Loop Transformations for Fault Detection in Regular Loops on Massively Parallel Systems

Chun Gong, Rami Melhem, *Member, IEEE Computer Society*,
and Rajiv Gupta, *Member, IEEE Computer Society*

Abstract—Distributed-memory systems can incorporate thousands of processors at a reasonable cost. However, with an increasing number of processors in a system, fault detection and fault tolerance become critical issues. By replicating the computation on more than one processor and comparing the results produced by these processors, errors can be detected. During the execution of a program, due to data dependencies, typically not all of the processors in a multiprocessor system are busy at all times. Therefore processor schedules contain *idle time slots* and it is the goal of this work to exploit these idle time slots to schedule duplicated computation for the purpose of fault detection. We propose a compiler-assisted approach to fault detection in regular loops on distributed-memory systems. This approach achieves fault detection by duplicating the execution of statement instances. After carefully analyzing the data dependencies of a regular loop, selected instances of loop statements are duplicated in a way that ensures the desired fault coverage. We first present duplication strategies for fault detection and show that these strategies use idle processor times for executing replicated statements, whenever possible. Next, we present loop transformations to implement these fault-detection strategies. Also, a general framework for selecting appropriate loop transformations is developed. Experimental results performed on the CRAY-T3D show that the overhead of adding the fault detection capability is usually less than 25%, and is less than 10% when communication overhead is reduced by grouping messages.

Index Terms— Compiler-assisted approach, data dependence analysis, distributed-memory systems, duplicating execution, execution pattern, fault detection, loop transformation.

1 Introduction

Massively parallel systems can incorporate thousands of processors at a reasonable cost. In order to achieve scalability in such systems, the memory is physically distributed among the processors. Due to the large number of processors, fault detection and fault tolerance are critical issues for such systems. Thus, techniques for fault detection and fault tolerance on distributed-memory systems is an important area of research.

Fault detection and tolerance can be achieved by introducing redundancy in a system at either the hardware level [6], [8], [27] or the software level [3,] [4], [14], [28]. Recently, it has been realized that the computing power of a multiprocessor system is rarely completely utilized. This has resulted in the development of techniques that achieve fault detection by replicating computations and comparing the results of these computations [16], [29]. Ideally by utilizing the spare capacity of the system to execute replicated computations, fault detection and tolerance can be achieved at a reduced cost.

Techniques for compiler-assisted fault detection and recovery have been developed in previous research. A compiler-assisted scheme to enable a process to quickly recover from transient faults is developed in [1] and a method that utilizes the VLIW compiler to insert redundant operations into idle functional units for fault detection purposes is presented in [7]. Compiler techniques are used in [24] to insert checkpoints into a program so that both the desired checkpoint intervals and reproducible locations are maintained. A source-to-source restructuring compiler for the synthesis of low-cost checks for scientific programs using the notion of algorithm-based checking is described in [5]. The compiler-assisted approaches are appealing since the compilers can apply a variety of analysis techniques to efficiently allocate resources in multiprocessor systems. Furthermore, adding fault detection capabilities to massively parallel systems is usually tedious and error prone. Thus, systematic techniques that can be implemented through compilers are most appropriate.

In this paper, we describe a compiler-assisted approach for achieving fault detection in regular loops on distributed-memory systems by duplicating computations at the statement level. This approach allows the compiler to utilize idle resources for the purpose of fault detection. It differs from the approach of [5] in that it presents a systematic approach to utilizing the spare processors' capacity for introducing redundancy in distributed memory environments. It also provides a means for analyzing the resulting overhead. Fault detection is achieved by duplicating the execution of statement instances on two processors and comparing the results of these executions. A detailed comparison between these two approaches will be given in Section 8. In the approach proposed in this paper, compile-time transformations are applied to duplicate statement instances and introduce checks to compare the computed results. Thus, 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.

- *C. Gong is with the Massachusetts Language Lab., Hewlett-Packard, 300 Apollo Drive, Chelmsford, MA 01824 Email: gong@apollo.hp.com.*
- *R. Melhem and R. Gupta are with the University of Pittsburgh, Pittsburgh, PA 15260.E-mail:{melhem, gupta}@cs.pitt.edu.*

Manuscript received May 24, 1994; revised Apr. 4, 1996.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95226.

The paper is organized as follows. Section 2 provides some notation and the general execution model and Section 3 introduces the concept of *execution patterns*. In Section 4, we present the principles of duplicating execution and introduce several duplicating strategies for regular loops. Implementations of the duplicating strategies are given in Section 5. In Section 6, we describe a general framework which, given a regular loop, selects appropriate duplicating strategy for fault detection. In Section 7, experimental results are given and in Section 8, some related work is discussed. Finally our concluding remarks are given in Section 9.

2 The Parallel Execution Model

We consider a system in which N processors are connected by an 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 information exchange between processors is achieved through message exchange.

There are two basic paradigms for programming distributed-memory machines: the explicit message passing model, and the shared-name-space model. Although the first model is more flexible, it is hard to program in that model. Therefore, many languages have been developed which provide the users with the conceptually easier programming model based upon a globally shared name space. The compiler is responsible for inserting necessary communication primitives in this model [9]. Examples of such languages include CM Fortran [13], C* [25], Vienna Fortran [10], and HPF [21]. The compiler-assisted methodology introduced in this paper achieves fault detection on distributed-memory systems when these systems are programmed using the shared-name-space model.

Under the shared-name-space model, a *scheduling function*, ϕ, is specified such that for each statement instance s_j, $\phi(s_j)$ 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 allow the specification of ϕ for scheduling of loop iterations on specific processors [23].
2) *The compiler chooses* ϕ *according to the "owner computes rule"*: In compilers for languages such as Fortran D and HPF, the user specifies data distribution and ϕ is chosen such that each statement is executed by the processor that owns the variable whose value is being computed.
3) *The compiler automatically selects* ϕ : Heuristics are employed for the selection of ϕ in conjunction with data distribution. The goal of the heuristic is to balance parallelism and communication in order to achieve good performance [2].

In this paper, we assume that ϕ is given by the user. Furthermore, the ϕ function specifies scheduling of loop iterations rather than individual statements. However, the techniques presented here can also be applied to the other methods described above [19].

We focus on loops since most of the idle time can be found in executing loops. We first define the notation used to describe data dependencies. There are three kinds of data dependencies. A statement instance s_i is said to be *flow dependent* upon statement instance s_j, denoted as $s_j \delta s_i$, if s_i reads from a memory location after s_j writes to the same memory location. A statement instance s_i is said to be *output dependent* upon statement instance s_j, if both s_i and s_j write into the same memory location. A statement instance s_i is *antidependent* upon s_j, if s_i writes into a memory location after s_j reads from the same memory location. Since output and antidependencies can be eliminated through renaming [15], we assume that the loop contains only flow dependencies. If there are instances of statements $s_1, s_2, \ldots, s_n$ such that $s_1 \delta s_2 \delta \ldots \delta s_n$, then we say that there is a dependence path of length $(n-1)$ from s_1 to s_n. There may be dependencies among different iterations of a loop. If a statement instance at the ith iteration of a loop is dependent upon a statement instance at the kth iteration, then this particular dependence is said to have a distance of $(i-k)$ [30]. For multiply-nested loops, a vector of dependence distances is used to represent the dependencies. If all the dependencies of a loop can be described by a set of dependence distances that are independent of the loop variables, then the loop is said to be a *regular loop*. In this paper, we consider regular loops which are perfectly nested and contain no branches. In the next section, we will give concrete examples of regular loops and describe their characteristics.

