_b(i) - b(i,999) + b(i,1)) + ($cs2_b(i)
2+ b(i,1000) - b(i,2))) / 4
    END DO
    DO j = 2,999
      DO i = 2,999
        a(i,j) = (b(i - 1,j) + b(i + 1,j) +
1b(i,j - 1) + b(i,j + 1)) / 4
      END DO
    END DO
    DO i = 2,999
      $cs2_b(i) = $cs2_a(i)
    END DO
    DO j = 2,999
      DO i = 2,999
        b(i,j) = a(i,j)
      END DO
    END DO
  END DO
  DO $i1 = 2,999
    $T($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $T($i1) = $T($i1) + a($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    IF (compare($T($i1),$cs2_a($i1)) .EQ. 1)
1CALL error_handler()
  END DO
  DO $i1 = 2,999
    $T_0($i1) = 0
  END DO
  DO $i1 = 2,999
    DO $i2 = 2,999
      $T_0($i1) = $T_0($i1) + b($i1,$i2)
    END DO
  END DO
  DO $i1 = 2,999
    IF (compare($T_0($i1),$cs2_b($i1)) .EQ. 1)
1CALL error_handler()
  END DO
  END
```

Figure 6: Jacobi code with checks

```
      PROGRAM jacobi
      DOUBLE PRECISION $T_0(1000,4)
      DOUBLE PRECISION $T(1000,4)
      INTEGER $p
      INTEGER $i2
      INTEGER $i1
      DOUBLE PRECISION $cs2_a(1000,4)
      DOUBLE PRECISION $cs2_b(1000,4)
      DOUBLE PRECISION $a(1000,1000)
      DOUBLE PRECISION $b(1000,1000)
      REAL a(1000,1000)
      REAL b(1000,1000)
      INTEGER k, j, i

!HPF$ PROCESSORS :: p(4,4)
!HPF$ DISTRIBUTE (BLOCK, BLOCK) ONTO p :: a, b
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, BLOCK) ONTO p :: template$0(1000, 1000)
!HPF$ ALIGN (hpf$0,hpf$1) WITH template$0(hpf$0 + 250,hpf$1) WRAP :: $a
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, BLOCK) ONTO p :: template$1(1000, 1000)
!HPF$ ALIGN (hpf$0,hpf$1) WITH template$1(hpf$0 + 250,hpf$1) WRAP :: $b
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) ONTO p :: TEMPLATE$2(1000,4)
!HPF$ ALIGN $cs2_a(hpf$0,hpf$1) WITH TEMPLATE$2(hpf$0+250,hpf$1+1) WRAP
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK) ONTO p :: TEMPLATE$3(1000,4)
!HPF$ ALIGN $cs2_b(hpf$0,hpf$1) WITH TEMPLATE$3(hpf$0+250,hpf$1+1) WRAP

      DO $i1 = 2,999
        DO $i2 = 2,999
          $a($i1,$i2) = a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,1000
          $b($i1,$i2) = b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          $cs2_a($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $cs2_a($i1,1) = $cs2_a($i1,1) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $cs2_a($i1,$p) = $cs2_a($i1,$p) + a($i1,($p - 1) * 250 + $i
     1      2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $cs2_a($i1,4) = $cs2_a($i1,4) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          $cs2_b($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $cs2_b($i1,1) = $cs2_b($i1,1) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $cs2_b($i1,$p) = $cs2_b($i1,$p) + b($i1,($p - 1) * 250 + $i
     1      2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $cs2_b($i1,4) = $cs2_b($i1,4) + b($i1,$i2)
        END DO
      END DO
      DO k = 1,100
        DO i = 2,999
          $cs2_a(i,1) = ($cs2_b(i - 1,1) + $cs2_b(i + 1,1) + $cs2_b(i,1
     1    ) + (b(i,2 - 1) - b(i,250)) + $cs2_b(i,1) + (b(i,250 + 1) - b
     2    (i,2))) / 4
          DO $p = 2,3
            $cs2_a(i,$p) = ($cs2_b(i - 1,$p) + $cs2_b(i + 1,$p) + $cs2_
     1      b(i,$p) + (b(i,250 * ($p - 1)) - b(i,250 * ($p - 1) + 250))
     2       + $cs2_b(i,$p) + (b(i,250 * ($p - 1) + 251) - b(i,250 * ($
     3      p - 1) + 1))) / 4
          END DO
          $cs2_a(i,4) = ($cs2_b(i - 1,4) + $cs2_b(i + 1,4) + $cs2_b(i,4
     1    ) + (b(i,751 - 1) - b(i,999)) + $cs2_b(i,4) + (b(i,999 + 1) -
     2     b(i,751))) / 4
        END DO
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1      ) / 4
          END DO
        END DO
        DO i = 2,999
          $cs2_b(i,1) = $cs2_a(i,1)
          DO $p = 2,3
            $cs2_b(i,$p) = $cs2_a(i,$p)
          END DO
          $cs2_b(i,4) = $cs2_a(i,4)
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $T($i1,1) = $T($i1,1) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $T($i1,$p) = $T($i1,$p) + a($i1,($p - 1) * 250 + $i2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $T($i1,4) = $T($i1,4) + a($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          IF (compare($T($i1,$p),$cs2_a($i1,$p)) .EQ. 1) CALL error_han
     1    dler()
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          $T_0($i1,$p) = 0
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 2,250
          $T_0($i1,1) = $T_0($i1,1) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 1,250
          DO $p = 2,3
            $T_0($i1,$p) = $T_0($i1,$p) + b($i1,($p - 1) * 250 + $i2)
          END DO
        END DO
      END DO
      DO $i1 = 2,999
        DO $i2 = 751,999
          $T_0($i1,4) = $T_0($i1,4) + b($i1,$i2)
        END DO
      END DO
      DO $i1 = 2,999
        DO $p = 1,4
          IF (compare($T_0($i1,$p),$cs2_b($i1,$p)) .EQ. 1) CALL error_h
     1    andler()
        END DO
      END DO
      END
```

Figure 7: Jacobi code incorporating checks and data distribution specifications

information.

We would like to point out at this point that the parallel version of the code in Fig. 3 would require $O(kn^2)$ computations in all, where n is the matrix size and k is the number of iterations executed, while the parallel version of the code in Fig. 7 would perform $O(n^2 + kn)$ extra operations due to the checksum computation, updates, and comparison. Thus, the overhead due to the check operations can be expected to be small for large problem sizes. By contrast, a straighforward duplication and check approach, such as one based on the code of Fig. 4 would require more than double the number of operations as the original program. The approach discussed in [12] would also be able to check the bodies of the two loops of the Jacobi code by checksum manipulations, since these happen to be performing linear transformations. However, since information about available checksums is not computed across statements, checksums would be regenerated prior to each loop nest enclosing a checksum manipulation statement for all checksums reqiured by the statement. Following the execution of the checksum code and the original loop being checked by it, a checksum based check would be generated for the array elements being assigned by the loop being checked. This is illustrated in Fig. 8. Note that recomputing the checksums and generating the checks for each loop incurs $O(n^2)$ overhead for each iteration of the `k` loop. By contrast, our approach, which performs dataflow analysis to determine that `$cs2_b` is available upon each entry into the `k` loop and that `$cs2_a` is available prior to the second manipulation statement, does not need to regenerate checksums and perform checks for each loop. Thus, only $O(n)$ overhead is incurred for checksum manipulation for each iteration of the `k` loop.

