Replicating Statement Execution for Fault Detection on Distributed Memory Multiprocessors

Chun Gong, Rami Melhem and Rajiv Gupta
Department of Computer Science
The University of Pittsburgh
Pittsburgh, PA 15260

Abstract

A compiler-assisted methodology is proposed for fault detection on distributed-memory systems. Selected instances of program statements are replicated in a way that ensures appropriate coverage. Replication strategies for the detection of permanent and transient faults are presented. These strategies use idle processor times for replicating statement execution, whenever possible. Two approaches are also discussed for implementing the proposed strategies on *single program, multiple data (SPMD)* parallel execution platforms. The first approach replicates program statements through source-to-source program transformations while the second approach achieves the replication of program statements indirectly by replicating data on multiple processors.

1 Introduction

Fault detection and tolerance can be achieved by introducing redundancy in a system at either the hardware level or the software level. Recently, it has been realized that the computing power of a multiprocessor system is rarely completely used. This has resulted in the development of techniques that achieve fault detection by replicating computations and comparing the results of these computations [7] [15]. Ideally by utilizing the spare capacity of the system to execute replicated computations, fault detection and tolerance can be achieved at a minimal cost.

In [1], the authors develop a compiler-assisted scheme to enable a process to quickly recover from transient faults, and in [4], the authors present a method that utilizes the VLIW compiler to schedule redundant operations on idle functional units for fault detection purposes. In [12], compiler techniques are used to insert checkpoints into a program so that both the desired checkpoint intervals and reproducible locations are maintained. In [3], the authors describe a source-to-source restructuring compiler for the synthesis of low-cost checks for numerical programs using the notion of algorithm-based checking.

Replication of computations has been carried out at various levels of granularity. In this paper, we consider replication at the statement level, and we describe a compiler-based strategy to achieve fault detection on distributed memory systems. This approach allows the compiler to utilize idle resources for fault detection purposes and it is applicable to both permanent and transient faults. It differs from the approach of [3] in that it presents a systematic approach to utilizing the spare processors' capacity for introducing redundancy in distributed memory environments. Our approach also provides a means for analyzing the resulting overhead.

When a shared name space is used for programming distributed memory machines, the compiler typically translates the shared memory program into one that executes under the *single-program, multiple-data* (SPMD) execution model [5]. Under SPMD, the same program is distributed to all processors, and each statement or statement instance in the program is either executed on all the processors or is executed on only one processor. A global statement is one that executes on all processors while a private statement is one that executes on only one processor or whose instances are distributed to different processors such that each instance is executed on one processor. For each instance of a private statement, a scheduling function identifies the processor that should execute the instance.

We propose to achieve fault detection at the statement level by duplicating the execution of statement instances on two processors and comparing the results of these executions. In order to specify this duplication, we generalize the scheduling function such that it may schedule an instance of a private statement on more than one processor. Whenever a duplicated statement is executed in the program, the processors executing that statement are tested. The degree of fault coverage is controlled by the degree of duplication.

In the following sections, we describe our approach for the detection of single faults in distributed memory multiprocessor systems. We assume that a processor is faulty if and only if it produces wrong results. We also assume that the faults are either permanent or transient and the interprocessor communication is fault free.

2 The Parallel Execution Model

We consider a system in which N processors are connected by some interconnection network. Each processor, which has a unique identifier from $0, 1, \cdots, N-1$, has its own local memory. There is no global memory, and interprocessor communication is carried out through message passing primitives.

A shared memory program is executed on such a system by providing a copy of the program to all the processors, specifying the distribution of data among the processors through a function α, and specifying the execution schedule through a function ϕ. The function $\alpha(d)$ provides the id of the processor at which a data item d resides. Typically, data arrays are distributed among the processors while scalars are replicated on all processors. Only statements inside loop bodies and array statements (as in Fortran 90) are considered private statements while other control flow statements are global statements that are executed by each processor. Because array statements can be easily translated into parallel loops, we will only consider the compilation of loops in this work. For each instance S^j of a private statement, S, the function $\phi(S^j)$ identifies the processor that will execute S^j. The following methods are commonly used to specify ϕ.