3 Execution Patterns for Perfectly Nested Loops

Given a loop and the function ϕ, by analyzing the data dependencies of the loop, a legal schedule that maximizes the parallelism of the loop can be determined. Here, the term "legal schedule" refers to a schedule that satisfies all the data dependencies of the loop. The *execution pattern* of a loop, under a legal schedule, specifies the time and the processor at which each loop instance is executed. This pattern can be identified by analyzing the data dependencies in the loop. We first consider the execution patterns of doubly-nested loops of the forms shown below and later generalize this concept for multiply-nested loops. In this loop, "On P_j" indicates that jth iteration of the inner loop is executed on processor P_j.

```
For i = 0 To M - 1 Do
  For j = 0 To N - 1 On P_j Do
      A(i, j) = F(A(i, j - λ),
      A(i - θ + λ, j + λ))
  EndFor
EndFor
```

Depending on the value of λ and θ, there might be data dependencies in the above loop that can prevent all processors from executing their assigned statement instances in a fully parallel fashion. In this paper, we assume that the execution of each statement instance takes the same amount of time (say, unit time). Fig. 1a shows a legal execution schedule for the above loop with $M=4$, $N=6$, $\lambda=1$, and $\theta=2$. At time 1, processor P_0 starts the execution of instance with $i=j=0$, then, at time 2, processor P_1 starts the execution of instance with $i=0$ and $j=1$, and so on, thus guaranteeing that data dependencies are satisfied. We refer to the above loop with

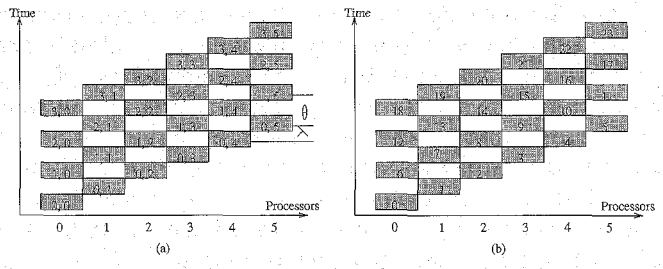


Fig. 1. (a) The Execution Pattern of the loop: $\theta = 2$ and $\lambda = 1$; (b) With the numbering function β_h.

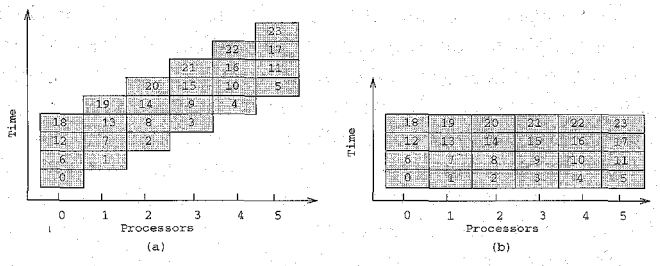


Fig. 2. Special cases of the general execution pattern: (a) skewed pattern with $\theta = 1$, $\lambda = 1$; and (b) parallel pattern with $\theta = 1$, $\lambda = 0$.

$\lambda \geq 1$ and $\theta \geq 2$ as the *general loop* and the execution pattern it generates the *general loop pattern*.

In Fig. 1a, each shaded square indicates a time slot during which a loop instance is being executed. In this example, a loop instance is a simple statement. In general, however, a loop instance can be composed of several statements or several loop iterations of a nested loop. The only requirement for a loop instance is that it must be executed in its entirety by one processor in a continuous time slot.

In order to describe an execution pattern, we introduce the following notation. A *numbering* of a loop is a one-to-one mapping from the set of loop instances to the set of natural numbers plus the symbol β which denotes an undefined value. In the following definition, we use $\sigma(i, j)$ to denote the instance with i and j as values of the outer and inner loop indices, respectively.

$$\beta_h(\sigma(i,j)) = \begin{cases} i \times N + j & 0 \leq i < M \wedge 0 \leq j < N \\ \perp & otherwise \end{cases}$$

This function maps a two-dimensional iteration space into a linear space. A numbering function facilitates the expression of execution patterns which are used to develop replication strategies. The execution pattern of Fig. 1a with the numbering function β_h is shown in Fig. 1b.

Given a numbering function β, we use σ^u to refer to the instance whose number under β is u, and $\phi(u)$ to denote the processor that will execute instance σ^u. For any given regular loop with a function ϕ, the compiler can determine a legal schedule which maximizes the parallelism of the execution of the loop. The timing function Δ describes the mapping between instances and time slots under the schedule. Thus, $\Delta(u)$ is the earliest time at which σ^u can be executed. Formally, the execution pattern of a loop is a pair of functions, $<\phi, \Delta>$. The general loop pattern may be described by the following general forms of ϕ and Δ:

$$\phi(u) = u \bmod N; \qquad \Delta(u) = \lambda(u \bmod N) + \theta\left\lfloor \frac{u}{N} \right\rfloor \qquad (1)$$

where λ is the execution skew among the processors and θ is the idle time between the execution of successive instances on each processor. For example, the *skewed loop pattern* of Fig. 2a is a special case of (1) with $\lambda = 1$ and $\theta = 1$ and the *fully parallel pattern* of Fig. 2b is a special case with $\lambda = 0$ and $\theta = 1$. The timing function Δ corresponding to a legal schedule satisfies the *data dependencies*, that is,

$$\sigma^v \delta \sigma^u \Rightarrow \Delta(u) > \Delta(v).$$

We obtained the execution pattern of Fig. 1 by assigning iterations of the inner loop to different processors. However, the general pattern given by (1) also represents the case where the processors are distributed along the outer loop, as shown in the following example.

For i = 0 **To** N - 1 **On** P_i **Do**
  For j = 0 **To** M - 1 Do
    $A(i, j) = F(A(i - 1, j))$
  EndFor
EndFor

This is a special case of (1) with $\lambda = 1$, $\theta = 1$. The numbering function for this case is defined as follows.

$$\beta_v(\sigma(i,j)) = \begin{cases} j \times M + i, & 0 \leq i < M \wedge 0 \leq j < N \\ \perp & otherwise \end{cases}$$

Fig. 2a shows a legal execution pattern under this numbering function. In the remainder of the paper, we consider the case in which the processors are assigned along the inner loop, since the other case can be handled through the appropriate numbering function.

Other types of loops we will consider in this paper are the *triangular loop* and the *bisection loop*. These two types of loops can be uniformly illustrated by the following loop form, where $f(i) = i$ or $f(i) = N - \frac{N}{2^i}$.

For i = 0 **To** M - 1 **Do**
  For $j = f(i)$ **To** N - 1 **On** P_j **Do**
    $A(i, j) = F(A(i - 1, j - 1))$
  EndFor
EndFor

A legal execution pattern for the case where $f(i) = i$ is shown in Fig. 3a and a legal execution pattern for the case where $f(i) = N - \frac{N}{2^i}$ is shown in Fig. 3b. For these types of loops, we can use the following numbering function to convert instances from the two-dimension space into a one-dimension space.

$$\beta_t(\sigma(i,j)) = \begin{cases} i \times N + j & 0 \leq i < M \wedge f(i) \leq j < N \\ \perp & otherwise \end{cases}$$

In the discussion so far, the number of processors is as-

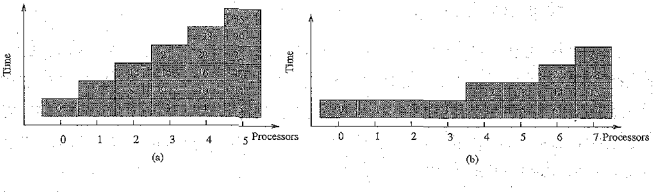


Fig. 3. (a) The triangular loop execution pattern; and (b) the bisection execution pattern.

sumed to be the same as the number of iterations of the loop. However, in practice the number of iterations is usually much larger than the number of processors. In order to generalize the concept of execution pattern, we assume that, in the following loop, the number of loop iterations (U) is greater than the number of processors (N). In this loop, $\sigma(i, j)$ is either a sequence of assignment statements or a nested loop.