```
      DO k = 1,100
C Recompute checksums of b
        DO i = 2,999
          $cs2_b(i) = 0
          DO j = 2,999
            $cs2_b(i) = $cs2_b(i) + b(i,j)
          ENDDO
        ENDDO
        DO i = 2,999
          $cs2_a(i) = ($cs2_b(i - 1) +
     1$cs2_b(i + 1) + ($cs2_b(i) - b(i,999)
     2+ b(i,1)) + ($cs2_b(i) + b(i,1000) -
     3b(i,2))) / 4
        END DO
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) +
     1b(i,j - 1) + b(i,j + 1)) / 4
          END DO
        END DO
C Check array a using $cs2_a
        DO i = 2,999
          T = 0
          DO j = 2,999
            T = T + a(i, j)
          ENDDO
          IF (compare(T,$cs2_a(i)) .EQ. 1) CALL
     1error_handler()
        ENDDO
C Recompute checksums of a
        DO i = 2,999
          $cs2_a(i) = 0
          DO j = 2,999
            $cs2_a(i) = $cs2_a(i) + a(i,j)
          ENDDO
        ENDDO
        DO i = 2,999
          $cs2_b(i) = $cs2_a(i)
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
C Check array b using $cs2_b
        DO i = 2,999
          T = 0
          DO j = 2,999
            T = T + b(i, j)
          ENDDO
          IF (compare(T,$cs2_b(i)) .EQ. 1) CALL
     1error_handler()
        ENDDO
      END DO
      END
```

Figure 8: Transformed Jacobi kernel with regeneration of checksums before each loop

4 System Overview

In this section we give an overview of the modules comprising our system. The input set accepted by our compiler consists of Fortran programs with HPF data distribution annotations [14]. Parafrase-2 [16], a parallelizing compiler for shared memory machines developed at the University of Illinois, is used as a front-end module to parse in the input program, build an abstract syntax tree representation of the program, perform dependence analysis, and build the flowgraph. The original compiler was not able to utilize information provided by HPF data distribution information and has been modified to do so [17]. Apart from being a state-of-the-art optimizing and parallelizing compiler, Parafrase-2 has also been designed as a developmental tool. The compiler may be easily augmented by the addition of passes to use and modify the information stored in its internal data structures. Several passes have been added to achieve our goal of generating error-detecting versions of programs. The first pass is responsible for generating duplicate statements corresponding to the statements in the program which operate on arrays. These statements perform the same transformations as the original statements, but on different arrays which we refer to as shadow arrays. The second pass then attempts to replace duplicate statements by statements which transform checksum variables computed by summing over one of the array dimensions wherever possible. The third pass performs information propagation and check code insertion. Information about available checksums and shadow arrays is propagated across statements in the program. If it is determined that a checksum is needed but not available at a point in the program, it is regenerated. Along with regenerating the checksum, checks are generated comparing the elements being summed with the shadow elements, if the latter are available at that point. Similarly, at points where shadow elements are required but are not available, they are copied over from the corresponding array values. If checksums covering these array values are available, a check is also generated to check the elements being copied over. The fourth pass is responsible for taking data distribution information into consideration and specifying suitable data distributions for the check data which was introduced. This may also involve expansion of certain dimensions of the checksum variables which were introduced and consequent modification of some of the statements transforming the checksums.

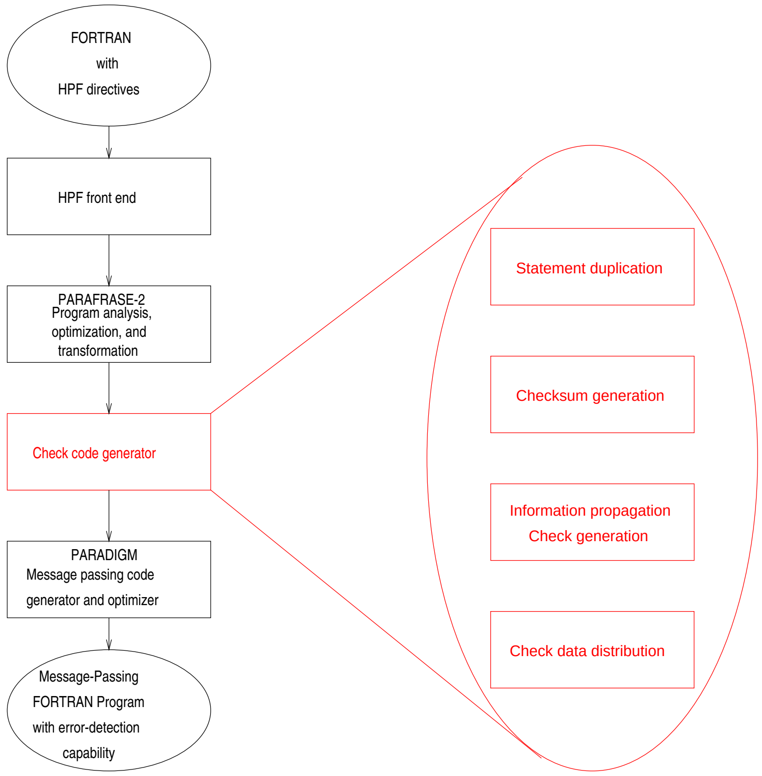

Figure 9: Overall organization of system for generating error-detecting parallel code

Data distribution information is used to specify distributions for checksums and shadow arrays in such a manner that an original array element and the corresponding shadow array element or checksum variable which checks it reside on different processors. The modified program is input to Paradigm [18], a distributed memory parallelizer developed at the University of Illinois, which can generate message passing code for a variety of target multicomputers. The final output is an error detecting parallel program for distributed memory multicomputers. The various modules in our system and their interactions are illustrated in Fig. 9.

5 Algorithms for Check Code Generation

5.1 Statement Duplication

The pass for duplicating statements which operate on arrays is fairly straightforward. However, not only does the pass create duplicate assignment statements for those which assign to or use arrays directly, but it also duplicates statements which use array elements indirectly. For example, a statement which used a scalar to which was assigned an array value prior to reaching the statement would also be duplicated. Also, if loop expressions or if conditionals depend directly or indirectly on array values, the entire loop or if statement, including its body, is duplicated. By duplication of a statement, we mean that a second statement is created performing the same transformations as the original statement, but with array elements and array-dependent variables replaced by different elements which we refer to as shadows. In order to perform statement duplication according to the rules described, it is necessary to determine which scalar variables use array values in their definition. This can be solved as a standard reaching definitions problem [19].

5.2 Checksum Introduction

Once statement duplication has been performed, the pass for determining affine transformations and replacing array elements by checksums is run. However, prior to this a loop distribution pass is run to separate out duplicate statements operating on shadow elements and the corresponding original elements into separate loops. The loop distribution pass also has the effect of separating out different duplicate statements into different loops, which increases opportunities for checksum introduction, as we explain later in this section. Loop distribution on the code in Fig. 4 yields the code shown in Fig. 10.

Given an array $A(l:u)$ which is transformed by a function f satisfying the following property

$$\sum_{i=l}^{u} f(A(i)) = f(\sum_{i=l}^{u} A(i)) \tag{1}$$

we say that the array $A(l:u)$ undergoes a linear transformation. Suppose we have another function g where $g(x) = f(x) + c$, where c is a constant, then clearly we have

$$\sum_{i=l}^{u} g(A(i)) = g(\sum_{i=l}^{u} A(i)) + (l-u)c \tag{2}$$

In this case, we say that the array $A(l:u)$ undergoes an affine transformation. Our approach to generating checksum-based checks is based on identifying duplicated assignment statements which perform a linear or affine transformation on some shadow array. The statement is then replaced by one which transforms checksum values rather than the elements of the shadow array. The expression for transforming the checksum values is derived from Eq. (1) or (2), as the case may be. The loop traversing the array dimension which was summed becomes redundant and may be removed. This often dramatically reduces the overheads contributed by the checksum statement over the statement that it checks.

We first determine perfect loop nests whose bodies consist solely of duplicate statements which are also assignment statements (Change of flow of control within the loop body due to the presence of an `IF` statement, for example, is not allowed. This is a conservative criterion chiefly designed to make the task of the propagation pass easier). For the code in Fig. 10, both the loop nests enclosing duplicate statements with `j` as the outer loop variable are such loop nests. Next, it is determined if the set of variables used by the subscript expressions in the block is a subset of the loop index variables of the perfect nest enclosing the block (This restriction is also made in order to make the task of the propagation pass easier). If these conditions are satisfied, the loop indices of the perfect nest are called the *potential checksum indices* for the statements in the block. For example, the potential checksum indices for both the duplicate assignment statements in Fig. 10 are `i` and `j`. Note that loop distribution increases the number of statements enclosed in perfect nests as well as the number of loops in each perfect nest, which results in an increase in the number of potential checksum indices for each statement. This benefits later stages of the pass.

Once the potential checksum indices have been determined for each duplicate assignment statement, the syntactic structure of each such statement is examined to determine a subset of the potential checksum

```
      DO k = 1,100
        DO j = 2,999
          DO i = 2,999
            $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j
     1       + 1)) / 4
          END DO
        END DO
        DO j = 2,999
          DO i = 2,999
            $b(i,j) = $a(i,j)
          END DO
        END DO
      END DO
      DO k = 1,100
        DO j = 2,999
          DO i = 2,999
            a(i,j) = (b(i - 1,j) + b(i + 1,j) + b(i,j - 1) + b(i,j + 1)
     1      ) / 4
          END DO
        END DO
        DO j = 2,999
          DO i = 2,999
            b(i,j) = a(i,j)
          END DO
        END DO
      END DO
      END
```

Figure 10: Jacobi kernel with duplicated statements after loop distribution

indices such that the statement could possibly be replaced by a checksum manipulation by computing checksums over the array dimensions involving these indices. The indices chosen are called the *candidate checksum indices* in order to distinguish them from the potential checksum indices determined earlier. The candidate checksum indices are computed by traversing the syntax tree associated with the statement under consideration in a bottom-up fashion and updating two sets, called the $AFFINE$ and $NOTAFFINE$ sets, for each node, depending on the value of these sets at its children. The $AFFINE$ set at the root of the tree contains the candidate checksum indices for the statement. The rules for updating the $AFFINE$ and $NOTAFFINE$ sets upon traversing the syntax tree in bottom up fashion are given in Fig. 11. Note that the condition for suitability of subscript expressions is primarily to aid the propagation pass, and could be made less restrictive if the propagation pass were made more sophisticated.

We illustrate the application of the rules in Fig. 11 to the assignment statement shown in Fig. 12. This is derived from one of the statements in Fig. 4, with the addition of a constant to make it an affine transformation rather than a linear transformation. The syntax tree and the computation of the $AFFINE$ and $NOTAFFINE$ sets for this statement are shown in Fig. 14.

Once the set of candidate checksum indices has been computed for each check statement, some additional conditions pertaining to the dependences the statements are involved in need to be verified before a candidate checksum index can actually be chosen as the index to compute the checksum over. A candidate checksum index which passes all additional tests to determine its validity as a checksum index is called a *valid checksum index*. Once the set of valid checksum indices has been determined for all check statements, one of these may be picked as the variable to sum over. This will be referred to as the *chosen checksum index*.

The first condition involves dependence cycles which the statement may be involved in. As an example, consider the loop nest shown in Fig. 26, which is similar to the first assignment statement in Fig. 10, except that the left hand side array has been changed from `$a` to `$b`. Both `i` and `j` are candidate checksum indices for the statement; however, neither is a valid checksum index since the statement is involved in dependence cycles carried by both the `i` loop as well as the `j` loop. To see this, consider the code that would be generated if `j` were the chosen checksum index, which is shown in Fig. 27. The old value of `$cs2_b(i)` is used for

(1) Unary expressions $u = \pm e$ where $\pm$ denotes a generic unary operator

$$\begin{aligned} AFFINE(u) &\leftarrow AFFINE(e) \\ NOTAFFINE(u) &\leftarrow NOTAFFINE(e) \end{aligned}$$

(2) Binary addition or subtraction $b = e_1 + e_2$ or $b = e_1 - e_2$

$$\begin{aligned} AFFINE(b) &\leftarrow (AFFINE(e_1) - NOTAFFINE(e_2)) \cup (AFFINE(e_2) - NOTAFFINE(e_1)) \\ NOTAFFINE(b) &\leftarrow NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \end{aligned}$$

(3) Binary multiplication $b = e_1 * e_2$

$$\begin{aligned} AFFINE(b) &\leftarrow ((AFFINE(e_1) - NOTAFFINE(e_2)) \cup (AFFINE(e_2) - NOTAFFINE(e_1))) \\ &\quad -(AFFINE(e_1) \cap AFFINE(e_2)) \\ NOTAFFINE(b) &\leftarrow NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \cup (AFFINE(e_1) \cap AFFINE(e_2)) \end{aligned}$$

(4) Binary division $b = e_1 / e_2$

$$\begin{aligned} AFFINE(b) &\leftarrow AFFINE(e_1) - (NOTAFFINE(e_2) \cup AFFINE(e_2)) \\ NOTAFFINE(b) &\leftarrow NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \cup AFFINE(e_2) \end{aligned}$$

(5) Constant c

$$\begin{aligned} AFFINE(c) &\leftarrow \emptyset \\ NOTAFFINE(c) &\leftarrow \emptyset \end{aligned}$$

(6) Variable i. L is the set of potential checksum indices.

$$\begin{aligned} AFFINE(i) &\leftarrow \emptyset \\ NOTAFFINE(i) &\leftarrow \{i\}, \quad i \text{ is a loop variable} \\ &\qquad \cup_j \{j \in L\}, \quad \text{otherwise} \end{aligned}$$

(7) Array A. L is as before and S is the set of variables appearing in the array subscripts. A subscript expression is suitable if (a) if A appears on the left hand side, it is of the form i, else it is of the form $i \pm c$, where i is a loop variable, c is a constant, and i does not appear in the subscript expression for any other dimension; or (b) it is of the form c, where c is a constant.

$$\begin{aligned} AFFINE(A) &\leftarrow \cup_{i \in S \cap L} \{i\}, \quad \text{all subscript expressions are suitable} \\ &\qquad \emptyset, \quad \text{otherwise} \\ NOTAFFINE(A) &\leftarrow \emptyset, \quad \text{all subscript expressions are suitable} \\ &\qquad \cup_{j \in L} \{j\}, \quad \text{otherwise} \end{aligned}$$

(8) Assignment a of the form $e_1 \leftarrow e_2$

$$\begin{aligned} AFFINE(a) &= AFFINE(e_1) - NOTAFFINE(e_2) \\ NOTAFFINE(a) &= NOTAFFINE(e_1) \cup NOTAFFINE(e_2) \end{aligned}$$

Figure 11: Rules for computing $AFFINE$ and $NOTAFFINE$ sets in bottom-up fashion

```
    DO j = 2,999
      DO i = 2,999
        $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j + 1)
1       ) / 4 + 10
      END DO
    END DO
```

Figure 12: Code fragment illustrating affine transformation