1. *The user specifies ϕ:* Languages such as Kali and the CRAY T3D Fortran [13] allow the specification of ϕ for scheduling of loop iterations on specific processors [11].

2. *The compiler uses ϕ to follow the "owner computes rule":* In compilers for languages such as the Fortran D and CM Fortran [6], ϕ is chosen such that each statement is executed by the processor that owns the variable whose value is being computed [10]. Thus, the ϕ function is defined at statement level rather than loop iteration level.

3. *The compiler automatically selects ϕ:* Heuristics are developed to select ϕ in conjunction with data distribution. The goal of the heuristic is to balance parallelism and communication so as to achieve good performance [2].

A computation $\mathcal{C}$ is considered to be *data parallel* for a given data distribution and scheduling function ϕ if it can be executed in parallel by the processors in three distinct steps: *operand communication*, *computation*, and *result communication*. During the *operand communication* step processors communicate with each other to exchange data values that although required by one processor during the execution of $\mathcal{C}$, reside at another processor. In the *computation* step, each processor executes the portion of computation $\mathcal{C}$ that is assigned to it. In the final step of *result communication*, processors communicate with each other to send the values of variables computed during $\mathcal{C}$ to the processors where they must be stored. It should be noted that in each of the above steps all processors execute in parallel and hence the parallelism in both the communication and computation is exploited. Furthermore, by grouping communication, the overhead of message passing is reduced and by ensuring that no communication is required during the computation phase the execution is allowed to proceed unhindered.

The compiler must identify data parallel computations and then generate code according to the scheme described above. It should be the goal of the compiler to find coarse-grain data parallel computation to improve the efficiency with which parallelism is exploited.

3 Statement Duplication in Regular Loops

Duplicating and comparing the results of executing statement instances is the basis for the proposed approach to fault detection in distributed-memory multiprocessors running shared data programs. In order to implement this statement duplication approach, two non-trivial issues need to be addressed. Namely, the *selection* of the statement instances to be duplicated and the scheme for *scheduling* duplicated executions and *comparing* results.

We will concentrate on duplicating statement instances within loop executions. We focus on loops because most of the execution time of programs is spent on executing loops. The statement instances that result from loop execution will be denoted by $\sigma^0 \ldots \sigma^{m-1}$ and $\phi(i)$ will denote the processor on which σ^i is scheduled for execution. To simplify the discussion we assume that m is an integral multiple of the number of processors N. It should be noted that this notation does not restrict the type of loops that generate the m instances or the granularity of each instance. The instances $\sigma^0 \ldots \sigma^{m-1}$ are ordered such that $\sigma^i \rightarrow \sigma^{i+1}$, where $\rightarrow$ defines the lexicographical ordering of the statement instances in the source program. That is, during serial execution, σ^i executes before σ^{i+1}.

The duplication of the execution of σ^i is specified by a function ϕ_r such that $\phi_r(i)$ is the set of processors on which σ^i is to execute. In general, if $\mid \phi_r(i) \mid = 2$, then an error in the execution of σ^i resulting from a fault in any of the two processors in $\phi_r(i)$ can be detected. The choice of ϕ_r is influenced by the required fault coverage. For example, if errors resulting from transient faults are to be detected, then every statement instance should be duplicated. On the other hand if we are interested in detecting permanent faults, then it may suffice to replicate selected statement instances. Specifically, if all the instances in the loop are of the same type, that is, their executions involve the same type of functional units, then error free execution of the entire loop may be guaranteed if the last instance executed on each processor is duplicated. If instances in the loop are of different types, then the above condition should be satisfied for each statement type.

