Compiler Assisted Fault Detection for Distributed-Memory Systems

Chun Gong
gong@cs.pitt.edu

Rami Melhem
melhem@cs.pitt.edu

Rajiv Gupta
gupta@cs.pitt.edu

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Distributed-memory systems provide the most promising performance to cost ratio for multiprocessor computers due to their scalability. However the issues of fault detection and fault tolerance are critical in such systems since the probability of having faulty components increases with the number of processors. We propose a methodology for fault detection through compiler support. More specifically, we augment the single-program multiple-data *(SPMD) execution model to duplicate selected data items in such a way that during execution, whenever a value of a duplicated data is computed, the owners of the data are tested. The proposed compiler assisted fault detection technique does not require any specialized hardware and allows for a selective choice of redundancy at compile time.*

1 Introduction

Distributed-memory systems provide a promising performance to cost ratio for multiprocessor computers. Such systems can incorporate thousands of processors at a reasonable cost. However due to the large number of processors, fault detection and fault tolerance become critical issues. The probability of having faulty components increases with the number of processors. The traditional approach to fault-tolerant computing is to add specialized hardware to perform fault detection efficiently [4] [12]. Recently, it has been realized that the computing power of a multiprocessor system is rarely completely used. Therefore it is natural to use the spare capacity of the multiprocessor systems for fault detection and fault location [6] [13]. By replicating a task on more than one processor and comparing the results obtained, errors can be detected and masked. The fault detection is performed at the task level and an error can be detected only after a task is completed. The main advantage of this approach is that no additional hardware is required for fault detection and location.

In this paper, we propose a compiler-assisted methodology for fault detection in distributed-memory systems. We augment the *single-program multi-data* (SPMD) execution model for programming distributed-memory systems with fault detection capabilities. In this model, a data item that is stored in the local memory of a processor is said to be owned by that processor, and it is assumed that the owner processor of a data executes the statements that compute the value of that data. If a data item, d, is replicated on more than one processor, the statements that compute the value of d will be executed by each of these processors, making it possible to test these processors by comparing their values of d. Given any program, the compiler selects the data items for replication in such a way that, during the execution of the program, all active processors are tested. The degree of fault coverage is controlled by the degree of data replication. If a data item is selected for replication, then whenever the value of the data item is computed in the program the owner processors of the data are tested. The advantages of the proposed compiler-assisted fault detection approach include:

1. Better efficiency: the fault detection is done at the statement level instead of the task level;
2. Flexibility: full control over which processor is to be tested and how should it be tested is exercised at compile time;
3. Ease of implementation: no specialized hardware is needed. Only the compilation techniques for SPMD execution are modified; and
4. On-line fault detection: the test is done during the execution of a program.

The idea of taking advantage of compiler techniques for the purpose of fault detection and tolerance is not new. In [2], a loop transformation to perform fault detection on multiprocessor systems is presented. In [1], a compiler-assisted scheme to enable a process to quickly recover from transient faults is given. A method that utilizes a VLIW compiler to schedule redundant operations into idle functional units for fault detection purposes is proposed in [3]. One reason that makes compiler approaches appealing is the fact that compilers can apply a variety of analysis techniques to efficiently allocate resources in multiprocessor systems. Our approach is unique in that it exploits the dependencies of a program to reduce the overhead incurred by fault detection strategies.

The multiprocessor system considered in this paper consists of N processors connected by an interconnection network. Each processor has its own local memory and there is no global memory. Interprocessor communication is carried out through message passing primitives. Transient or permanent faults may occur in any of the N processor modules, and it is assumed that a fault causes the processor to produce a wrong result. It is also assumed that two faulty processors do not produce the same wrong result.

We organize the paper as follows. Section 2 introduces the SPMD execution model. In Section 3, we first present the principle of fault detection through data duplication and then discuss several different fault detection strategies. In Section 4, some experiment results are presented, and our conclusions are given in Section 5.

2 The SPMD Execution Model

The SPMD execution model provides a powerful approach for programming distributed-memory systems [5] [10] [11] [7] [9]. In this paper, we assume that the user writes a sequential program and uses directives to specify the distribution of data. The compiler then translates the program to execute in SPMD fashion according to the owner-computes rule. A processor examines the statements in the program sequentially, and for each statement, it takes one of the following actions: (a) performs the operation indicated by the statement if it owns the data element whose value is being computed; (b) sends a local data to the processor that needs the data to execute the statement; or (c) skips the statement. Communication instructions are introduced by the compiler. The communication primitives are non-blocking *send* and blocking *receive*. We use the following notation for *send* and *receive* instructions:

```
$S(name, P_s \rightarrow P_d)$ :
    $P_s$ $Sends$ $\{name = val(name)\}$ $to$ $P_d$
$R(name, P_d \leftarrow P_s)$ :
    $P_d$ $Receives$ $\{name = val(name)\}$ $from$ $P_s$
```