For i = 0 **To** M - 1 **Do**
  For j = 0 **To** U - 1 **On** $P_{d(j)}$ **Do**
    $\sigma((i, j)$
  EndFor
EndFor

We use the following processor assignment functions for the above loop.

$$d_b(j) = j \bmod N; \qquad d_c(j) = \left\lfloor \frac{j}{\alpha} \right\rfloor \tag{2}$$

where $\alpha = \left\lceil \frac{U}{N} \right\rceil$. Processor assignment function $d_b(j)$ assigns iterations 0, N, $2N$, ..., on processor P_0; iterations 1, $N + 1$, $2N + 1$, ..., on P_1, and so on. Assignment function $d_c(j)$ divides the iterations into contiguous blocks with identical sizes, except maybe the last one, and assigns each block to one processor. Here the assignment functions of (2) are equivalent to *Cyclic* and *Block* distribution of loop iterations among the processors. For the case of function $d_b(j)$, iterations $0 \le i < M$, $0 \le j < N$ can be represented as one execution pattern; iterations $0 \le i < M$, $N \le j < 2N$ as another identical execution pattern, and so on. For the case of function $d_c(j)$, for any two integer constants k and l, we can take iterations $i = k$, $lN \le j < (l + 1)N$ as one instance. Then the whole loop can be represented as an execution pattern of a doubly nested loop.

Singly nested loops and multiply nested loops with nesting level greater than two can also be easily handled by our approach. As singly nested loop is just a special case of double loop with $M = 0$. For the general cases of multiply nested loop of the form,

For i_1 = 0 **To** M_1 - 1 Do
...
  For i_{l-1} = 0 **To** M_{l-1} - 1 **Do**
    For i_l = 0 **To** M_{l-1} **On** Pd(i_l) **Do**
      ...
        For i_n = 0 **To** M_{n-1} **Do**
          Statements
          ...
EndFor

we can take the inside loop body after index i_l as one statement. Then the loop body starting from index i_{l-1} can be represented as an execution pattern as above. Here, the function $d(i_l)$ specifies the distribution of loop iterations among the available processors, that is, the ϕ function.

For the convenience of the readers, we end this section by summarizing, in Table 1, the important symbols used in this paper. Some of the symbols in Table 1 have not been discussed yet and will be introduced in the next section.

TABLE 1
IMPORTANT SYMBOLS USED IN THIS PAPER

Symbols	Meaning
$\phi(u)$, $\phi_r(u)$	scheduling functions
δ	data dependence relation
$\sigma(i, j)$, σ^u	statement instances
β_h, β_v, β_t	numbering functions
$\Delta(u)$, $\Delta_r(u)$	timing functions
$d_b(j)$, $d_c(j)$	processor assignment functions

4 DUPLICATION STRATEGIES FOR REGULAR LOOPS

Duplicating and comparing the results of executing statement instances is the basis for the proposed approach to fault detection. In order to implement this statement duplication approach, two 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. Our goal is to reduce, as much as possible, the overhead incurred by the fault-detection mechanism. Before we discuss duplication strategies we discuss the fault model used in this paper. We consider only single transient faults. The case of permanent faults is simpler to handle and is discussed in [19]. We assume that a processor is faulty if and only if it produces wrong results for all input data. We also assume that interprocessor communication is fault free. If errors resulting from transient faults are to be detected, then every statement instance should be duplicated.

We concentrate on duplicating instances within loop executions. The duplication of the execution of σ^u is specified by a function ϕ_r such that $\phi_r(u)$ is the set of processors on which σ^u is to execute. In general, if $|\phi_r(u)| = 2$, then an error in the execution of σ^u resulting from a fault in any of the two processors in $\phi_r(u)$ can be detected. The replica of an instance σ^u is called a *secondary instance* and is denoted by σ_s^u. To avoid confusion, we will refer to the original instance of σ^u as the *primary instance* σ_p^u. 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 $\phi_s(u)$ be the processor on which σ_s^u is to execute. Thus, if σ^u is duplicated, its mapping function can be expressed by a multivalued function $\phi_r(u) = \{\phi_p(u), \phi_s(u)\}$, where $\phi_p(u) = \phi(u)$. For a non-duplicated statement, only primary instances exist and thus, $\phi_r(u) = \{\phi_p(u)\}$.

As is the case in the absence of duplication, the copies created by duplication must also satisfy data dependencies. For an instance, σ^u, the value of the timing function Δ is now a set $\Delta_r(u) = \{\Delta_p(u), \Delta_s(u)\}$, where $\Delta_p(u)$ is the time at which σ^u executes and $\Delta_s(u)$ is the time at which σ_s^u executes. Note that $\Delta_p(u)$ may or may not be equal to $\Delta_s(u)$. Moreover, if σ^u is not duplicated, then $\Delta_r(u) = \{\Delta_p(u)\}$.

Unlike the situation of nonduplicated execution, for the duplicated timing function Δ_r, the data dependence condi-

tion is determined by the semantics of the duplication. Two different semantics are possible, the *wait-for-both* and the *wait-for-one* semantics. Specifically, if before duplication, an instance, u depends on data calculated by another instance v, then after duplication, the *wait-for-both* semantics requires that copies of σ^u do not execute before all copies of σ^v execute. This condition can be expressed as follows:

$$min\{\Delta_p(u), \Delta_s(u)\} > max\{\Delta_p(v), \Delta_s(v)\} \quad (3)$$

where we abused the notation by assuming that $max\{\Delta_p(v), \Delta_s(v)\} = \Delta_p(v)$ if σ^v is not duplicated and $min\{\Delta_p(u), \Delta_s(u)\} = \Delta_p(u)$ if σ^u is not duplicated. The wait-for-both 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.

Under the *wait-for-one* semantics, data comparison does not delay execution at the cost of possible delaying in error detection and possible error propagation. In this case, copies of σ^u can execute as soon as σ_p^v finishes execution. The results of σ_p^v and σ_s^v are compared whenever σ_s^v finishes execution. The data dependence condition under wait-for-one semantics can be expressed as $min\{\Delta_p(u), \Delta_s(u)\} > \Delta_p(v)$. 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 ensure that

$$\forall u, \Delta_s(u) \geq \Delta_p(u). \quad (4)$$

The data dependence condition under the wait-for-both semantics is stronger than that under the wait-for-one semantics. If a duplication scheme satisfies the former condition, it also satisfies the latter one. But the reverse is not true. Some of the duplication schemes discussed in this section can only satisfy the weaker data dependence condition.

After analyzing the execution pattern of a regular loop, by carefully choosing 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 of adding duplicated instances to the loops considered in the preceding section. We only present the duplication strategies. The loop transformations that use these strategies to generate the corresponding execution patterns will be presented in the next section.

Different strategies may be used to duplicate the execution of the instances in the loop. 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:

$$\phi_r(u) = \begin{cases} \{\phi(u), \phi(u)-1\} & \text{if } \phi \text{ is odd} \\ \{\phi(u), \phi(u)+1\} & \text{if } \phi \text{ is even} \end{cases} \quad (5)$$

For patterns with $\theta > 1$, no duplication overhead is associated with the duplication strategy if the duplicated computations are interleaved with the original computations as shown in Fig. 4a. If $\theta = 1$, however, the duplication penalty is 100% (see Fig. 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 M. However, if $\lambda(N-1)$ is large compared to M, then an attempt should be made to use the idle time for duplicate computations. One technique that may be used if $\lambda(N-1) \geq 2M$ 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 shown in Fig. 5. In this case, the addition of the fault detection capability does not cause any duplication overhead. Formally, if $\alpha = \left\lceil \frac{M}{\lambda} \right\rceil$, then the duplication scheme is specified as follows.

$$\phi_r(u) = \begin{cases} \{\phi(u), \phi(u)+\alpha\} & \text{if } \phi(u) \bmod \alpha \text{ is even} \\ \{\phi(u), \phi(u)-\alpha\} & \text{if } \phi(u) \bmod \alpha \text{ is odd} \end{cases} \quad (6)$$

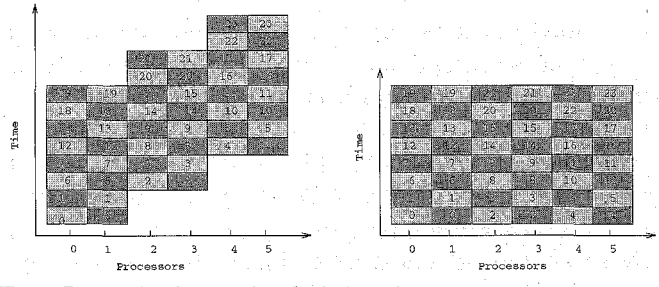


Fig. 4. Duplication for transient fault detection.