The duplication mechanism should not affect the correctness of the calculated results. In general, the mapping function, ϕ, the program structure and the data dependence determine a timing function Δ which satisfies a *scheduling dependence condition* and a *data*

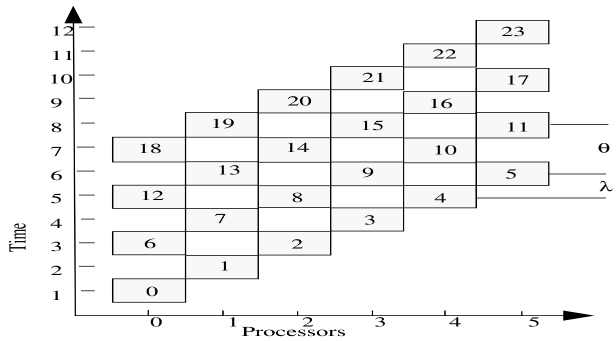

Figure 1: The General Pattern of Parallel Loop Execution: θ=2 and λ=1.

dependence condition. Specific restrictions to guarantee that these two conditions are satisfied after duplication depend on the implementation of the duplication strategies. For non-duplicated programs, however, the *scheduling dependence condition* states that if σ^i and σ^j are scheduled on the same processor, and $i > j$, then $\Delta(i) > \Delta(j)$, where $\Delta(i)$ and $\Delta(j)$ are the times at which σ^i and σ^j execute, respectively. The *data dependence condition* states that if σ^i uses data computed in σ^j, then $\Delta(i) > \Delta(j)$.

In order to analyze the duplication schemes, we assume that instances are of the same type and require the same execution time (a single time unit). With this assumption, the execution pattern of a loop, which is determined by the functions ϕ and Δ may be represented graphically by plotting instances with respect to two axes, a horizontal processor axis and a vertical time axis. In this paper, we limit our discussion to loops with execution patterns similar to the one shown in Figure 1. In this figure, 24 instances execute on 6 processors. Each shaded square indicates a time slot during which an instance is being executed. The pattern may be described by the following general forms of ϕ and Δ:

$$\phi(i) \quad = \quad \pi(i) \; mod \; N \tag{1}$$

$$\Delta(i) \quad = \quad \lambda(\pi(i) \; mod \; N) + \theta \lfloor \frac{\pi(i)}{N} \rfloor \tag{2}$$

where λ is a parameter that defines the execution skew among the processors, θ is a parameter that defines the idle time between the execution of successive instances on each processor, and π is a permutation function that maps instances to time slots. We use this function to allow the separation of the execution characteristics of the pattern (specified by θ and λ) from the actual assignment of instances to time slots. Two particular functions π are of interest. Namely, $\pi_h(i) = i$ and $\pi_v(i) = (i \; mod \; u) \; N + \lfloor \frac{i}{u} \rfloor$, where $u = \frac{m}{N}$.

When π_h is used in (1), the instance σ^i is mapped to processor $i \; mod \; N$, while when π_v is used in (1), the instance σ^i is mapped to processor $\lfloor \frac{i}{u} \rfloor$. For example, Figure 2a is a special case of the pattern (1)/(2) with $\lambda = 1$, $\theta = 1$ and $\pi = \pi_v$. The fully parallel pattern of Figure 2b is also a special case of (1)/(2) with $\lambda = 0$, $\theta = 1$ and $\pi = \pi_h$.

The fault detection technique proposed in this work relies on the duplication of instance execution. The replica of an instance σ^i is called a secondary instance and is denoted by σ^i_s. To avoid confusion, we will refer to the original instance of σ^i as the primary instance, σ^i_p. In order to exploit idle slots, the compiler appropriately selects the instances to be duplicated as well as the processors where secondary instances are to execute. Let $\phi_s(i)$ be