```
    DO i = 2,999
      $cs2_a(i) = ($cs2_b(i - 1) + $cs2_b(i + 1) + ($cs2_b(i) - b(i
1     ,999) + b(i,1)) + ($cs2_b(i) + b(i,1000) - b(i,2))) / 4 + 10 * 998
    END DO
    DO j = 2,999
      DO i = 2,999
        $a(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j + 1)
1       ) / 4 + 10
      END DO
    END DO
```

Figure 13: Checksum code fragment illustrating affine transformation

both accesses `b(i,j+1)` as well as `b(i,j-1)`. However, `b(i,j-1)` actually uses values modified during the current iteration of the `i` loop, and thus the old checksum value does not correctly represent the sum over these elements.

Now we state and prove some theorems which indicate when a candidate checksum index is also a valid checksum index for the case of a single statement enclosed in a perfect nest of loops. This case actually covers a large fraction of the situations in practice since loop distribution is applied to the original code, which separates out statements not involved in dependence cycles into separate loop nests.

Theorem 1 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement S. Suppose I_1, the loop variable for the outermost loop L_1, is a candidate checksum index for S. Then I_1 is a valid checksum index for S if there is no flow dependence from S to itself*

Proof: Since there are no flow dependences from S to itself, all values used by S are assigned prior to the loop nest. Thus, the required checksums of the right hand side elements may be computed prior to entering the loop nest and may be transformed to generate the new checksum □

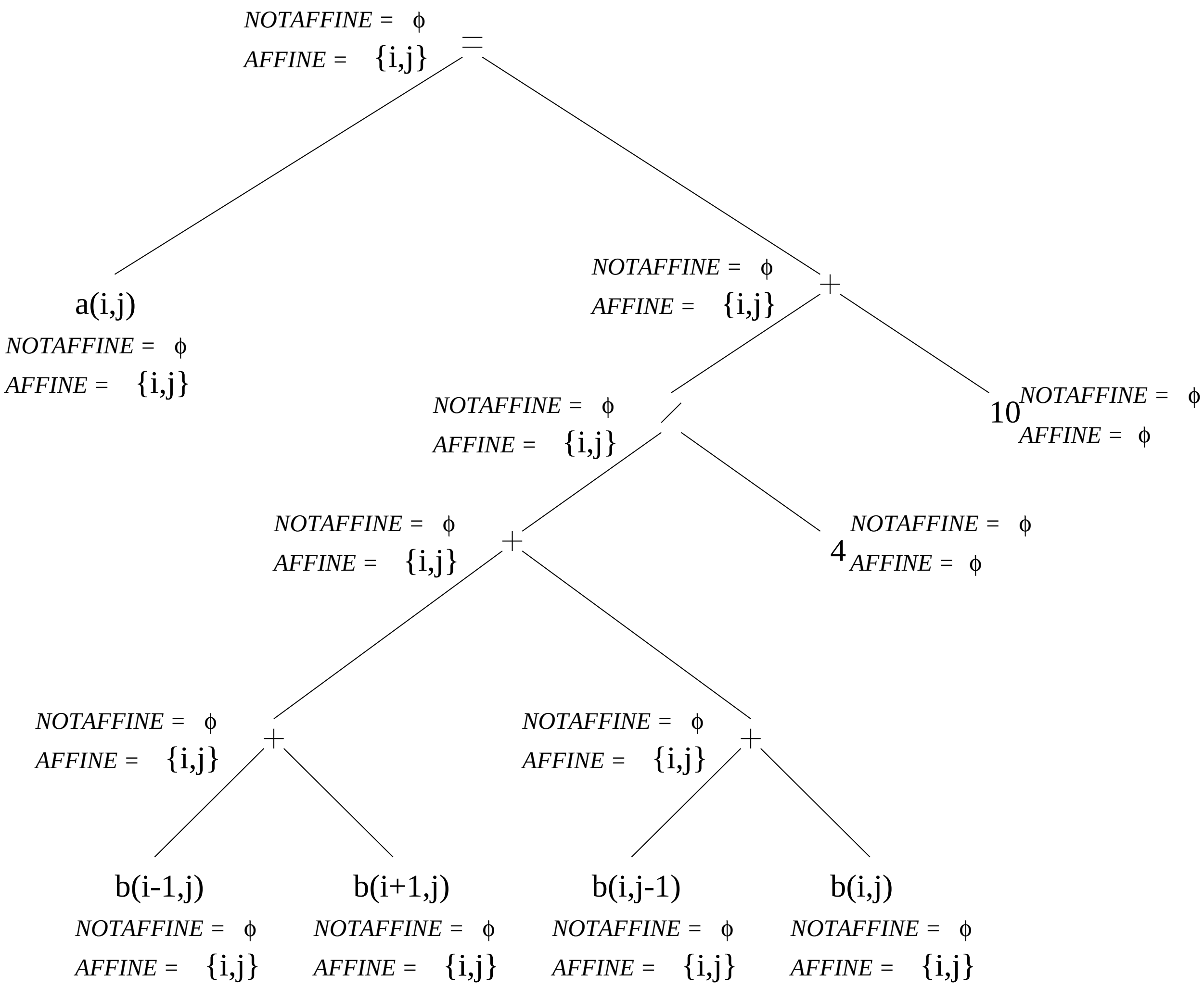

Figure 14: Syntax tree showing $AFFINE$ and $NOTAFFINE$ sets

```
                                    DO L1
                                      DO L2
                                        ⋮
Candidate checksum index  ------->        DO Lc
                                            ⋮
                                              DO Lm
                                                S
                                              ENDDO
                                            ⋮
                                          ENDDO
                                        ⋮
                                      ENDDO
                                    ENDDO
```

Figure 15: Loop nest with single assignment statement in loop body

For the next two theorems, a loop nest of the form shown in Fig. 15 is considered, with the loop variable for the cth loop L_c being a candidate checksum index. If this were chosen as the checksum index without verifying dependence conditions, the code shown in Fig. 16 would be generated, with CS being the checksum statement which would be generated for S. The next two theorems characterizes when the code of Fig. 16 correctly updates the checksums for checking the code in Fig. 15 when dependences from S to itself are taken into account.

Theorem 2 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement S. Suppose I_c, the loop variable for the cth loop L_c, is a candidate checksum index for S. Then I_c is a valid checksum index for S if there is no flow dependence from S to itself carried by loops at level c or greater*

Proof: We unroll the first $c-1$ loops, creating a separate loop nest for each value taken by I_j, $1 \leq j < c$. In each instance of the unrolled loop nests, we may apply Theorem 1 to conclude that I_c is a valid checksum index. Generating the checksum statement for each instance of the unrolled loops and rerolling to obtain the original loop ordering proves the theorem □

In the following theorem, condition (3) is only violated in rare cases. We mention when this happens and

```
DO L1
 DO L2
   ⋮
  DO Lc-1
   DO Lc+1
     ⋮
    DO Lm
     CS
    ENDDO
     ⋮
   ENDDO
   DO Lc
    DO Lc+1
      ⋮
     DO Lm
      S
     ENDDO
      ⋮
    ENDDO
   ENDDO
  ENDDO
   ⋮
 ENDDO
ENDDO
```

Figure 16: Check code for loop nest of Fig. 15

what to do in this case after the theorem.

Theorem 3 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement S. Suppose I_c, the loop variable for the cth loop L_c, is a candidate checksum index for S. Let CS be the checksum statement corresponding to S with I_c chosen as the checksum index. Then I_c is a valid checksum index for S if the following conditions are satisfied:*

(1) L_c does not carry any flow dependences.

(2) There is some valid reordering of the loops such that L_c can be moved inside all loops carrying flow dependences.

(3) There are no dependences between CS and S in the reordered loops.

Proof: It is clear that if the reordering of condition (2) exists, then Theorem 2 applies and I_c is a valid checksum index for the reordered loops. In the reordered loops, there are two types of dependences: dependences from S to itself and dependences from CS to itself. Dependences from CS to S and dependences from S to CS are excluded by condition (3) of the theorem. It is clear that if the loops are reordered back to their original ordering after CS has been introduced, the dependences from S to itself are not violated. Dependences from CS to itself are caused by the reading and writing of checksum variables. A checksum variable in CS is derived from the corresponding array variable in S and has identical subscript expressions except for the subscript involving I_c, which is missing. Thus, the dependence vectors from CS to itself are identical to the dependence vectors from S to itself except for the component corresponding to the loop L_c, which is missing. Thus, if interchanging loops I_j, I_k does not violate dependences from S to itself, then it also does not violate dependences from CS to itself. Thus, reordering the loops to obtain the original ordering after CS has been introduced does not violate dependences from CS to itself. Thus, reordering loops back to their original ordering after checksum introduction is valid since no dependences are violated. $\square$

Condition (3) of Theorem 3 may only be violated if the original statement assigns to and accesses the same array with different subscript expressions for the subscript involving the checksum index, as in the

```
DO I = 1,100
  DO J = 1,100
    A(I,J) = A(I+1,2) + A(I,J-1) * B(J)
  ENDDO
ENDDO
```

Figure 17: Code fragment illustrating necessity of loop reordering

```
DO J = 1,100
  CS1_A(J) = CS1_A(2) + A(101,2) - A(1,2) + CS1_A(J-1) * B(j)
  DO I = 1,100
    A(I,J) = A(I+1,2) + A(I,J-1) * B(J)
  ENDDO
ENDDO
```

Figure 18: Checksum introduction for the code in Fig. 17 after reordering (correct)

code in Fig. 17. This results in some of the array elements assigned to by the original code being added and subtracted off the checksum statement. This introduces flow dependences between the original statement and the checksum statement and antidependences from the latter to the former, which may prevent the reordering of the loops back into their original ordering after the checksum statements have been introduced. However, in this case, we may physically reorder the loops to obtain an ordering in which loops enclosed within the loop whose index is the candidate checksum index do not carry flow dependences, and then introduce the checksum statements. The point is illustrated in Figs. 17, 18 and 19. The code in Fig. 17 has a flow dependence carried by the `J` loop. However, the loops may be validly reordered to move the `J` loop outwards and the `I` may be chosen as the checksum index to obtain the code in Fig. 18. An attempt to obtain the original loop ordering after checksums have been introduced results in the code in Fig. 19. However, this clearly does not yield the same results as the code in Fig. 18 since the values of `A(1,2)` used by the checksum statement in the first case is the value assigned in the first iteration of the previous `I` loop, while the value used in the second case is the old value of `A(1,2)` before executing any iterations of the `I` loop.

Now we state and prove some theorems indicating when a candidate checksum index is a valid checksum index for a block of assignment statements enclosed within a perfect loop nest. However, before we can do

```
DO J = 1,100
  CS1_A(J) = CS1_A(2) + A(101,2) - A(1,2) + CS1_A(J-1) * B(J)
ENDDO
DO I = 1,100
  DO J = 1,100
    A(I,J) = A(I+1,2) + A(I,J-1) * B(J)
  ENDDO
ENDDO
```

Figure 19: Checksum introduction for the code in Fig. 17 without reordering (incorrect)

```
 DO i = 2,100
   a(i) = a(i) + c(i-1)
   b(i) = b(i) + a(i+1)
END DO
```

Figure 20: Code fragment illustrating backward dependence

this, we need to point out problems that can be caused by backward dependences. This is illustrated by the code fragment shown in Fig. 20. Here, there is a backward dependence (actually, an antidependence) from the second assignment statement to the first. Also, the loop variable `i` is a candidate checksum index for both assignment statements. Introducing checksum manipulations after choosing `i` as the checksum index, we obtain the code in Fig. 21. However, the checksum manipulations do not yield the desired checksum values. This is because `b(i)` uses values of `a` computed prior to the loop, while `$cs1_b` uses the newly transformed value of `$cs1_a`. Thus, a spurious flow dependence exists between the two checksum statements, while no such flow dependence exists between the two assignment statements in the original code fragment.

First we state an analog of Theorem 1 for the case when a block of statements is enclosed within a perfect