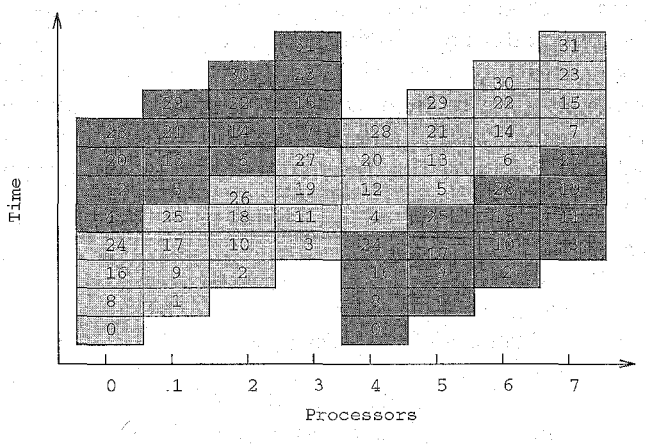


Fig. 5. Efficient duplication for Transient fault detection with $\lambda = \theta = 1$, $\lambda(N-1) \geq 2M$.

Finally, a duplication strategy for the triangular loop and the bisection loop is given by the following formula (assuming N is even).

$$\phi_r(u) = \{\phi(u), N-1-\phi(u)\} \quad (7)$$

No duplication overhead is associated with this strategy if the duplicated computation follows the execution pattern shown in Fig. 6.

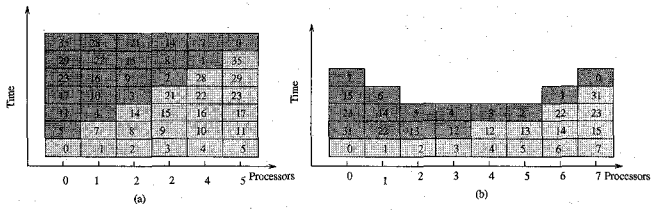


Fig. 6. (a) Duplication for triangular loop pattern; and (b) Duplication for bisection loop.

The execution patterns in Figs. 4 and 5 satisfy the stronger condition (3). This is because in all those patterns (except the one for the fully parallel loop pattern), $\Delta_s(u) = \Delta_p(u)$, for any instance u. Therefore condition (3) becomes $v\delta u \Rightarrow \Delta_p(u) > \Delta_p(v)$. In other words, if the execution patterns without duplication satisfy the data dependence condition, then the execution patterns with duplication satisfy the data dependence condition (3). In the case of the fully parallel loop pattern, since there is no data dependencies, condition (3) is automatically satisfied. It is easy to check that the execution patterns of Fig. 6 satisfy only the weaker condition (4).

5 Duplication Through Loop Transformations

In order to automatically implement duplication strategies through the compiler we must devise loop transformations that achieve duplicated computations and comparisons. Corresponding to the situations discussed in the preceding section, we will develop several *duplication transformations.*

For $i = 0$ **To** $M - 1$ **Do**
  For $j = 0$ **To** $N - 1$ **On** P_j Do
    If j *mod* $2 = 0$ **Then**
      $A(i, j) = F(A(i, j - \lambda), A(i - \theta + \lambda, j + \lambda))$
      $Temp(j) = F(A(i, j - \lambda + 1), A(i - \theta + \lambda, j + \lambda + 1))$;
      $Check(Temp(j) = A(i, j + 1))$
    Else
      $Temp(j) = F(A(i, j - l - 1), A(i - \theta + \lambda, j + \lambda - 1))$;
      $Check(Temp(j) = A(i, j - 1))$
      $A(i, j) = F(A(i, j - l), A(i - \theta + \lambda, j + \lambda))$
    EndIf
  EndFor
EndFor

Fig. 7. Loop transformations τ_{TFD}: for transient fault detection.

The first task of the transformations is to duplicate appropriate statement instances and schedule their execution on appropriate processors. We present duplication transformation for the general loop where execution pattern was expressed in terms of parameters λ and θ. Some examples of the execution patterns after transformation were illustrated in Fig. 4 of the preceding section. To detect transient faults, each instance executed by a processor is duplicated on a neighboring processor. The execution of duplicated instances is scheduled to exploit idle slots that may be present in the original loop's schedule, as shown in Fig. 7.

The second task of the transformation is to introduce code for performing the checks. Specifically, if a statement $A = f(d_1, ..., d_n)$ is selected for duplication, then the compiler inserts a secondary statement, $Temp = f(d_1, ..., d_n)$, where *Temp* is an auxiliary variable not used elsewhere in the program. The variable *Temp* should 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 execute the same statement. Thus, the two processors that execute statement $A = f(d_1, ..., d_n)$ and $Temp = f(d_1, ..., d_n)$ are able to check each other.

To show that the transformation τ_{TFD} of Fig. 7 produces the execution pattern of Fig. 4, we observed that:

1) According to the numbering function β_h, the instance which computes $A(i, j)$ is σ_p^{i*N+j}; the secondary instance σ_s^{i*N+j} is the one that computes $Temp(j + 1)$ when j is even, and the one that computes $Temp(j - 1)$ when j is odd;
2) When j is even, processor P_j will execute the instances in the following order:
$$\sigma_p^i, \sigma_s^{j+1}, \sigma_p^{N+j}, \sigma_s^{N+j+1}, \sigma_p^{2*N+j}, \sigma_s^{2*N+j+1}, \ldots$$
3) When j is odd, processor P_j will execute the instances in the following order:
$$\sigma_s^{j-1}, \sigma_p^j, \sigma_s^{N+j-1}, \sigma_p^{N+j}, \sigma_s^{2*N+j11}, \sigma_p^{2*N+j}, \ldots$$

So this transformed loop provides exactly the execution pattern shown in Fig. 4 that satisfies the data dependence condition (3).

For transformation τ_{TFD}, the checking could be done in several different ways with different efficiency and overhead. The straightforward way of checking is to perform the checking immediately after the execution of each instance, as is shown in Fig. 7. This approach results in minimal delay between the occurrence of a fault and its detection. However, in order to perform the comparison checks, the processors that have independently executed a statement must exchange the results, which generates communication traffic. The communication overhead generated due to the exchange of results could be reduced at the cost of delaying fault detection. These overhead-reducing transformations are shown in Fig. 8. The delay allows all the checks to be grouped into a single computation of larger granularity (the last loop in Fig. 8a). This, in turn, allows the messages to be combined. Even though the detection of a transient fault is delayed, we are able to identify the precise loop iteration during which an error occurred. This reduced communication-overhead is at the expense of memory overhead since it uses a two dimensional array $Temp(i, j)$. Finally we can borrow the idea from [5] to delay the checking to the end of the loop and instead of exchanging all the results, the processors exchange a single value which is the sum of all the results that are to be compared, see Fig. 8b. Thus, the overhead of communication is further reduced. However, the precise iteration in which an error occurred cannot be detected using this technique.

For i $= 0$ **To** $M - 1$ **Do**
  For j $= 0$ **To** $N - 1$ **On** P_j **Do**
    If j mod $2 = 0$ **Then**
      $A(i, j) = F(A(i, j - \lambda), A(i - \theta + \lambda, j + \lambda))$

```
        Temp(i, j) = F(A(i, j - λ + 1), A(i - θ + λ, j + λ + 1))
      Else
        Temp(i, j) = F(A(i, j - λ - 1), A(i - θ + λ, j + λ - 1))
        A(i, j) = F(A(i, j - 1),
        A(i - θ + λ, j + 1))
      EndIf
    EndFor
  EndFor
```

– The following code performs checking:

```
For i = 0 To M - 1 Do
    For j = 0 To N - 1 On Pj Do
      If j mod 2 = 0
      Then Check(Temp(i, j) = A(i, j + 1))
      Else Check(Temp(i, j) = A(i, j - 1))
      EndIf
    EndFor
  EndFor
```

(a) τ_{GC}: Delayed Checking with Grouped Communication.

```
Initialize Sum and Temp arrays to 0
For i = 0 To M - 1 Do
    For j = 0 To N - 1 On Pj Do
      If j mod 2 = 0
      Then A(i, j) = F(A(i, j - λ), A(i - θ + λ, j + λ));
        Sum(j) = Sum(j) + A(i, j)
        Temp(j) = Temp(j) + F(A(i, j - λ + 1), A(i - θ + λ, j + λ + 1))
      Else
        Temp(j) = Temp(j) + F(A(i, j - λ - 1), A(i - θ + λ, j + λ - 1))
        A(i, j) = F(A(i, j - l), A(i - θ + l, j + l));
        Sum(j) = Sum(j) + A(i, j)
      EndIf
    EndFor
  EndFor
```

– The following code performs checking

```
For j = 0 To N - 1 On Pj Do
    If j mod 2 = 0
      Then Check(Temp(j) = Sum(j + 1))
      Else Check(Temp(j) = Sum(j - 1))
    EndIf
  EndFor
```

(b) τ_{RC}: Delayed Checking with Reduced Communication.