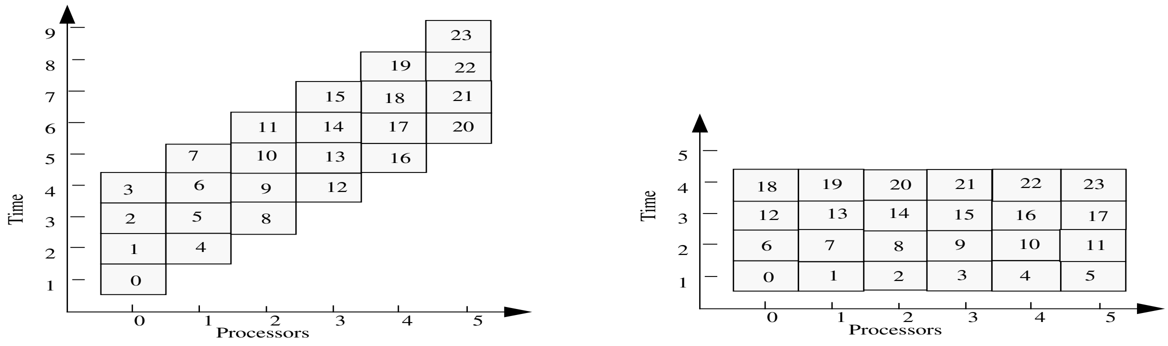

Figure 2: Execution Patterns with Different π Functions: (a) $\theta = 1$, $\lambda = 1$, $\pi = \pi_v$; and (b) $\theta = 1$, $\lambda = 0$, $\pi = \pi_h$.

the processor on which σ_s^i is to execute. Thus, if σ^i is duplicated, its mapping function can be expressed by a multi-valued function $\phi_r(i) = \{\phi_p(i), \phi_s(i)\}$, where $\phi_p(i) = \phi(i)$. For a non-duplicated statement, only primary instances exist and thus, $\phi_r(i) = \{\phi_p(i)\}$.

As is the case in the absence of duplication, the mapping function, the program structure and the data dependences determine the timing function for the duplicated copies. For an instance, σ^i, this function is now a set $\Delta_r(i) = \{\Delta_p(i), \Delta_s(i)\}$, where $\Delta_p(i)$ is the time at which σ_p^i executes and $\Delta_s(i)$ is the time at which σ_s^i executes. Note that $\Delta_p(i)$ may or may not be equal to $\Delta(i)$. Moreover, if σ^i is not duplicated, then $\Delta_r(i) = \{\Delta_p(i)\}$.

The timing function for a given duplicated mapping, ϕ_r, should satisfy the schedule dependence and the data dependence conditions. The schedule dependence condition requires that if two instances (primary or secondary) execute on the same processor, then the order of their execution should follow the lexicographical ordering of the program. More formally, for any σ_y^i and σ_z^j, where y and z are either p or s, the following should be satisfied:

$$\sigma_y^i \rightarrow \sigma_z^j \ and \ \phi_y(i) = \phi_z(j), \ implies \ \Delta_y(i) < \Delta_z(j). \tag{3}$$

The data dependence condition is determined by the semantics of the duplication. Two different semantics are possible, the *wait-for-all* and the *wait-for-one* semantics. Specifically, if before duplication, an instance, σ^i depends on data calculated by another instance σ^j, then after duplication, the *wait-for-all* semantics requires that copies of σ^i do not execute before all copies of σ^j execute. This condition can be expressed as $min\{\Delta_p(i), \Delta_s(i)\} > max\{\Delta_p(j), \Delta_s(j)\}$, where we abused the notation by assuming that $max\{\Delta_p(j), \Delta_s(j)\} = min\{\Delta_p(j), \Delta_s(j)\} = \Delta_p(j)$ if σ^j is not duplicated.

The wait-for-all semantics allows the data computed by duplicated instances to be compared before it is used, thus preventing error propagation, at the cost of possible execution delay. In the *wait-for-one* semantics, data comparison does not delay execution at the cost of possible error detection delay and possible error propagation. In this case, copies of σ^i can execute as soon as σ_p^j finishes execution. The results of σ_p^j and σ_s^j are compared whenever σ_s^j finishes execution. In this case, the data dependence condition is $min\{\Delta_p(i), \Delta_s(i)\} > \Delta_p(j)$. Moreover, if the duplication does not change the time at which the primary instances execute, that is if $\Delta_p = \Delta$, and assuming that Δ satisfies the data dependence condition, then for Δ_r to satisfy the data dependence condition it is sufficient to have

$$\Delta_s(i) > \Delta_p(i) \ \ for \ any \ i. \tag{4}$$

By carefully chosing the function, ϕ_r, it is possible to control the timing function Δ_r and thus control the overhead in the duplicated execution pattern. In the remainder of this section, we give some examples for adding duplicate instances to the general loop execution pattern given by (1)/(2). The resulting duplicated patterns satisfy (3) and (4) and thus represent correct SPMD executions.