```
$cs1_a = $cs1_a + $cs1_c + c(1) - c(100)
$cs1_b = $cs1_b + $cs1_a + a(101) - a(2)
DO i = 2,100
  a(i) = a(i) + c(i-1)
  b(i) = b(i) + a(i+1)
END DO
```

Figure 21: Incorrect checksum code for code fragment in Fig. 20 illustrating problem caused by backward dependence

```
DO L1
  DO L2
    S1
    S2
  ENDDO
ENDDO
```

Figure 22: Loop nest with multiple assignment statements in loop body

loop nest. After proving the theorem, we indicate when condition (3) is violated and what to do in this case.

Theorem 4 *Consider a perfect nest of loops consisting of loops $L_1, L_2, \ldots, L_m$ enclosing assignment statement $S_1, S_2, \ldots, S_n$, with S_i lexically preceding S_j if $i < j$. Suppose I_1, the loop variable for the outermost loop L_1, is a candidate checksum index for each S_i. Let CS_i be the checksum statement corresponding to S_i with I_1 chosen as the checksum index. Then I_1 is a valid checksum index for each S_i if all the following conditions are satisfied:*

(1) No S_i is involved in a dependence cycle

(2) There is no dependence from S_i to S_j if $i > j$

(3) CS_i does not access any array elements assigned by S_j for any $1 \leq i, j \leq m$

Proof: The proof is sufficiently illustrated by the case when $n = 2$, i.e., there are two statements enclosed within the loop nest. This is illustrated in Fig. 22. Since S_1 and S_2 are not involved in a dependence cycle by condition (1), and all dependences are from S_1 to S_2 by condition (2), loop distribution may be applied to yield the loop nests in Fig. 23. Theorem 1 then applies to each loop nest in Fig. 23. Thus, I_1 is a valid checksum index for each loop nest in Fig. 23. Introduction of the checksum statements for the loop nests in Fig. 23 yields the code in Fig. 24. By condition (3), no dependences exist between the CS_i's and S_j's in Fig. 24. Applying loop fusion to Fig. 24 yields the code in Fig. 25. Since this is precisely what would be generated upon choosing I_1 as the checksum index, we conclude that I_1 is a valid checksum index $\square$

Condition (3) may be violated if some of the checksum statements need to add or subtract off array variables in order to adjust the checksum value, and these variables are assigned to by some of the original statements in the loop. In this case, however, loop distribution could be used to separate out the statements

```
DO L1
  DO L2
    S1
  ENDDO
ENDDO

DO L1
  DO L2
    S2
  ENDDO
ENDDO
```

Figure 23: Loop distribution applied to loop nest of Fig. 22

```
DO L2
  CS1
ENDDO

DO L1
  DO L2
    S1
  ENDDO
ENDDO

DO L2
  CS2
ENDDO

DO L1
  DO L2
    S2
  ENDDO
ENDDO
```

Figure 24: Introduction of checksum statements for loop nests of Fig. 23

```
DO L2
 CS1
 CS2
ENDDO

DO L1
 DO L2
  S1
  S2
 ENDDO
ENDDO
```

Figure 25: Loop fusion applied to loop nests of Fig. 24

in the body of the loop nest into separate loop nests, and Theorem 1 applied to generate checksum statements for each of the loop nests. This would result in code resembling that in Fig. 24, with checksum statements alternating with original statements.

Theorem 5 *Consider a block of statements $S_1, S_2, \ldots, S_n$ enclosed within a nest of loops $L_1, L_2, \ldots, L_m$. Suppose the loop index of the cth loop, I_c, is a candidate checksum index for each statement S_i, $1 \leq i \leq n$. Then I_c is a valid checksum index if the following three conditions hold*

(1) No S_i is involved in a dependence cycle involving dependences carried solely by loops L_j, $c \leq j \leq m$

(2) There is no dependence from S_i to S_j which is loop independent or carried by loops L_k, $c \leq k \leq m$, if $i > j$

(3) CS_i does not access any array elements assigned by S_j for any $1 \leq i, j \leq m$

Proof: The proof proceeds by first unrolling the first $c-1$ loops in the loop nest and then applying Theorem 4 to the resulting loop nests. Condition (3) is then used to separate out the CS_is to a separate loop nest, and loop fusion is applied to recover the original loop nest enclosing the S_is □

As before, if condition (3) is violated, loop distribution may be applied to the loops at level c and deeper, and checksum statements may be generated embedded within the outer $c-1$ levels of loops.

Often, even if the conditions stated in Theorems 2 and 5 are violated because of cycles of dependences

```
      DO j = 2,999
        DO i = 2,999
          $b(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j
     1      + 1)) / 4
        END DO
      END DO
```

Figure 26: Code fragment illustrating dependence cycle

```
      DO i = 2,999
        $cs2_b(i) = ($cs2_b(i - 1) + $cs2_b(i + 1) + ($cs2_b(i) - b(i
     1  ,999) + b(i,1)) + ($cs2_b(i) + b(i,1000) - b(i,2))) / 4
      END DO
      DO j = 2,999
        DO i = 2,999
          $b(i,j) = ($b(i - 1,j) + $b(i + 1,j) + $b(i,j - 1) + $b(i,j
     1      + 1)) / 4
        END DO
      END DO
```

Figure 27: Checksum code illustrating problem caused by dependence cycle (incorrect code)

carried by some loops inner to the loop whose index variable is the candidate checksum index (which we will refer to as the *candidate checksum loop*), it may be possible to use loop reordering to move the candidate checksum loop inside all loops carrying dependences. The candidate checksum index then becomes a valid checksum index for the reordered loops.

Similarly, the condition prohibiting backward dependences may be enforced by reordering some statements in the loop body. Thus, the statements in the `i` loop in Fig. 20 may be validly reordered, since there is a loop carried antidependence from the second statement to the first, but no dependence from the first statement to the second. Generating the checksum statements for the reordered code yields the correct results since the checksum statements are also generated in reordered fashion.

Once the set of valid checksum indices has been determined for each statement, one of them is chosen as the index to compute the checksum over. For example, in the code in Fig. 5, the chosen checksum index is `j`, which corresponds to the second dimension of the arrays `$a` and `$b`. Once a valid checksum index has been chosen as the index to sum over for a statement S, the corresponding checksum statement is generated in the following manner. The intermediate nodes for the syntax tree for S are annotated with the $AFFINE$ and $NOTAFFINE$ sets which were computed while computing the candidate checksum indices for S while traversing the the tree in bottom-up fashion. Now, the syntax tree is traversed in top-down fashion in order to determine subexpressions which do not involve the checksum index `j`. This is indicated by the fact that the $AFFINE$ set associated with the node in the syntax tree which is the root of the subexpression does not contain `j`. The entire subexpression is then multiplied by the number of times the `j` loop is executed. The algorithm for expanding constants in expressions is shown in Fig. 28. Only the rules for the commonly occurring operators are shown for brevity. As an example, on using `expand_constants` on the tree of Fig. 14, the only node encountered with `j` absent from its $AFFINE$ set is the node for 10. Thus, this results in the expression 10 * 998 in the corresponding checksum statement, which is shown in Fig. 13.

After constants have been expanded in affine expressions, arrays used by the expression which involve the checksum index as a subscript need to be replaced by checksums. These arrays may be found by a top-down traversal of the syntax tree and replaced by a checksum variable with the same subscript expressions in all dimensions except the one involving the checksum index, which vanishes. However, a correction needs to be made to the checksum variable in the event that the subscript involving the checksum index, say `j`, is of the form `j+c` or `j-c`, where `c` is a positive constant. This point is illustrated by the code in Fig. 12 and the corresponding check code in Fig. 13. We assume that upon entering the `j` loop, the checksum of `b(i,j)`, for `j` ranging from 2 to 999 (the values taken by the `j` loop), are available. However, the checksum over `b(i,j+1)` is required. This may be derived from the checksum over `b(i,j)` by subtracting and adding one element, as illustrated by the code in Fig. 13. The other accesses to `b` in the right hand side expression are similarly replaced by checksums incorporating the addition and subtraction of some extra elements. The information propagation pass is responsible for making available the checksums over `b(i,2:999)` at the entry to the `j` loop.

```
expand_constants(expr, ci, numiter)
/* expr is an expression ;
   ci is a checksum index ;
   numiter is the number of iterations
   executed by the loop enclosing expr
   with ci as its index variable */
{
   if(! in_set(ci, affine_set(expr))
      return build_binary_op(TIMES, numiter, expr);

   switch(TYPE(expr))
   {
     case BIPLUS:
     case BIMINUS:
       return build_binary_op(TYPE(expr),
                              expand_constants(left_op(expr), ci, numiter),
                              expand_constants(right_op(expr), ci, numiter));

     case BITIMES:
       if(in_set(ci, affine_set(left_op(expr))
         return  build_binary_op(BITIMES,
                              expand_constants(left_op(expr), ci, numiter),
                              right_op(expr));
      else
        return build_binary_op(BITIMES,
                              left_op(expr),
                              expand_constants(right_op(expr), ci, numiter);

     case BIDIVIDE:
       return build_binary_op(BIDIVIDE,
                              expand_constants(left_op(expr), ci, numiter),
                              right_op(expr));

     case ARRAY_REF:
     case VARIABLE:
       return copy_expr(expr);
     }
 }
```

Figure 28: Algorithm for expanding constants in affine expressions