Fig. 8. Transformations for efficient checking.

To show how to achieve the efficient duplication shown in Fig. 5, we present the following transformation for the case where $\lambda = \theta = 1$ and $(N - 1) \geq 2M$. In this transformed loop, we do not show the checking statement(s). The checking approaches shown in Fig. 8 can also be applied here.

For the triangular loop and the bisection loop the transformation shown in Fig. 10 achieves the capability of detecting transient faults. In this loop, $mirror(j) = N - 1 - j$. With an argument similar to the one used for the transformations of Fig. 7, it is easy to show that transformations τ_{IP} and τ_{TL} result in the execution patterns of Figs. 5 and 6, respectively.

```
For i = 0 To M - 1 Do
    For j = 0 To N - 1 On Pj Do
      If j < N/2 Then
        A(i, j) = F(A(i, j), A(i, j - 1))
      Else
        Temp(i, j - N/2) =
        F(A(i, j - N/2), A(i, j - N/2 - 1))
      EndIf
    EndFor
  EndFor
  For i = 0 To M - 1 Do
    For j = 0 To N - 1 On Pj Do
      If j < N/2 Then
        Temp(i, j + N/2) = F(A(i, j + N/2), A(i, j + N/2 - 1))
      Else
        A(i, j) = F(A(i, j), A(i, j - 1))
      EndIf
    EndFor
  EndFor
```

Fig. 9. τ_{IP}: Transformation for efficient usage of idle processors.

```
For i = 0 To M - 1 Do
  For j = f(i) To N - 1 On Pj Do
    A(i, j) = F(A(i - 1, j - l))
  EndFor
EndFor
For i = 0 To M - 1 Do
  For j = i To N - 1 On P_mirror(j) Do
    Temp(i, j) = F(A(i - 1, j - l))
  EndFor
EndFor
```

Fig. 10. τ_{TL}: Transformation of triangle and bisection loops.

In all the loop transformations shown in this section, in computing the secondary copy *Temp*, we always use the data computed in a primary copy, that is, operands are accessed from array *A*. Actually, in all the transformations, except the one for the triangular and bisection loops, it is also possible to use the values computed by a secondary copy, that is, operands can be accessed from array *Temp*, as long as the transformation generates a legal execution pattern that satisfies the wait-for-all condition. The choice of whether the value computed by the primary or the secondary copy is used may be driven by other consideration, such as communication efficiency. For the transformation of the triangular and bisection loops, since it satisfies only the wait-for-one condition, it must use the values computed by the primary copies to compute all secondary copies.

6 A General Loop Transformation Framework

The loop transformation techniques described in the last section rely on data dependence analysis. Accurate data dependence information enables efficient loop transformation which can reduce the overhead caused by the added fault-detection capability. In this section, we present a general framework which, for a given regular loop, selects the appropriate transformation for fault detection. Furthermore, the framework can estimate the overhead incurred by the transformed loop.

Given a multiply-nested regular loop, each data dependence can be represented by a dependence distance vector $\delta = (d_1, \ldots, d_m)$, where d_i is the dependence distance along the ith dimension. Thus for any regular loop L, all the dependencies can be represented by a set of dependence vectors, $D(L) = \{\delta_1, \ldots, \delta_l\}$. For example, for the following loop, the dependence distance vector set is $D(L) = \{(0, 1), (1, -1)\}$.

```
For i = 0 To M - 1 Do
  For j = 0 To N - 1 Do
```

$$\sigma(i, j)\begin{cases} A(i, j) = A(i, j-1) + A(i-1, j+1) \\ B(i, j) = A(i, j) * C \end{cases}$$

EndFor
EndFor

A loop is called a fully parallel loop if all the processors allocated to this loop can execute their assigned instances in a fully parallel fashion. The following theorem captures the property of a fully parallel loop in terms of its dependence vectors.

THEOREM 1. *Let L be a doubly-nested regular loop, with a scheduling function, $\phi(\sigma(i, j)) = j \pm c$ for some integer constant c. All processors can execute their assigned instances in a fully parallel fashion, if the following condition holds.*

$$(D(L) = \varnothing) \quad \vee \quad (\forall \delta = (d_1, d_2) \in D(L), d_1 > 0) \tag{8}$$

The above condition implies that L is the general execution pattern with $\lambda = 0$ and $\theta = 1$. This theorem is also true for the case in which $\phi(\sigma(i, j)) = i \pm c$ and $d_2 > 0$.

PROOF. Let instance $\sigma(i, j)$ corresponds to loop iteration (i, j). Processor P_j executes instances $\sigma(i, j)$ where $0 \le i \le M$. If $D(L) = \varnothing$, namely there is no dependence at all, then it is obvious that the loop is a fully parallel loop. Suppose $D(L) \neq \varnothing$, we can prove the theorem by induction on the time t. For $t = 1$, processor P_j has to execute instance $\sigma(0, j)$. Since $d_1 > 0$ for any dependency $\delta = (d_1, d_2)$, $\sigma(0, j)$ doesn't depend on any other instance of the loop. So for the first time unit, all processors P_j, where $0 \le j < N$, can execute their first instance in parallel. Suppose that for $t < n$, all processors execute their instances in parallel. Then at time unit $t = n$, instances $\sigma(i, j)$, $i < n$ are already finished. Again, since $d_1 > 0$ for any dependency, instance $\sigma(n, j)$ can only depend on some instance $\sigma(i, j)$ with $i < n$ which has already been executed. So processor P_j can execute $\sigma(n, j)$ at time n. □

From the data dependence condition, we know that for any two instances σ and σ', if there is a dependence path form σ to σ' of length p, then the earliest time slot in which σ' can be executed is at least p time slots later than the time slot in which σ is executed. The following theorem specifies the condition under which there will be a dependence path of length p from one instance to another instance.

THEOREM 2. *For any doubly-nested regular loop L, if there is a set $\{\delta_1, ..., \delta_p\}$ such that*

$$\left(\{\delta_1, ..., \delta p\} \subseteq D(L)\right) \wedge \left(\sum_{k=1}^{p} \delta_k = (d_1, d_2)\right) \wedge (d_1 \neq 0 \vee d_2 \neq 0) \tag{9}$$

then there is a dependence path of length p from $\sigma(i, j)$ to $\sigma(i + d_1, j + d_2)$.

PROOF. We prove this theorem by using induction on p. For the case $p = 1$, since there is a dependence vector $\delta_1 = (d_1, d_2)$, by definition of dependence distance, it is true that $\sigma(i, j)\delta\sigma(i + d_1, j + d_2)$. Assume the result is true for $p = n - 1$. For $p = n$, let the sum of the first $n - 1$ dependence distance vectors be (d_{11}, d_{12}), the last one be $\delta_n = (d_{21}, d_{22})$, and $d_{11} + d_{21} = d_1$, $d_{12} + d_{22} = d_2$. Since the result holds for $p = n - 1$, there is a dependence path of length $n - 1$ from $\sigma(i, j)$ to $\sigma(i + d_{11}, j + d_{12})$; and due to δ_n, $\sigma(i + d_{11}, j + d_{12})\delta\sigma(i + d_{11} + d_{21}, j + d_{12} + d_{22}) = \sigma(i + d_1, j + d_2)$. So there is a dependence path of length n from $\sigma(i, j)$ to $\sigma(i + d_1, j + d_2)$. □

COROLLARY. *For any double-nested regular loop L, if there is a set $\{\delta_1, ..., \delta_p\}$ such that*

$$\left\{\delta_1, \ldots, \delta_p\right\} \subseteq D(L) \wedge \sum_{k=1}^{p} \delta_k = (1, 0) \tag{10}$$

then there is a dependence path of length p from $\sigma(i, j)$ to $\sigma(i + 1, j)$.