Assuming an even number of processors, a possible duplication for the pattern (1)/(2) is given below.

$$\phi_r(i) = \begin{cases} \{\phi(i), \phi(i) - 1\} & \text{if } \pi(i) \geq m - N \text{ and } \phi(i) \text{ is odd} \\ \{\phi(i)\} & \text{otherwise} \end{cases} \tag{5}$$

In other words, ϕ_r is such that the last instance that executes on an odd processor P is duplicated on processor $P - 1$. Clearly, this strategy detects any permanent fault that causes an error in the execution of the loop. Execution patterns corresponding to this strategy are shown in Figure 3 for the two patterns of Figure 2. The darkly shaded squares represent duplicated computations. As indicated by the figure, the strategy does not utilize any additional time if $\lambda \neq 0$ since the duplicated statements are executed on idle processors. If, $\lambda = 0$, then the execution increases by only one time unit.

In order to obtain complete coverage for errors resulting from transient faults, it is necessary to duplicate the execution of all the instances in the loop. Different strategies may be used to accomplish this goal. For instance, if N is even, then it is possible to duplicate the instances executing on a processor, P, on either processor $P - 1$ or $P + 1$. Specifically, the duplication strategy may be defined as follows.

$$\phi_r(i) = \begin{cases} \{\phi(i), \phi(i) - 1\} & \text{if } \phi(i) \text{ is odd} \\ \{\phi(i), \phi(i) + 1\} & \text{if } \phi(i) \text{ is even} \end{cases} \tag{6}$$

For patterns with $\theta > 1$, the duplicated computations are interleaved with the original computations as shown in Figure 4a. In this case, no duplication overhead is associated with the strategy. If $\theta = 1$, however, the duplication penalty is 100% (see Figure 4b). The 100% overhead in the case $\theta = 1$ and $\lambda = 0$ cannot be avoided since the loop is fully parallel and no processor is idle during the execution of the non-duplicated loop. A similar argument applies if $\theta = 1$, $\lambda > 0$ and the idle time resulting from the skewed execution is much smaller than the total execution time of the loop. This is the case when $\lambda(N - 1)$ is much smaller than $\frac{m}{N}$. However, if $\lambda(N - 1)$ is large compared to $\frac{m}{N}$, then an attempt should be made to use the idle time for duplicate computations. One technique that may be used if $\lambda(N - 1) \geq 2\frac{m}{N}$ is to duplicate the instances executing on a processor, P, on either processor $P - \frac{N}{2}$ or $P + \frac{N}{2}$ depending on whether P is larger or smaller than $\frac{N}{2}$, respectively. A duplication pattern corresponding to this strategy is presented pictorially in Figure 5. In this case, the addition of the fault detection capability does not cause any duplication overhead.

4 Implementation of the Fault Detection Strategies

In this section, we briefly describe two approaches for implementing fault detection strategies. The first approach relies on automatic program transformations to duplicate statements and introduce checks in a program. The transformed program can be compiled by