```
change = TRUE;
while(change)
{
  change = FALSE;
  for(i = 0; i < nfgblks; i++)
  {
     in[i] =  ∩   out[j];
          j ε fgpred(i)
     oldout = out[i];
     out[i] = gen[i]  ∪  (in[i] - kill[i]);
     if(sets_not_equal(out[i], oldout)) change = TRUE;
  }
}
```

Figure 29: Outline of generic iterative dataflow algorithm

5.3 Information Propagation and Check Generation

After checksum manipulation statements have been introduced, the information propagation pass is run. The pass may be divided into two stages. In the first stage, an iterative dataflow algorithm is executed to determine the checksum and array values available at various points in the program. In the second stage, the information about available checksums and arrays is used to regenerate checksums and arrays as required. We now explain each of these stages in detail.

The outline of the basic iterative dataflow algorithm is shown in Fig. 29. For a more detailed description of the iterative dataflow approach, see [19].

Now we discuss the specifics of the algorithm as applied to our problem, viz., computing the ranges of checksums and arrays available at each block in the flowgraph.

The flowgraph for the code in Fig. 5 is shown in Fig. 30. Note that the fact that the loops are non-zero trip has been used in constructing the flowgraph. Also, a dummy start node has been inserted.

We introduce two sets, called $AVAILARRAY$ and $AVAILCS$ with every node in the flowgraph. $AVAILARRAY$ and $AVAILCS$ store the ranges of the shadow arrays and checksums which are available at the end of the block of statements comprising the flowgraph node. Associated with every node which is a loop

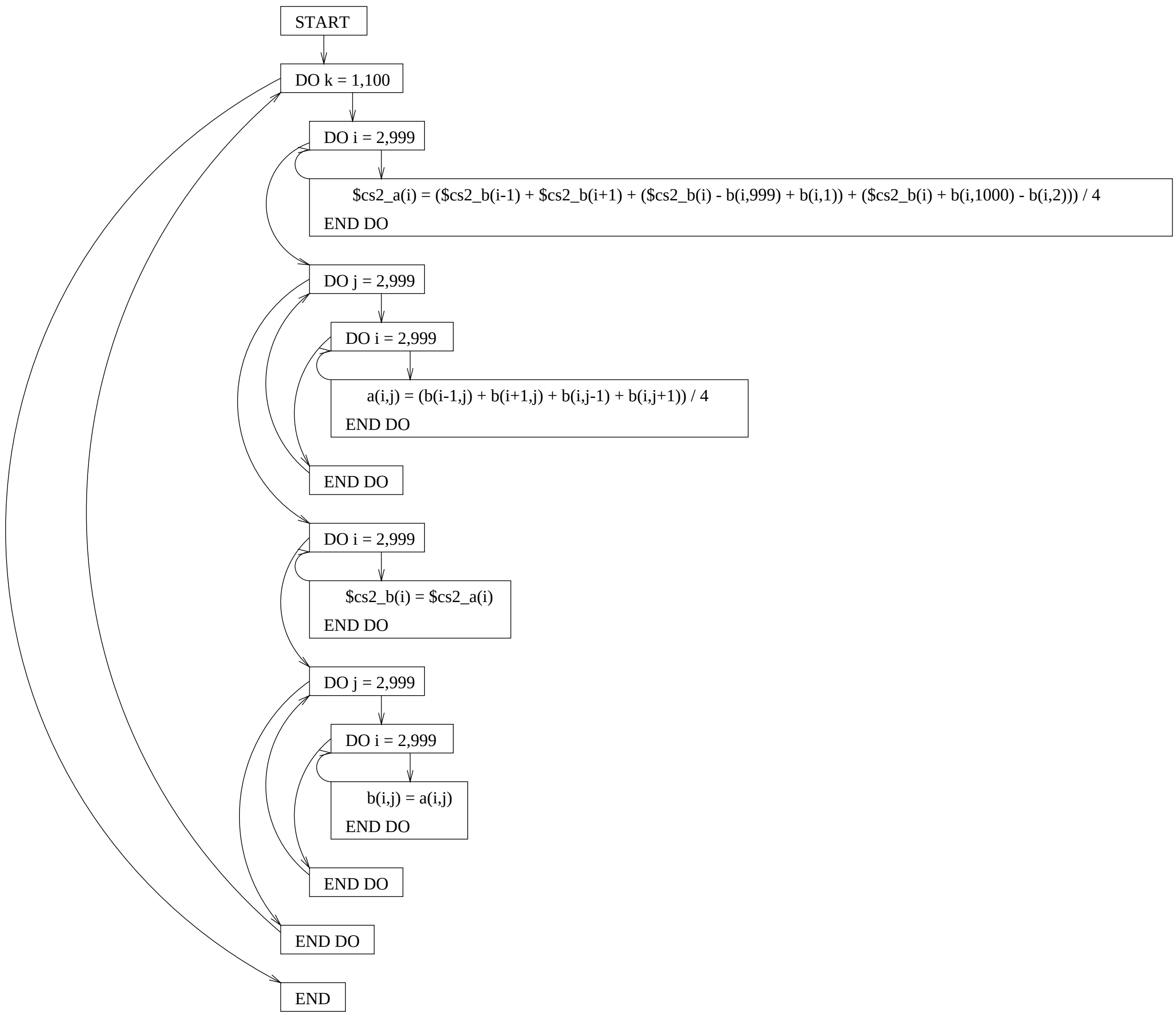

Figure 30: Control flow graph for Jacobi code with checksums

header are two more sets, which we call $AVAILARRAY\_ON\_DO\_EXIT$ and $AVAILCS\_ON\_DO\_EXIT$, which store the arrays and checksums available when the loop is finally exited. The latter two sets are introduced to ensure a less conservative computation of available arrays and checksums than would otherwise occur.

In order to conveniently propagate range information, we ensure that there is only one entry per variable in each of the above sets, and the ranges covered by any entry cover a contiguous portion of the array. We ensure this by making conservative choices, if necessary, on updating the sets.

Prior to executing the iterative algorithm, the above mentioned sets are initialized for every node in the flowgraph. Essentially, the statements in a basic block are traversed in lexical order and the sets updated for each statement as it is encountered. Initially, at the start of each basic block, the sets are initialized to the empty set. We make an exception in the case of the start node in the flowgraph. For this node, the $AVAILARRAY$ set is initialized to include all the shadow arrays which have been introduced for the program. The $AVAILCS$ set is initialized to include all the checksum variables which occur in the program, with the ranges computed from the first occurrence of the variable as the program is traversed in lexical order.

For each iteration of the dataflow algorithm, all the nodes in the flowgraph, except the dummy start node, are traversed one after the other (A depth-first ordering may be used for efficiency [19]). Upon entry to a basic block associated with a flowgraph node, the initial values of $AVAILARRAY$ and $AVAILCS$ are computed by taking an intersection of the $AVAILARRAY$ and $AVAILCS$ sets arriving along the incoming edges to the node. However, an exception is made in the case that the node is one which follows an exit from a loop. In this case, instead of the $AVAILARRAY$ and $AVAILCS$ sets associated with the loop header, the $AVAILARRAY\_ON\_DO\_EXIT$ and the $AVAILCS\_ON\_DO\_EXIT$ sets are used in computing the intersection. Similarly, in the event that the node under consideration is a loop header itself, and the loop is not zero-trip, the $AVAILARRAY\_ON\_DO\_EXIT$ and the $AVAILCS\_ON\_DO\_EXIT$ are set to the $AVAILARRAY$ and $AVAILCS$ sets, respectively, which are available along the backedge. In the event that we cannot determine if the loop is always non-zero trip, the $AVAILARRAY\_ON\_DO\_EXIT$ and the

$AVAILCS\_ON\_DO\_EXIT$ sets are set to the $AVAILARRAY$ and $AVAILCS$ sets associated with the loop header, which are computed by taking intersections of all the $AVAILARRAY$ and $AVAILCS$ on the incoming edges (which also includes the backedge).

For each statement encountered in a basic block, the sets are updated in the following manner. Only check statements result in the sets being updated, and different actions are taken for check statements which are checksum statements and statements which are duplicates of the original statement, operating on shadow arrays. First, we discuss the update equations for the sets when a checksum statement *cksumstmt* is encountered. Let the checksum variable assigned to by *cksumstmt* be `$cs1_a`. Let the original array variable corresponding to `$cs1_a` be `a`. First, entries for all checksum variables corresponding to `a`, except possibly an earlier entry for `$cs1_a`, are removed. An earlier entry for `$cs1_a` will also be removed if the ranges covered by the checksum dimension in the set and in the statement are not identical. Next, the ranges for each dimension are computed from the bounds of the loops enclosing *cksumstmt* and the subscript expressions for `$cs1_a`. Recall that the subscript corresponding to the checksum index is removed from the checksum variable; this subscript expression is determined from the variable being assigned to by the original statement corresponding to this check statement. The set $AVAILCS$ is updated to include `$cs1_a` if it doesn't already contain an entry for `$cs1_a`. If $AVAILCS$ already contains an entry for `$cs1_a`, the newly computed ranges are merged with the old range information. If the two ranges are disjoint, then new entry replaces the old only if it covers a larger portion of the array. This is a conservative criterion enforced due to our requirement that there be a single entry for each variable in each set at any time. Also, if the set $AVAILARRAY$ contains the shadow array variable, say `$a`, corresponding to `$cs1_a`, then the ranges covered by `$cs1_a` which were entered into $AVAILCS$ are removed from the ranges covered by `$a` in $AVAILARRAY$. If this results in an empty range in some dimension or in fragmentation of the ranges, then the entry for `$a` is removed from $AVAILARRAY$.

If instead of a checksum statement, a duplicate statement is encountered, two cases need to be distinguished. The first case occurs when all enclosing loop bounds are constants, and all subscript expressions occurring in the statement are of the form i, $i + c$ or $i - c$, where i is a variable and c is a constant (recall that these same restrictions must be satisfied by a checksum statement). In this case, the range covered by the left hand side variable, say `$a`, is determined. `$a` and the range covered by it are entered into $AVAILARRAY$, or are merged with the range information already in $AVAILARRAY$ if there is already an entry in $AVAILARRAY$ for `$a`. If there is an entry for a checksum variable corresponding to `$a` (such as `$cs1_a` or `$cs2_a`, for example) in $AVAILCS$, the ranges covered by `$a` which were added to the $AVAILARRAY$ set are removed from the corresponding checksum variable entry in $AVAILCS$. If this leads to some dimension becoming empty or fragmented, the checksum variable entry is removed from $AVAILCS$.

The second case occurs when the duplicate statement's subscript expressions or the bounds of the loops enclosing it do not satisfy the conditions mentioned earlier. In this case, all arrays accessed by the statement are added to the $AVAILARRAY$ set, with the ranges covering the entire array (Code copying the entire original array into the corresponding shadow array will be generated just prior to the execution of the statement by the second stage of the propagate pass). All checksum variables corresponding to the shadow arrays added to $AVAILARRAY$ are removed from $AVAILCS$.

The rules for updating the $AVAILARRAY$ and $AVAILCS$ sets in each iteration of the flowgraph are shown in Fig. 31. Some of the details discussed in the previous paragraphs have been omitted from the figure.

Once the dataflow algorithm has converged, the second stage of the pass, which involves regeneration of checksums and shadow elements, is performed. First, one more pass over the flowgraph is used to compute the $AVAILARRAY$ and $AVAILCS$ sets for each individual statement in the program, rather than the final values at the end of each basic block. The list of statements comprising the program is then traversed in lexical order. Recall that checksum statements are enclosed in perfect loop nests whose bodies consist solely of checksum statements. When such a loop nest is encountered, the checksums which are used by each statement in the body are determined by traversing the syntax tree representing the right hand side of each statement, collecting the checksum variables which appear, and computing the ranges covered from the subscript expressions and the enclosing loop bounds. The set of these checksums is denoted by $REQDCS$.

Update rules upon entering node n of the flowgraph

$$\begin{aligned} AVAILCS(n) &\leftarrow \cap_{j \in fgpred(n)} AVAILCS(j) \\ AVAILARRAY(n) &\leftarrow \cap_{j \in fgpred(n)} AVAILARRAY(j) \end{aligned}$$

Update rules for a checksum statement CS

$$\begin{aligned} GENCS &\leftarrow \text{all checksum elements accessed by } CS \\ AVAILCS &\leftarrow AVAILCS \cup GENCS \\ KILLARRAY &\leftarrow \text{all shadow array elements covered by checksum elements} \\ & \quad\ \text{assigned to by } CS \\ AVAILARRAY &\leftarrow AVAILARRAY - KILLARRAY \end{aligned}$$

Update rules for a duplicate check statement DC

$$\begin{aligned} GENARRAY &\leftarrow \text{all shadow array elements accessed by } DC \\ AVAILARRAY &\leftarrow AVAILARRAY \cup GENARRAY \\ KILLCS &\leftarrow \text{all checksum elements covering any portion of the shadow array} \\ & \quad\ \text{elements assigned to by } DC \\ AVAILCS &\leftarrow AVAILCS - KILLCS \end{aligned}$$

Figure 31: Rules for updating $AVAILARRAY$ and $AVAILCS$ sets

The values which actually need to be regenerated at the start of the loop nest (which we denote by $GENCS$) are determined by subtracting the $AVAILCS$ set at the entry to the loop nest enclosing the checksum statements from the $REQDCS$ set. which are used by the statement. Once $GENCS$ has been determined for the loop body, code for recomputing these checksums is inserted at the beginning of the loop body. Also, it is determined if the shadow array values for the array values which are summed to regenerate the checksums are available upon entry to the loop nest. If so, code for performing a comparison check of the array values being summed and the corresponding shadow array values is inserted prior to the loop nest. These rules are summarized in Figs. 32.

An example code fragment with values of the various sets is shown in Fig. 33 and the check code that would be generated for it is shown in Fig. 34.

Loop nests which enclose check statements (but not checksum statements) are handled in a different manner. As before, the entire body of the loop nest is traversed. For each assignment statement encountered in the loop nest, the shadow array variables and ranges are computed from the expression tree for the

For a loop nest enclosing checksum statements

$$
\begin{aligned}
REQDCS &\leftarrow \text{all checksum elements used by all checksum} \\
&\quad\ \text{statements in loop body} \\
GENCS &\leftarrow REQDCS - AVAILCS \\
SUMVALUES &\leftarrow \text{array values covered by } GENCS \\
CHECKVALUES &\leftarrow SUMVALUES \cap AVAILARRAY
\end{aligned}
$$

Regenerate checksum elements in $GENCS$ by summing the corresponding array values.

Compare shadow array values in $CHECKVALUES$ with original array values.

For a loop nest enclosing duplicate check statements

$$
\begin{aligned}
REQDARRAY &\leftarrow \text{all shadow array elements used by all check} \\
&\quad\ \text{statements in loop body} \\
GENARRAY &\leftarrow REQDARRAY - AVAILARRAY \\
SUMCS &\leftarrow \text{all checksums that can be generated from } GENARRAY \\
CHECKCS &\leftarrow SUMCS \cap AVAILCS
\end{aligned}
$$

Regenerate shadow array elements in $GENARRAY$ by copyinging the corresponding array values.

Generate the checksums in $CHECKCS$ by summing the original array values they cover.

Compare the checksums in $CHECKCS$ with the corresponding checksums in $AVAILCS$.

Figure 32: Rules for regenerating checksums and shadow arrays