Suppose $\sigma(i, j)$ and $\sigma(i + 1, j)$ are two consecutive instances assigned to the same processor, then the longest dependence path from $\sigma(i, j)$ to $\sigma(i + 1, j)$ will determine the idle slots for the processor.

THEOREM 3. *Let L be a double-nested regular loop, with a scheduling function $\phi(\sigma(i, j)) = j$. If the longest dependence path from $\sigma(i, j)$ to $\sigma(i + 1, j)$ is of length k, for some $k > 1$, then L is a general rectangular loop with $\lambda > 0$ and $\theta = k$. This is also true for the case that $\phi(\sigma(i, j)) = i$ and the longest dependence path from $\sigma(i, j)$ to $\sigma(i, j + 1)$ is of length k, for some $k > 1$.*

PROOF. Since there is a dependence path of length greater than 1 from $\sigma(i, j)$ to $\sigma(i + 1, j)$, there must exist at least one dependence from instance $\sigma(i, j)$ to $\sigma(i, j + c)$ for some $c > 0$ and there is a dependence path from $\sigma(i, j + c)$ to $\sigma(i + 1, j)$. This is also true for $i = 0$. Therefore not all instances $\sigma(0, j)$ can be executed at time $t = 0$ due to the dependence, that is, $\lambda > 0$. Furthermore, the value of λ is the minimal c such that $\sigma(i, j)\delta\sigma(i, j + c)$. Suppose instance $\sigma(i, j)$ is executed at time t_i and $\sigma(i + 1, j)$ at time t_j by the same processor, due to the dependence path from $\sigma(i, j)$ to $\sigma(i + 1, j)$, it must be the case that $t_j - t_i > 1$, that is, $\theta > 1$. □

The next theorem characterizes a skewed loop in terms of its dependence vectors.

THEOREM 4. *Given a doubly-nested regular loop L with a scheduling function $\phi(\sigma(i, j)) = j$ such that the processors cannot execute their assigned instances in parallel. The loop L is a general rectangular loop with $\lambda > 0$ if*

$$\forall \delta = (d_1, d_2) \in D(L), d_2 > 0 \tag{11}$$

PROOF. Since the processors cannot execute their instances in parallel the conditions in theorem 1 are not true. Thus, there must be at least one dependence $\delta = (d_1, d_2)$ with $d_1 = 0$ which inhibits parallel execution of all instances $\sigma(0, j)$. Consequently, $\lambda > 0$. □

The above theorems provide the foundation for the following general loop transformation framework. This framework examines the following regular loop L, where $f(i)$ is either i or 0, and determines the transformation that should be applied to the loop for fault detection.

For $i = 0$ **To** M -1 **Do**
  For $j = f(i)$ **To** N - 1 **Do**
    $\sigma(i, j)$
  EndFor
EndFor

STEP 1. Determine the scheduling function $\phi(i, j)$ for statement instance (i, j).

STEP 2. Apply data dependence analysis to find all data dependencies, $D(L) = \{\delta_1, ..., \delta_l\}$.

STEP 3. Select a transformation for the loop L according to its execution pattern as follows.

(a) **If** $f(i) > i$ **Then** apply the transformation t_{TL}.
(b) **Else If** $(D(L) = \varnothing) \vee (\forall\ \delta = (d_i, d_j) \in (L), d_i > 0)$ **Then**
/* a fully parallel loop by theorem 1.*/
apply transformation τ_{TFD} to detect faults;
(c) **Else If** $\exists D'(L) \subseteq D(L), |D'(L)| > 1 \wedge \Sigma\ D'(L) = (1, 0)$ **Then**
/* a general loop by theorem 3 */
apply transformation τ_{TFD} to detect faults;
(d) **Else If** $\forall\ \delta = (d_i, d_j) \in D(L), d_j$ 8 0 **Then**
/* a skewed loop by Theorem 4. */
If $\lambda(N - 1) \geq 2M$ **Then** apply transformation τ_{TP};
Else apply transformation τ_{TFD} to detect faults;

In the above framework, in the place of transformation τ_{PFD}/τ_{TFD}, we may use the transformation τ_{GC} or τ_{RC} to reduce the overhead if we do not care when or where the fault occurs. In this framework, we first obtain the ϕ function and apply data dependence analysis to get the set of dependence distance vectors $D(L)$. Then based on the ϕ function and $D(L)$, the execution pattern of the loop L can be determined.

For Case 3b, by Theorem 1, L is a fully parallel loop. The overhead of detecting transient fault is the sum of the original execution time, comparison time, and communication time. This overhead would be much more than 100%. By applying the grouping communication technique and delayed comparison shown in Fig. 8, the overhead can be reduced to almost 100%, which is the best one can expect for a fully parallel loop. To detect permanent fault, only the last statement instance of each processor needs to be duplicated. Generally, the overhead would be just a small percentage of the original execution time. For Case 3c, by Theorem 2, L is the general rectangular loop with $\theta > 1$ and $\lambda > 0$. Each processor will be idle for $(\theta - 1)$ time slots after it executes one instance. So we can exploit the spare capacity for the purposes of fault detection.

7 Performance Measurements

In order to empirically estimate the overhead of the compiler-assisted fault-detection approach, we have developed a testing environment (TE) on the multiprocessor computer CRAY-T3D using PVM [18]. At this time, the TE does not have a full-fledged compiler for the parallel programming model. Instead, programs are translated into parallel intermediate code which is executed by interpreters running on each processors of the CRAY-T3D. Interprocessor communication is carried out by the PVM primitives. We first request a subsystem of the CRAY-T3D computer (up to 128 processing elements) and then load the interpreter to each of those PEs. The intermediate code provided to the interpreter includes the necessary communication instructions. The interpreter(s) read in the program from a file and execute it through interpretation. We carried out experiments using benchmark programs belonging to various categories. First we explain results for the *general loop* with $\lambda = 1$, $\theta = 2$, and then *skewed loop*, and finally we summarize the results for all of the benchmark loops considered.

We applied transformations τ_{PFD}, τ_{TFD}, τ_{GC} and τ_{RC} to the following doubly-nested loop and executed it under different granularity and on different number of processors.

For i = 0 **To** 511 **Do**
  For j = 0 **To** K **On** $P_{d_b(j)}$ **Do**
    $A(i, j) = (A(i, j - 1) + A(i - 1, j + 1) +$
    $A(i - 1, j) + A(i + 1, j))/4$
  EndFor
EndFor

The above loop can be found in applications such as image processing and the numeric solution of two-dimensional partial differential equations. It produces the general execution pattern with $\lambda = 1$ and $\theta = 2$. We first fixed the processor number N to 16 and varied the iteration number K. Then we fixed $K = 65{,}536$ and used different number of processors. The results are shown in Fig. 11. The overhead is given by

$$O_r = \frac{T_t - T}{T} \times 100\%$$

where T_t and T are the execution time (in CPU clock cycle) of the loop after and before the transformation, respectively. Since the function $d_b(j)$ distributes a *block* to each processor, when K becomes larger each processor gets a larger *block*. From Fig. 11a, we can see that for large computation instances (large K) the overhead ratio increases slightly. From Fig. 11a and 11b, it is clear that the overhead is relatively low for wide ranges of K and N (between 7% and 24%). Intuitively, the overhead of duplicating execution should be more than 100% without exploiting any idle slots. By exploiting the idle slots, the overhead is reduced to less than 25%. Furthermore, the delayed checking technique is quite effective.