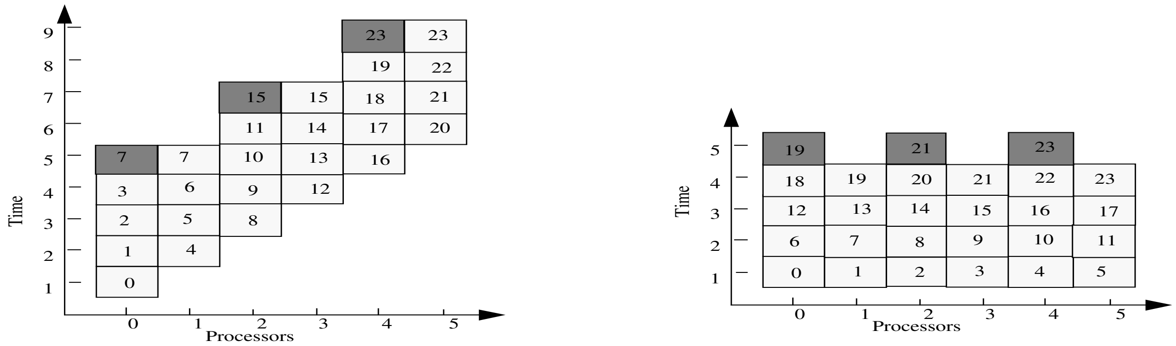

Figure 3: Duplication for Permanent Fault Detection of Patterns in Figure 2.

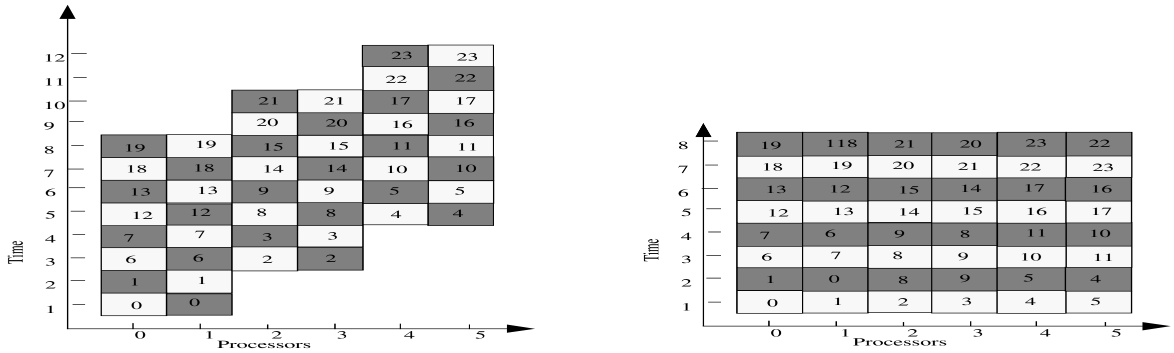

Figure 4: Duplication for Transient Fault Detection (a) $\theta = 2$, $\lambda = 1$, $\pi = \pi_h$; and (b) $\theta = 1$, $\lambda = 0$, $\pi = \pi_h$.

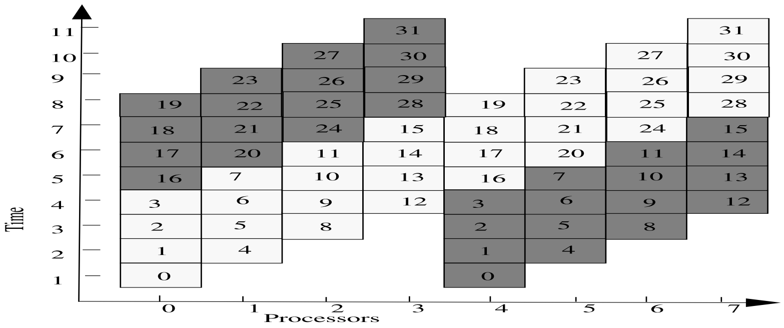

Figure 5: Efficient Duplication for Transient Fault Detection with $\lambda = \theta = 1$, $\pi = \pi_v$, $\lambda(N-1) \geq 2\frac{m}{N}$.

an existing SPMD code generator. The second approach achieves duplication of statements indirectly through the duplication of data in compilers based on the "owner computes rule".