```
C AVAILARRAY = {$B(6:10,1:10)}, AVAILCS = {$CS2_B(1:5,1:10)}
C REQDCS = {$CS2_B(1:10,1:10)}
C GENCS = REQDCS - AVAILCS = {$CS2_B(6:10,1:10)}

       DO I = 1,10
         $CS2_A(I) = $CS2_B(I) + 10*10
       ENDDO

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

Figure 33: Code fragment for checksum regeneration showing $AVAILARRAY$, $AVAILCS$, $REQDCS$, and $GENCS$ sets

```
C AVAILARRAY = {$B(6:10,1:10)}, AVAILCS = {$CS2_B(1:5,1:10)}
C REQDCS = {$CS2_B(1:10,1:10)}
C GENCS = REQDCS - AVAILCS = {$CS2_B(6:10,1:10)}

C REGENERATE CHECKSUMS

      DO I = 6,10
        $CS2_B(I) = 0
        DO J = 1,10
          $CS2_B(I) = $CS2_B(I) + B(I,J)
        ENDDO
      ENDDO

C CHECK ELEMENTS WHICH WERE ADDED

      DO I = 6,10
        DO J = 1,10
          IF (COMPARE($B(I,J),B(I,J)) .EQ. 1)
            CALL ERROR_HANDLER
        ENDDO
      ENDDO

      DO I = 1,10
        $CS2_A(I) = $CS2_B(I) + 10*10
      ENDDO

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

Figure 34: Checksum regeneration for code fragment in Fig. 33

```
C AVAILARRAY = {$B(1:5,1:10)}, AVAILCS = {$CS2_B(6:10,1:10)}
C REQDARRAY = {$B(1:10,1:10)}, GENARRAY = {$B(6:10,1:10)}

      DO I = 1,10
        DO J = 1,10
          $A(I,J) = $B(I,J)*$B(I,J)
        ENDDO
      ENDDO

      DO I = 1,10
        DO J = 1,10
          A(I,J) = B(I,J)*B(I,J)
        ENDDO
      ENDDO
```

Figure 35: Code fragment for shadow array regeneration showing $AVAILARRAY$, $AVAILCS$, $REQDARRAY$ and $GENARRAY$ sets

statement, the subscript expressions and the loop bounds. If the ranges cannot be computed due to the loop bounds or subscript expressions being complicated, or because the statement is not enclosed in a perfect loop nest, it is assumed that the entire array is used by the statement. The array elements accessed by check statements within the loop nest are stored in a set called $REQDARRAY$. The array elements which are actually required by the statement (which we store in a set called the $GENARRAY$) are computed by subtracting the entries in the $AVAILARRAY$ set for the statement from the $REQDARRAY$ set for the loop nest. Code for copying over the values of the corresponding original array elements into the shadow array elements in $GENARRAY$ is then generated prior to entering the loop nest. The $AVAILCS$ set for the loop header is examined to determine if any checksum variables are available to check the array elements being copied over. If this is the case, then code is also inserted to sum the elements being copied over and perform a comparison check against the available checksum values. The code fragment in Fig. 35 shows the $AVAILARRAY$, $AVAILCS$, $REQDARRAY$ and $GENARRAY$ sets associated with a loop nest enclosing a nonlinear check statement and the check code corresponding to this code fragment is shown in Fig. 36.

As an example, the values of $AVAILARRAY$ and $AVAILCS$ after convergence are shown for selected edges of the control flow graph of Fig. 30 in Fig. 37.

```
C AVAILARRAY = {$B(1:5,1:10)}, AVAILCS = {$CS2_B(6:10,1:10)}
C REQDARRAY = {$B(1:10,1:10)}, GENARRAY = {$B(6:10,1:10)}

C COPY ARRAY ELEMENTS INTO SHADOW ARRAYS

       DO I = 6,10
         DO J = 1,10
           $B(I,J) = B(I,J)
         ENDDO
       ENDDO

C CHECK ELEMENTS WHICH WERE COPIED

       DO I = 6,10
         T = 0
         DO J = 1,10
           T = T + B(I,J)
         ENDDO
         IF (COMPARE($CS2_B(I),T) .EQ. 1)
           CALL ERROR_HANDLER()
       ENDDO

       DO I = 1,10
         DO J = 1,10
           $A(I,J) = $B(I,J)*$B(I,J)
         ENDDO
       ENDDO

       DO I = 1,10
         DO J = 1,10
           A(I,J) = B(I,J)*B(I,J)
         ENDDO
       ENDDO
```

Figure 36: Shadow array regeneration for code fragment in Fig. 35

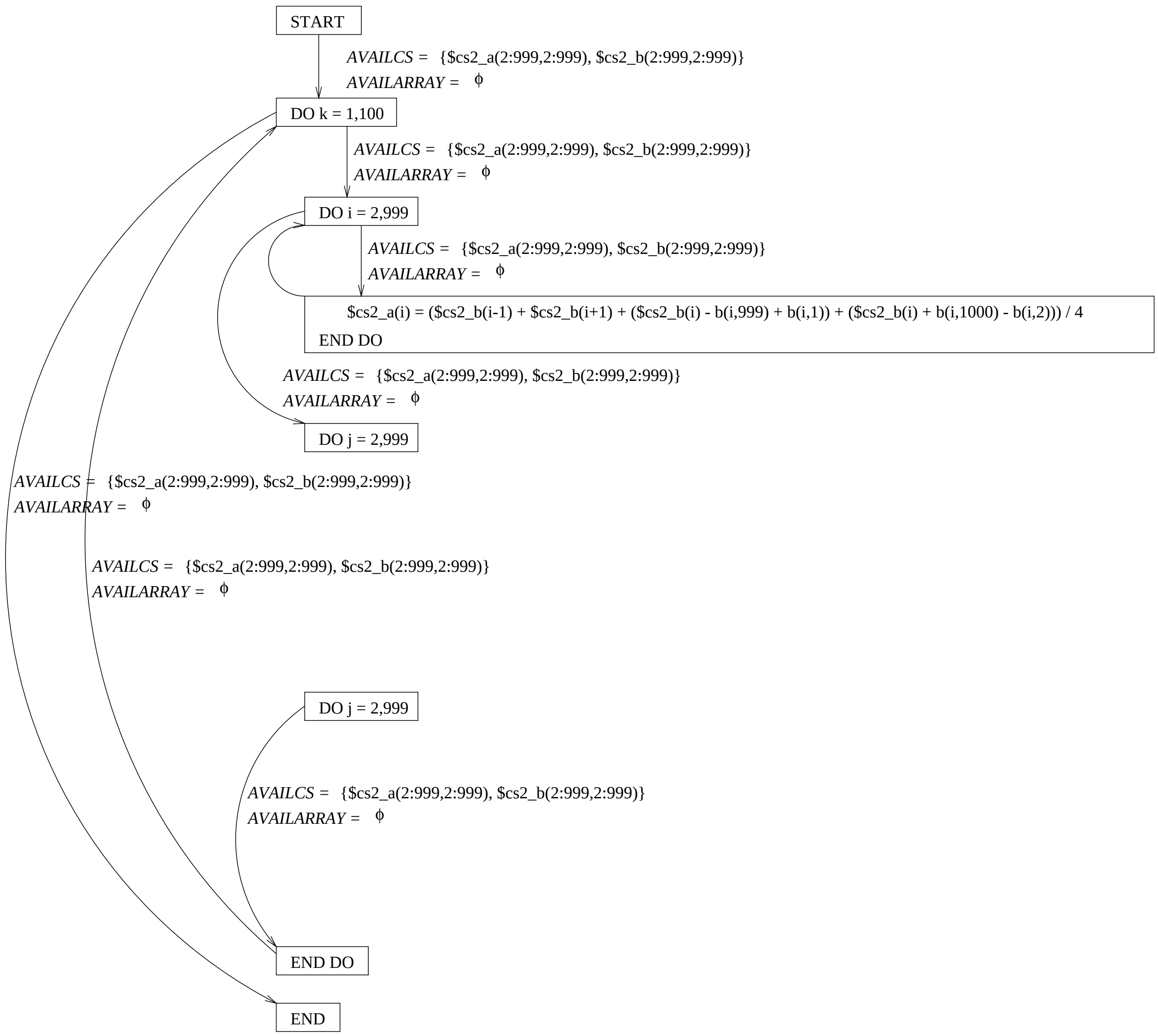

Figure 37: Final values of $AVAILARRAY$ and $AVAILCS$ on selected edges of the flow graph of Fig. 30

```
      DOUBLE PRECISION b(64,64)
      DOUBLE PRECISION $b(64,64)