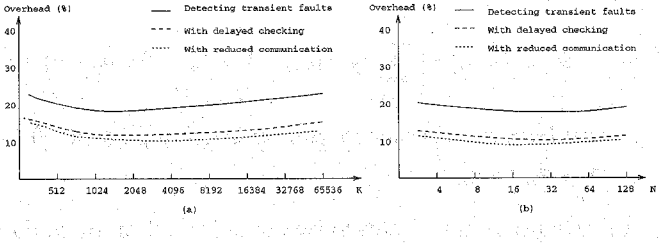


Fig. 11. Overhead for general loop ($\lambda = 1$, $\theta = 2$) with respect to (a) varying granularity K; and (b) varying number of processors N.

To measure the overhead of different transformations on a *skewed loop* (execution pattern with $\lambda = \theta = 1$), we considered Livermore Loop 5 and rewrote it into the form given below:

For j = 0 **To** K **On** $P_{d_b(j)}$ **Do**

$X(j) = Z(j)*(Y(j) - X(j - 1))$

EndFor

Since $M = 1$ for this loop, $\lambda(N - 1) > 2M$ is true for $N > 3$. Therefore we applied the transformation τ_{IP} of Fig. 9. The experiment results, which are given in Fig. 12, are consistent with the previous results.

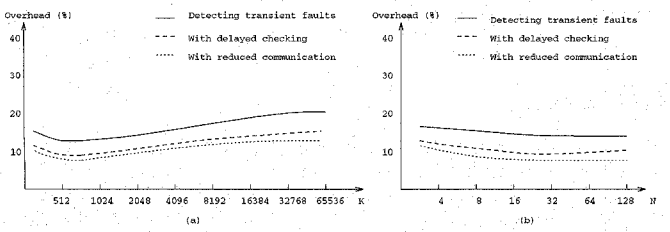


Fig. 12. Overhead for skewedloop ($\lambda = 1$, $\theta = 2$) with respect to (a) varying granularity K; and (b) varying number of processors N.

We applied loop transformations for fault detection to all the regular loops from the Livermore Loop benchmark [17]. The processor number is fixed as $N = 64$, and inner loop iteration is fixed as $K = 65{,}536$. The results are summarized in the Table 2.

TABLE 2
AVERAGE OVERHEAD OF FAULT DETECTION FOR LIVERMORE LOOPS

	skewed loops	triangular loops	bisection loops	fully parallel loops
Number of loops	5	2	2	8
Detect transient faults	19.5	12.3	12.3	115.4
Delayed checking	8.1	7.1	7.0	107.9
Reduced communication	8.0	7.0	7.0	107.6

As can be seen, the overhead of detecting either transient or permanent faults is relatively low for *skewed* and *triangular* loops. For the eight fully parallel loops, more than 100% average overhead for detecting transient faults is no surprise since there is no idle time slots to exploit. However, for skewed loops and triangular loops the overhead is small because the idle slots were effectively exploited by the transformations.

The impact of the extra communications introduced by the checking on the total execution time is relatively small because

1) the results sent for checking can usually be sent in messages carrying the data needed for computation;
2) the communication delay is usually overlapped with computations; and
3) the results for one duplicated instance are grouped in one message and usually the granularity of a duplicated instance is much larger than one statement instance.

Hence, the actual number of communications required by the checking operations is much smaller than the number of duplicated results.

8 Related Work

Replication of computations for fault detection and fault tolerance can be carried out at various levels of granularity. For example, it could be done at process level [29], at transaction level [28], at procedure level [14], or at statement level as we proposed in this paper. While replication of computation at a coarse-grained level incurs smaller overhead, it has a serious drawback. A fault is detected only at the end of the whole computation irrespective of the time at which the fault occurs. For computation that needs long time to complete, this is undesirable.

The approach of [5] is also a statement level duplication approach. The authors extend the algorithm-based checking techniques to deal with more general applications by exploiting *linearity property* of Fortran Do loops. For a loop with loop variables $1 \le i \le m$ and j, suppose $A(i)$ is the set of array variables with index i, $B(j)$ the set of array variables with index j, and $C(i, j)$ the set of array variables with indices (i, j). Then, a statement $D(i, j) = f(A(i), B(j), C(i, j))$ is said to be i-linear if for some w_k, $1 \le k \le m$,

$$\sum_{k=1}^{m} w_k f\big(A(k), B(j), C(k, j)\big) = f\left(\sum_{k=1}^{m} A(k), B(j), \sum_{k=1}^{m} w_k C(k, j)\right) \quad (12)$$

The variable i is said to be the *checking variable*. For loops that satisfy the *linearity property*, the two sides of (12) can be computed on two different processors and compared to detect a fault. We compare the approach of [5] and the approach of this paper as follows.

1) *Class of Loops*: While the approach proposed in this paper deals with the class of *regular loops*, the approach of [5] deals with the class of loops with *linearity property*. These two classes are not the same and neither of these two classes is a superset of the other.
2) *Overhead*: For loops that are regular and satisfy the linearity property, we can compare the two approaches as follows. The approach proposed in this paper achieves low cost by exploiting the idle time slots during the execution of a regular loop. The approach of [5] does not explicitly exploit idle processors that might exist during the execution of a program. Instead, it tries to reduce the overhead through carefully choosing the checking variable to reduce the duplicated computations and data exchanges. This technique achieves low cost fault-detection only for special applications such as matrix multiplication. For example, the image processing loop given in the last section satisfies the linearity property. Even though there are many idle time slots during the execution of the loop, the approach of [5] will not be able to exploit them since it has to compute the two sides of (12) at the end of the loop. Therefore, for these kind of loops, the approach of this paper will achieve lower overhead than the approach of [5]. The advantage of the approach of [5] is in dealing with fully parallel loops, since it can detect transient faults with less than 100% overhead. It also pioneered the using of delayed checking technique.

3) *Round Off Error*: Another problem of the approach of [5] is that the round off errors accumulate in different ways for the two sides of (12), thus making it highly unlikely that the equality is preserved exactly [12]. This problem does not exist in the approach proposed in this paper since exactly the same computations are duplicated.

From the above discussion, we can see that the two approaches are complementary to each other. It would be a good idea to combine the approach of this paper and the approach of [5] to develop a more powerful system that can

1) deal with both *regular loops* and loops with *linearity property*;
2) deal with both loops that produce many idle time slots at execution time and loops that produce few or no idle time slots.

9 Concluding Remarks

We proposed a compiler-assisted approach to fault detection on distributed-memory systems and developed a loop transformation framework through which a regular loop is transformed to duplicate computations and introduce checks to detect faults. This approach achieves the following goals:

1) *Applicability*: No specialized hardware is required and the techniques used are applicable to existing systems. The fault detection approaches are implemented entirely in software through a compiler.
2) *On-line Checking*: The checking is performed concurrently with the execution of application programs. This is essential for detecting transient faults which occurs more often than permanent faults.
3) *Efficiency*: Analysis techniques are developed to enable the merging of duplicated computation with the original computation in a manner that reduces the increase in overall execution time. The results of experiments show that full coverage can be obtained with small overhead for non-fully-parallel regular loops.
4) *Automatic Implementation*: The implementation of the analysis techniques and fault detection algorithms are carried out through the compiler. Thus, the augmentation of a program for fault tolerance can be achieved without any assistance from the user.

In this paper, we concentrated on scheduling duplicated computations on idle slots that result from data dependencies. Communication delay may also create idle slots during execution. The efficient utilization of those slots for scheduling duplicated computations is studied in detail in [20]. Furthermore, the work of this paper can be expanded as follows. First, the extension of the proposed approach to fault location and error masking, specifically, by triplicating computation instances on three processors. Second, the study of efficient replication strategies for program constructs that are more general than the loop construct discussed in this paper. Third, the design of a user interface that provides necessary information to the user and allows the user to select the duplication strategy.

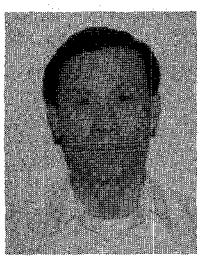
Acknowledgments

The authors thank the anonymous referees for their thorough reviewing and helpful comments. The presentation of this paper has been greatly improved after the authors followed the suggestions made by the anonymous referees. This work was supported, in part, by an MPC grant ASC-8092826 and National Science Foundation through a Presidential Young Investigator Award CCR-9157371 to the University of Pittsburgh.