4.1 Statement Duplication through Program Transformations

Duplication strategies may be automatically implemented by the compiler through program transformations. The transformations must perform two tasks. First, they must duplicate appropriate statement instances and schedule their execution on appropriate processors. Second, they should introduce code for performing the checks. Specifically, if a statement $A = f(d_1, \ldots, d_n)$ is selected for duplication, then the compiler inserts another statement, $Temp = f(d_1, \ldots, d_n)$, where $Temp$ is an auxiliary variable not used elsewhere in the program. The ϕ function for the duplicated statement must also be specified explicitly in the code and the variable $Temp$ will reside in the memory of the processor that executes the duplicated statement. In addition, we introduce the execution of statement $Check(A, Temp)$ in the two processors that now execute the same statement. Thus, the two processors that execute statements $A = f(d_1, \ldots, d_n)$ and $Temp = f(d_1, \ldots, d_n)$ are able to check each other.

We clarify this technique by an example. Consider the regular loop pattern that was expressed in terms of parameters λ and θ in the preceding section. The form of the loop in which regular execution pattern is readily observable is:

For j = 0 **To** $n-1$ **Do**
    For i = 0 **To** $N-1$ **On** P_i **Do**
        A(i,j) = $\mathcal{F}$(A(i-λ,j), A(i+λ,j-θ+λ))

To detect permanent faults using the strategy illustrated in Figure 3, the last instance of the statement executed by a processor is replicated on a neighboring processor by adding the following code to the above loop:

For j = 0 **To** $n-1$ **Do**
    For i = 0 **To** $N-1$ **On** P_i **Do**
        A(i,j) = $\mathcal{F}$(A(i-λ,j), A(i+λ,j-θ+λ))
    – the following statement performs permanent fault detection
    If j = $n-1$ **Then**
      For i = 0 **To** $N-1$ **On** P_i **Do**
          If i mod 2 = 0 **Then**
            Temp(i) = $\mathcal{F}$(A(i-λ+1,j), A(i+λ+1,j-θ+λ)); Check(Temp(i) = A(i+1,j))
          Else Check(Temp(i-1) = A(i,j))

The transformed loop enables the detection of permanent faults in processors P_1 to P_N. Similar transformations may be applied to implement the transient fault detection strategy of Figure 4.

4.2 Duplication through Owner-computes Implementation

The owner computes rule is a powerful approach to compile SPMD programs for distributed-memory systems [5] [11] [14]. This approach uses the data distribution, α, to specify the scheduling function ϕ. Specifically, an assignment statement, S, of the form $A =$

$f(d_1, \ldots, d_n)$ is scheduled to execute on the processor on which A is stored (called the owner of A). That is $\phi(S) = \alpha(A)$. With this owner computes rule, the compiler generates code for a computation, $\mathcal{C}$, such that each processor examines $\mathcal{C}$ and takes one of the following actions: (a) performs the operation indicated by C if it owns the data element whose value is being computed. Non-local operands are received from other processors; (b) sends a local data to the processor that needs the data to execute C; or otherwise (c) skips C. The communication primitives introduced by the compiler are non-blocking *send* and blocking *receive*. Compiler optimization techniques are applied to maximize the granularity of $\mathcal{C}$.

A natural way of duplicating statement execution when the owner computes rule is applied is to duplicate data. Specifically, following the SPMD execution model, if A is owned by more than one processor, then the statement $A = f(d_1, \ldots, d_n)$ will be automatically executed by all the owners of A. This form of data duplication requires that the operands of f be sent to both owners of A.

In order to express data duplication, we generalize the owner function α such that $\alpha_r(A)$ is the set of processors that owns A. Hence, for a statement, S of the form $A = f(d_1, \ldots, d_n)$, the mapping function is given by $\phi_r(S) = \alpha_r(A)$. Detailed implementations of this duplication technique are given in [8]. These implementations are obtained by modifying the communicator modules that exchange data between owners and consumers during SPMD execution. Replicated execution of computation instance σ^i in a loop is achieved through data ownership replication. The choice of ϕ_r (and thus, α_r) should be guided by the fault coverage required. Note, however, that duplicating data ownership duplicates execution without a need for changing the syntax of the program or adding any statement to it. In fact, σ_p^i and σ_s^i result from the execution of the same program statement by two processors. Hence, lexicographically, σ_p^i and σ_s^i are equivalent in the sense that $\sigma_s^i \rightarrow \sigma^j$ if and only if $\sigma_p^i \rightarrow \sigma^j$. In this context, the scheduling dependence condition (3) is rewritten as

$$\phi_y(i) = \phi_z(j) \text{ and } i < j \text{ implies } \Delta_y(i) < \Delta_z(j). \tag{7}$$

The data dependence condition remains as specified by (4).