!HPF$ PROCESSORS :: p(4)
!HPF$ DISTRIBUTE (*, BLOCK) ONTO p :: b
!HPF$ TEMPLATE, DISTRIBUTE (*, BLOCK) ONTO p :: template$0(64, 64)
!HPF$ ALIGN (:,:) WITH template$0(:,:) :: b
!HPF$ ALIGN (:,hpf$0) WITH template$0(:,hpf$0 + 16) WRAP :: $b
```

Figure 38: Data distribution specification for a block distributed array

5.4 Data Distribution Specification for Check Data

In most of the following examples, we will concentrate on block distributed arrays; the ideas behind handling arrays which are distributed in a cyclic or block-cyclic fashion are similar.

Data distribution specification for check data (checksums and shadow arrays) needs to be specified so that the original data and the data checking it reside on different processors. This, together with the owner computes rule, ensures that each data element is subjected to a check on a different processor, thus increasing the likelihood of detecting single processor faults. Essentially, in the case of shadow arrays, a distribution is chosen which is almost identical to the distribution of the corresponding original array, except that the data elements in one of the distributed dimensions are shifted cyclically so that a data element and the corresponding shadow element reside on different processors. This is indicated for a block distributed array in Figs. 38 and 39. Note that the `WRAP` directive is used to specify that the elements which "fall off the end" of the template are to wrap around to the first processor. `WRAP` is not a standard HPF directive; however, the same effect can be achieved in a somewhat roundabout manner by using only standard HPF. Instead, we use `WRAP` for brevity.

In order to determine how a checksum variable is to be distributed, we first determine how the shadow array variable corresponding to the original array being checked by the checksum, would be distributed. Two cases are distinguished. The first correspponds to the case when the dimension being summed over is sequentialized. In this case, the other dimensions of the checksum are distributed in a manner identical to the distribution of the shadow array. This case is illustrated in Figs. 40 and 41.

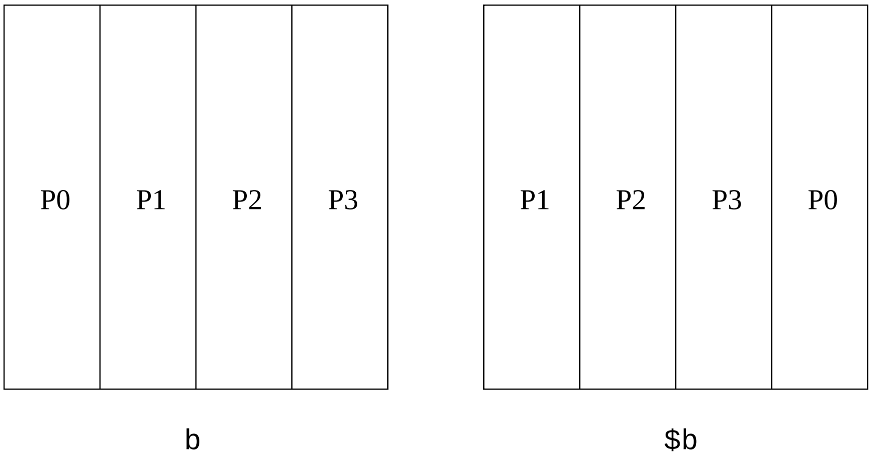

Figure 39: Illustration of data distribution for declaration in Fig. 38

```
       DOUBLE PRECISION c(64,64)
       DOUBLE PRECISION $c(64,64)
C $cs2_c is obtained by summing the second dimension of c
       DOUBLE PRECISION $cs2_c(64)

!HPF$ PROCESSORS :: p(4)
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$0(64, 64)
!HPF$ ALIGN (:,:) WITH template$0(:,:) :: c
!HPF$ ALIGN (hpf$0,:) WITH template$0(hpf$0 + 16,:) WRAP :: $c
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO p :: TEMPLATE$1(64)
!HPF$ ALIGN $cs2_c(hpf$0) WITH TEMPLATE$1(hpf$0+16) WRAP
```

Figure 40: Checksum data distribution when the dimension being summed over is sequentialized

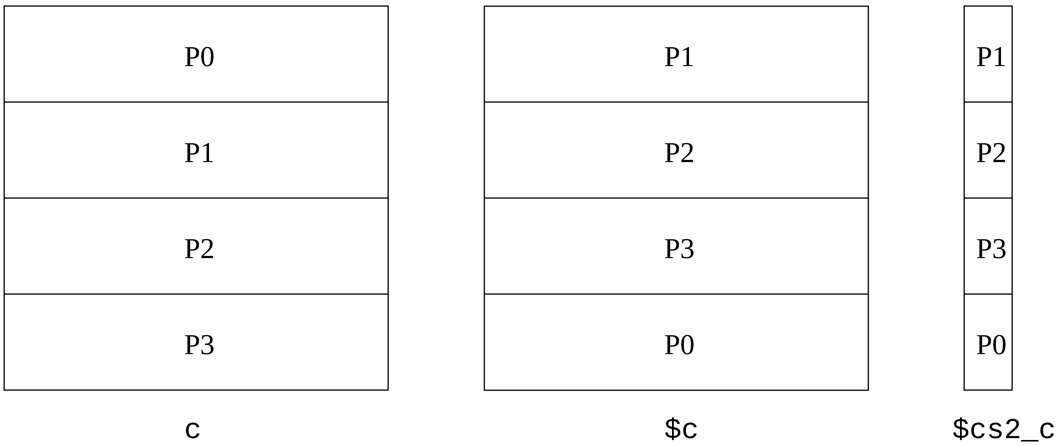

Figure 41: Illustration of data distribution for declaration in Fig. 40

```
       DOUBLE PRECISION c(64,64)
       DOUBLE PRECISION $c(64,64)
C $cs1_c is obtained by summing the first dimension of c
       DOUBLE PRECISION $cs1_c(4,64)