References

[1] N. Alewine, S. Chen, C. Li, W. Fuchs, and W. Hwu, "Branch Recovery With Compiler-Assisted Multiple Instruction Retry," *Proc. 22nd Ann. Int'l Symp. Fault-Tolerant Computing*, pp. 66-73, 1992.
[2] J.M. Anderson and M.S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, pp. 112-125, 1993.
[3] T. Anderson, P. Barrett, D. Halliwell, and M. Moulding, "Software Fault Tolerance: An Evaluation," *IEEE Trans. Software Eng.*, vol. 11, no. 12, pp. 1,502-1,510, Dec. 1985.
[4] A. Avizienis and J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer*, vol. 17, no. 8, pp. 67-80, Aug. 1984.
[5] V. Balasubramanian and P. Banerjee, "Compiler-Assisted Synthesis of Algorithm-Based Checking in Multiprocessors," *IEEE Trans. Computers*, vol. 39, no. 4, pp. 436-446, Apr. 1990.
[6] D. Blough and G. Masson, "Performance Analysis of a Generalized Concurrent Error Detection Procedure," *IEEE Trans. Computers*, vol. 39, no. 1, pp. 47-62, Jan. 1990.
[7] D. Blough and A. Nicolau, "Fault Tolerance in Super-Scalar and VLIW Processors," *Proc. IEEE Workshop Fault-Tolerant Parallel and Distributed Systems*, pp. 193-200, 1992.
[8] M. Breuer and A. Ismaeel, "Roving Emulation as a Fault Detection Mechanism," *IEEE Trans. Computers*, vol. 35, no. 11, pp. 933-939, Nov. 1986.
[9] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *J. Supercomputing*, pp. 151-169, Feb. 1988.
[10] B. M. Chapman, P. Mehrotra, and H. P. Zima, "Programming in Vienna Fortran," *Scientific Programming*, pp. 31-50, Jan. 1992.
[11] B. M. Chapman, P. Mehrotra, and H. P. Zima, "High Performance Fortran Without Templates: An Alternative Model for Distribution and Alignment," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 92-101, 1993.
[12] A. R. Chowdhury and P. Banerjee, "Tolerance Determination for Algorithm-Based Check Using Simplified Error Analysis Techniques," *Proc. 1993 FTCS*, pp. 290-298, June 22-24, 1993.
[13] "CM Fortran User's Guide for the CM-5," *Thinking Machines*, 1992.
[14] E. Cooper, "Replicated Distributed Programs," *Proc. 10th ACM Symp. Operating System Principles*, pp. 63-78, Dec. 1-4, 1985.
[15] R. Cytron and J. Ferrante, "What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation," *Proc. Int'l Conf. Parallel Processing*, pp. 19-27, Aug. 1987.
[16] A. Dahbura, K. Sabnani, and W. Hery, "Spare Capacity as a Means of Fault Detection and Diagnosis in Multiprocessor Systems," *IEEE Trans. Computers*, vol. 38, no. 6, pp. 881-891, June 1989.
[17] J. Feo, "An Analysis of the Computational and Parallel Complexity of the Livermore Loops," *Parallel Computing*, pp. 163-185, July 1988.
[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 User's Guide and Reference Manual," Oak Ridge National Laboratory, Oak Ridge, TN 37831.
[19] C. Gong, R. Melhem, and R. Gupta, "Compiler Assisted Fault Detection for Distributed-Memory Systems," *Proc. 1994 Scalable High Performance Computing Conference*, Knoxville, Tenn, pp. 373-380, May 23-25, 1994.
[20] C. Gong, "Compiler-Assisted Approaches to Fault Detection on Distributed-Memory Systems, PhD thesis.

[21] "High Performance Fortran Forum," *DRAFT High Performance Fortran Language Specification, Ver. 1.0.* Technical Report, Rice Univ., Jan. 1993.
[22] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng, "An Overview of the Fortran D Programming System," *Proc. Fourth Workshop Languages and Compilers for Parallel Computing,* pp. e1-e17, 1991.
[23] C. Koelbel, P. Mehrotra, and J. Rosendale, "Supporting Shared Data Structure on Distributed Memory Architectures," *Proc. Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming,* pp. 177-186, SIGPLAN, ACM, 1990.
[24] J. Long, W. Fuchs, and J. Abraham, "Compiler-Assisted Static Checkpoint Insertion," *Proc. 22nd Ann. Int'l Symp. Fault-Tolerant Computing,* pp. 58-65, 1992.
[25] M. Quinn, P. Hatcher, and K. Jourdenais, "Compiling C* Programs for a Hypercube Multicomputer," *Proc. ACM/SIGPLAN PPEALS,* pp. 57-65, July, 1988.
[26] M. Quinn and P. Hatcher, "Compiling SIMD Programs for MIMD Architecthres," *Proc. Int'l Conf. Computer Languages,* pp. 291-296, 1990.
[27] L. Shombert and D. Siewiorek, "Using Redundancy for Concurrent Testing and Repairing of Systolic Arrays," *Proc. 17th Int'l Symp. Fault-Tolerant Computing,* pp. 244-249, 1987.
[28] T. P. Ng, "Replicated Transactions," *Proc. Ninth Int'l Conf. Distributed Computing Systems,* pp. 474-481, 1988.
[29] S. Tridandapani, A. Somani, and U. Sandadi, "Low Overhead Multiprocessor Allocation Strategies Exploiting System Spare Capacity for Fault Detection and Location," *IEEE Trans. Computers,* vol. 44, no.7, pp. 865-877, July, 1995.
[30] M. Wolfe, *Optimizing Supercompiler for Supercomputers.* Cambridge, Mass.: MIT Press, 1989.



Chun Gong received the BS degree in computer science from the Peking University, Beijing, People's Republic of China, in 1982, and the PhD degree in computer science from the University of Pittsburgh in 1995. From 1985 to 1988,he was a research assistant at the Institute of Software, Academia Sinica. From 1988 to 1990, he was visiting the School of Computer Science, Carnegie Mellon University. Currently, he is a software design engineer at Hewlett-Packard's Massachusetts Language Lab. His primary interests include compiler optimization, parallelizing compilers, and reliable distributed computing.

Rami Melhem received a BE degree in electrical engineering from Cairo University in 1976, an MA degree in mathematics and an MS degree in computer science from the University of Pittsburgh in 1981, and the PhD degree in computer science from the University of Pittsburgh in 1983.

He is a professor of computer science at the University of Pittsburgh. Previously, he was an assistant professor at Purdue University and an assistant and associate professor at the University of Pittsburgh.

He has published numerous papers in the area of fault tolerance, systolic architectures, parallel computing, and optical computing. He served on program committees of several conferences and workshops, and is general chair of the Third International Conference on Massively Parallel Processing Using Optical Interconnections.

He is on the editorial board of *IEEE Transactions on Computers*, and was a guest editor of a special issue of the *Journal of Parallel and Distributed Computing on Optical computing and Interconnection Systems.*

Dr. Melhem is a member of the IEEE Computer Society, the ACM, and the International Society for Optical Enginering.

Rajiv Gupta received the BTech degree in electrical engineering from the Indian Institute of Technology, New Delhi, in 1982, and the PhD degree in computer science from the University of Pittsburgh, in 1987. He was a senior member of the research staff at Philips Laboratories from 1987 to 1990. Currently, he is an associate professor in the Department of Computer Science at the University of Pittsburgh.

His primary areas of research interest include parallelizing compilers, parallel architectures, and compilation techniques. Dr. Gupta is a recipient of the National Science Foundation's 1991 Presidential Young Investigator Award. He serves as an Associate Editor for *Parallel Computing Journal*, and has served as a program committee member for several conferences in the field of parallel architectures and compilation techniques.

Dr. Gupta is a member of the IEEE Computer Society, ACM, Sigplan, and Sigarch.6