It is straight forward to prove that for the pattern (1)/(2), the duplication strategy (5) always satisfies (7) (Figure 3). The duplication strategy (6) also satisfies (7) if $\theta > 1$ (Figure 4a). For $\theta = 1$, however, the strategy (6) satisfies (7) only if $\pi = \pi_h$ (Figure 4b). Finally, the duplication strategy shown in Figure 5 satisfies (7) only if $\pi = \pi_v$. Note that program transformation techniques may be used to obtain either π_v or π_h in the loop pattern (1)/(2).

5 Conclusion

We have presented a methodology for fault detection which relies on duplicating the execution of statement instances. This methodology is *general* and can be adapted to match the requirements of a given fault model. It does not require specialized hardware and is implemented entirely in software through a compiler that translates shared memory programs for execution on a distributed memory machine. The methodology is *efficient* since analysis techniques can be developed to enable the merging of duplicated computation with the original computation in a manner that reduces the increase in overall execution time.

We have analyzed, in some details, the application of the methodology to a general class of parallel loops and we have discussed two different methods for duplicating computation instances. The details of implementing the two methods are presented in [8] and [9], respectively, along with experimental results showing that the duplication overhead is, in general, relatively low. We are currently studying efficient replication strategies for program constructs that are more general than the loops considered in this paper.

References

[1] N. Alewine, S. Chen, C. Li, W. Fuchs, and W. Hwu. Branch recovery with compiler-assisted multiple instruction retry. In *The 22nd Annual International Symposium on Fault-Tolerant Computing*, 1992.

[2] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN Conf. on Programming Language Design and Implementation*, 1993.

[3] V. Balasubramanian and P. Banerjee. Compiler-assisted synthesis of algorithm-based checking in multiprocessors. *IEEE Trans. on Computers*, 39, 4 1990.

[4] D. Blough and A. Nicolau. Fault tolerance in super-scalar and vliw processors. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems.* IEEE, 1992.

[5] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[6] Cm fortran user's guide for the cm-5. *Thinking Machines*, 1992.

[7] A. Dahbura, K. Sabnani, and W. Hery. Spare capacity as a means of fault detection and diagnosis in multiprocessor systems. *IEEE Trans. on Computers*, 38, 6 1989.

[8] C. Gong, R. Melhem, and R. Gupta. Compiler assisted fault detection for distributed-memory systems. *Proc. of SHPCC, Scalable High Performance Computing Conference*, 1994.

[9] C. Gong, R. Melhem, and R. Gupta. Loop transformation for fault detection on massively parallel systems. *Technical Report 94-39, Department of Computer Science, The University of Pittsburgh*, 1994.

[10] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the fortran d programming system. In *Proc. of the Fourth Workshop on Languages and Compilers for Parallel Computing*, 1991.

[11] C. Koelbel, P. Mehrotra, and J. Rosendale. Supporting shared data structure on distributed memory architectures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 177–186. SIGPLAN, ACM, 1990.

[12] J. Long, W. Fuchs, and J. Abraham. Compiler-assisted static checkpoint insertion. In *The 22nd Annual International Symposium on Fault-Tolerant Computing*, 1992.

[13] T. MacDonald, D. Pase, and A. Meltzer. Addressing in cray research's mpp fortran. *Third Workshop on Compilers for parallel Computers*, 1992.

[14] M. Quinn and P. Hatcher. Compiling simd programs for mimd architecthres. In *Proceedings of International Conference on Computer Languages*, pages 291–296, 1990.

[15] S. Tridandapani and A. Somani. Efficient utilization of spare capacity for fault detection and location in multiprocessor systems. In *International Symposium on Fault-Tolerant Computing.* IEEE, 1992.