!HPF$ PROCESSORS :: p(4)
!HPF$ DISTRIBUTE (BLOCK, *) ONTO p :: c
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK, *) ONTO p :: template$0(64, 64)
!HPF$ ALIGN (:,:) WITH template$0(:,:) :: c
!HPF$ ALIGN (hpf$0,:) WITH template$0(hpf$0 + 16,:) WRAP :: $c
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ONTO p :: TEMPLATE$1(4,64)
!HPF$ ALIGN $cs1_c(hpf$0,:) WITH TEMPLATE$1(hpf$0+1,:) WRAP
```

Figure 42: Checksum data distribution when the dimension being summed over is distributed

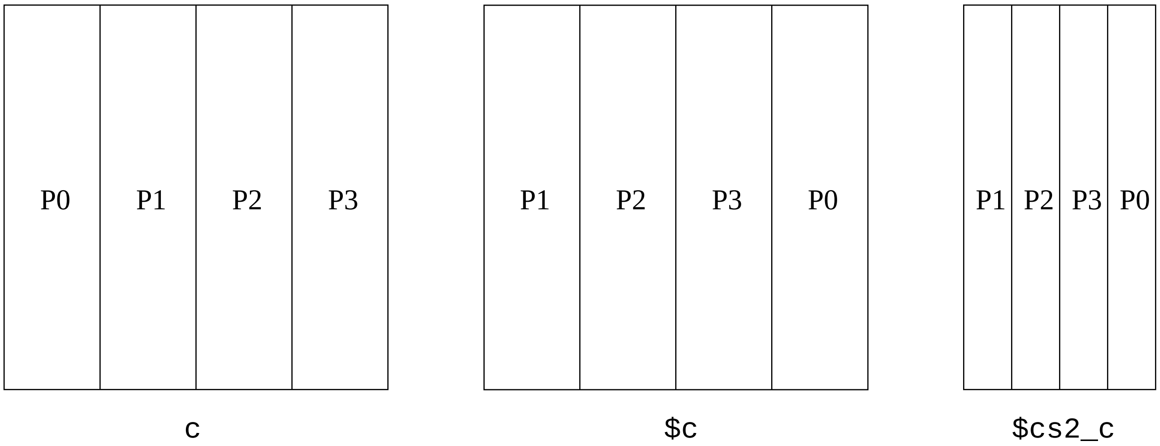

Figure 43: Illustration of data distribution for declaration in Fig. 42

The second case occurs when the dimension which was summed over was not sequentialized but distributed. In this case, the checksum is expanded to include an extra dimension corresponding to the dimension being summed over, with the number of elements in the expanded dimension being equal to the number of processors in that dimension. The expanded dimension is then distributed so that each processor gets one element in each dimension, with the WRAP directive being specified if the corresponding shadow array would have been wrapped around for this dimension. This case is illustrated in Fig. 42 and 43.

In the case of a block distribution, each checksum element now stores the sum over a contiguous block of elements in the dimension being summed over, instead of the sum over the entire dimension. This may require the replacement of checksum manipulation statements in the code by statements manipulating the new checksums. Usually, this takes the form of a prologue, a body, and an epilogue. This transformation may be illustrated by comparing the checksum manipulation statements in Figs. 6 and 7.

6 Results

We now compare our approach of generating checks after a dataflow analysis has been done to determine information about available checksums at various points in the program with the approach of [12], which also uses the checksum based approach to detect errors but does not perform any dataflow analysis. Clearly, the approach of [12] would generate far more checks than our approach. For the jacobi solver example presented here, our approach would generate a check at the very end of the program, while the approach of [12] would generate checks in each iteration. These two versions of code were run on 16 nodes of an Intel Paragon distributed memory multicomputer. Fig. 44 shows the effect of varying the matrix size on the overhead of the two error-detecting versions of the algorithm. The number of iterations was kept constant at 1000. Overheads imposed by our approach diminishes with matrix size and is less than 5% for matrix sizes larger than 300×300, while the overheads imposed by the approach of [12] remains relatively constant at around 80%. Fig. 45 shows the effect of varying the number of iterations while keeping the matrix size constant at 100×100. The overheads imposed by both approaches increase with the number of iterations initially while appearing to stabilize for matrix sizes beyond 300×300. However, while the overheads imposed by our approach are only around 20%, the overheads imposed by the approach of [12] are much higher, around 80%.

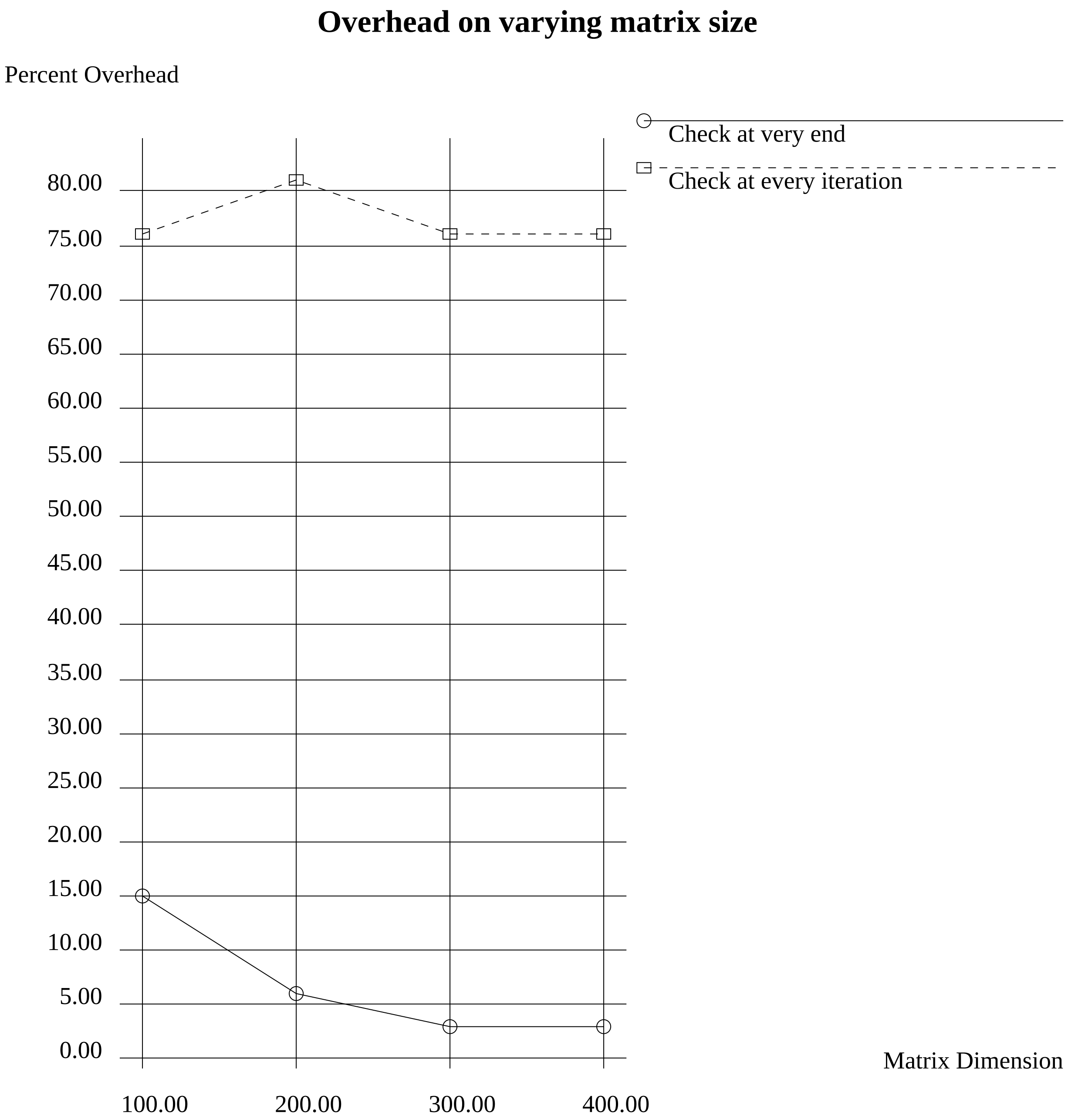

Figure 44: Overhead on varying matrix size

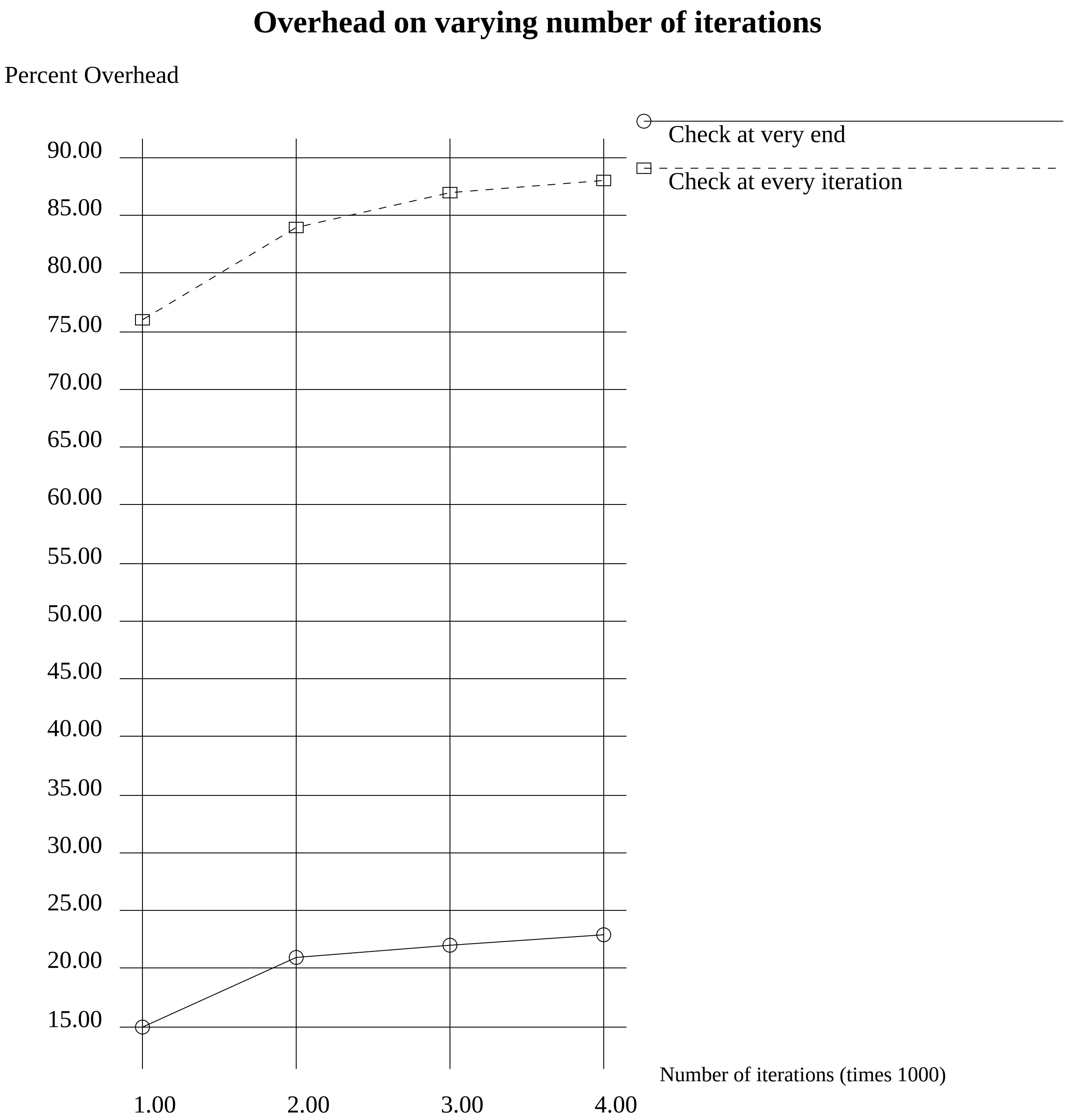

Figure 45: Overhead on varying number of iterations

7 Summary and Conclusions

In this report we have proposed a method for generating error-detecting parallel numerical programs at compile time. The input to the compiler is a serial Fortran program with HPF data distribution annotations. The compiler is used to identify portions of code which implement affine transformations by examining the syntactic structure of the statements. If in addition to possessing a suitable syntactic structure, a statement satisfies some additional conditions pertaining to the dependencies it is involved in, the statement may be checked by a checksum manipulation, the checksums being computed by summing over a selected dimension of the arrays being transformed by the statement. Typically, the actual manipulation of checksums involves far fewer operations than the original statement, leading to a cheaper check than duplication of the code. Portions of the code which do not implement affine transformations are checked by duplication.

Although the idea of using checksum manipulations to check selected portions of code is similar to [12], our approach is superior for many reasons. An important fact established by our work is that apart from the syntactic structure, one also needs to examine the dependencies a statement is involved in in order to determine whether it performs an affine transformation. We have established sufficient conditions for when a candidate checksum statement (one which possesses the necessary syntactic structure) is actually a valid checksum statement (one that can actually be checked using checksum manipulations). Ignoring the dependence conditions may lead to incorrect code in some cases. Another important improvement is the introduction of a dataflow analysis phase which inserts checks and recomputes checksums only where necessary, as opposed to the earlier approach, which would recompute checksums prior to and generate checks after each loop nest. Finally, we have actually implemented the entire system, as opposed to the earlier approach which was not implemented.

Although the lower overhead of our approach over the earlier approach is quite obvious, we have demonstrated this for an example code on an actual parallel computer. Although further work needs to be done in order to handle more complicated array subscripts and loop expressions as well as procedure calls, we feel we have demonstrated the potential of our approach to generate error-detecting parallel programs with much less overhead than simple duplication and comparison and with no extra effort on the part of the user.

References

[1] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, pp. 518–528, June 1984.

[2] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham, "Algorithm-based fault tolerance on a hypercube multiprocessor," *IEEE Trans. Comput.*, vol. 39, pp. 1132–1145, September 1990.

[3] Y.-H. Choi and M. Malek, "A fault-tolerant fft processor," *IEEE Trans. Comput.*, vol. 37, pp. 617–621, May 1988.

[4] V. Balasubramanian and P. Banerjee, "Tradeoffs in the design of efficient algorithm-based error detection schemes for hypercube multiprocessors," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 183–194, February 1990.

[5] V. Balasubramanian, "*The Analysis and Synthesis of Efficient Algorithm-Based Error Detection Schemes for Hypercube Multiprocessors.*" Ph.D. dissertation, Univ. of Illinois, Urbana-Champaign, February 1991. Tech. Report no. CRHC–91–6, UILU–ENG–91–2210.

[6] J. S. Plank, Y. Kim, and J. J. Dongarra, "Algorithm-based diskless checkpointing for fault-tolerant matrix operations," *Proc. FTCS-25*, June 1995.

[7] A. Roy-Chowdhury and P. Banerjee, "Algorithm-based fault location and recovery for matrix computations," *Proc. FTCS-24*, June 1994.

[8] J. G. Holm and P. Banerjee, "Low cost concurrent error detection in a VLIW architecture using replicated instructions," *Proc. ICPP-21*, August 1992.

[9] D. M. Blough and A. Nicolau, "Fault tolerance in super-scalar and VLIW processors," *Proc. IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, 1992.

[10] C. Gong, R. Melhem, and R. Gupta, "Compiler assisted fault detection for distributed-memory systems," *Proc. SHPCC*, May 1994.

[11] C. Gong, R. Melhem, and R. Gupta, “Replicating statement execution for fault detection on distributed memory multiprocessors,” *Proc. IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, June 1994.

[12] V. Balasubramanian and P. Banerjee, “Compiler-assisted synthesis of algorithm-based checking in multiprocessors,” *IEEE Trans. Comput.*, vol. 39, pp. 436–446, April 1990.

[13] P. Banerjee, V. Balasubramanian, and A. Roy-Chowdhury, *Foundations of Dependable Computing*, vol. III: System Implementation, ch. Compiler Assisted Synthesis of Algorithm-Based Checking in Multiprocessors, pp. 159–211. Kluwer Academic Publishers, 1994.

[14] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, and M. E. Zosel, *The High Performan Fortran Handbook.* Cambridge, MA: MIT Press, 1994.

[15] S. Hiranandani, K. Kennedy, and C. Tseng, “Compiling fortran d for mimd distributed memory machines,” *CACM*, vol. 35, August 1992.

[16] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten, “Parafrase–2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors,” *Proc. ICPP–18*, pp. II:39–48, August 1989.

[17] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, “The paradigm compiler for distributed-memory message passing multicomputers,” *Proc. First International Workshop on Parallel Processing, Bangalore, India*, December 1994.

[18] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, “Compiler optimizations for distributed-memory multicomputers used in the PARADIGM compiler,” *Proc. ICPP–23*, pp. II:1–10, August 1994.

[19] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Reading, MA: Addison Wesley, 1988.