The basic compilation rules for SPMD execution are given in [5]. Here, we briefly discuss some key points of the basic compilation. Two functions, *Owner* and *Loc*, are defined for each data element d. The function $Owner(d)$ returns the identifier of the processor on which d resides, and the function $Loc(d)$ returns the location of d in the local memory of the processor $Owner(d)$. These functions can be stored in a table on each processor's local memory. Two statements, LOAD and STORE, are introduced with the following semantics:

```
LOAD $d, t, pid$:
    IF $(Owner(d) = mypid)$ THEN
        IF $(pid = mypid)$ THEN $t \leftarrow$Loc(d)
        ELSE $S(d, mypid \rightarrow pid)$
    ELSEIF $(pid = mypid)$ THEN
        $R(d, mypid \leftarrow Owner(d))$
        $Move\ \ d, t$

STORE $d, f(t_1, \ldots, t_n)$:
    IF $(Owner(d) = mypid)$ THEN
        $t \leftarrow f(t_1, \ldots, t_n)$
        $Move\ \ t, Loc(d)$
```

where t and $t_i, 1 \leq i \leq n$, are local temporary memory locations, *mypid* is the identifier of the processor that executes the LOAD or the STORE statement, and *pid* is the identifier of the processor to which d is to be loaded. These two statements do not appear in the source program. They are used only by the compiler. With these two statements, a standard assignment statement $d = f(d_1, \ldots, d_n)$ is compiled to the following sequence:

```
LOAD $d_1, t_1, Owner(d)$
    ...
LOAD $d_n, t_n, Owner(d)$
STORE $d, f(t_1, \ldots, t_n)$
```

Here, $Owner(d)$ is said to be a consumer processor for $d_1, d_2, \ldots, d_n$.

Our implementation of the SPMD model is to have two modules on each processor, an *executor* and a *communicator*. The *executor* is responsible for the execution of a statement and the *communicator* is responsible for sending and receiving data. When the executor needs to send a data item to another processor, it gives the data item and destination to the communicator and continues execution. When the executor needs to receive a data item, it sends a request to the communicator and waits until the communicator has provided the requested data. More specifically, when the executor executes the instruction $S(d, P_i \rightarrow P_j)$, it is the communicator that actually sends the data. When the executor executes the instruction $R(d, P_j \leftarrow P_i)$, the communicator holds the executor until the data requested arrives, then the communicator gives the data to the executor and the executor continues its execution. Both the communicator and the executor have access to the functions *Owner* and *Loc*. In addition, the communicator also maintains a hash table for storing received data temporarily. For normal SPMD execution, it is enough for each data element to have only one *owner processor*. In order to achieve fault detection, we will replicate selected data items and modify the basic compilation rule. For this purpose, we introduce two modified communication mechanisms: *multicast* and *multireceive*. In the specification of these mechanisms given below, P_d is a processor identifier and PID is a set of processor identifiers.

```
$S(d, P_d \rightarrow PID)$:          $R(d, P_d \leftarrow PID)$:
For all $p \in PID$ Do        For all $p \in PID$ Do
    $S(d, P_d \rightarrow p)$             $R(d, P_d \leftarrow p)$
```

Multicast allows one processor to send a data to a set of processors and multireceive allows a processor to receive multiple copies of a data item from a set of processors. For multireceive instruction, the variable d is a vector that can hold multiple copies of the same data item. These multiple copies may be then compared for fault detection purposes. If $P_d \in PID$, then $S(d, P_d \rightarrow P_d)$ means that the executor of P_d transfers the value of d to the communicator of P_d and $R(d, P_d \leftarrow P_d)$ means that the communicator of P_d gets a value of d from the executor of P_d.

3 Fault Detection Through Data Duplication

Replicated execution of an assignment statement may be achieved during SPMD execution by replicating the data item at the left hand side of the statement on different processors. If a data item d is replicated, then $Owner(d)$ is a set of processors, and when a statement such as $d = f(d_1, \ldots, d_n)$ is encountered, all the processors in $Owner(d)$ execute the statement and compute their own values for d. These values may be compared to determine if an error has occurred in one of the owners of d. One of the following alternatives may be used for achieving the comparison:

1. *Check before use*: After a replicated data d is computed, all the processors in $Owner(d)$ continue their execution until the value of d is used in another statement. At that time, all the owners of d send their copies of d to the consumer processors and the communicators of the consumer processors compare these copies. This alternative adheres to the spirit of SPMD execution, where data is sent to the consumer only when it is needed.

2. *Check after definition*: Each processor in $Owner(d)$ sends the computed value of d to the other processors in $Owner(d)$ as soon as d is computed, thus allowing an immediate comparison of the computed values of d. This way, an error is detected as soon as it occurs.

If data is replicated on more than two processors, then it is possible to locate and mask faults. However, it is sufficient to duplicate data for fault detection. For this reason, we will assume in this paper that $|Owner(d)| \leq 2$ for any data item d. To detect transient faults, $|Owner(d)| = 2$ for each data d. To detect permanent faults, we require that for each processor P_i, there must be at least one data item d such that $P_i \in Owner(d)$ and $|Owner(d)| = 2$. However we do not need to duplicate every data item. In the following two sections, we will discuss, in some details, the above alternatives for fault detection.

3.1 Check Before Use Strategy

In this strategy, comparison is performed just before a duplicated data item is used. In order to describe the strategy, we need to modify the semantics of the LOAD statement to account for the fact that a data item, d, may be owned by two processors and may be used in a statement executed by two processors. Specifically, the following version of the LOAD statement loads one or two copies of d (depending upon $Owner(d)$) into a set, PID, of consumer processors:

```
LOAD $d, t, PID$:
IF ($Owner(d) = \{mypid\}$) THEN
  IF ($mypid \in PID$) THEN $t \leftarrow Lod(d)$
  $S(d, mypid \rightarrow PID - \{mypid\})$
ELSE
  IF ($mypid \in Owner(d)$) THEN
     $S(d, mypid \rightarrow PID)$
  IF ($mypid \in PID$) THEN
    $R\_C(d, mypid \leftarrow Owner(d))$
    Move $d, t$
```

In the above description, $(PID - \{mypid\}) = PID$ if $mypid$ is not in PID. Note that $t \leftarrow Loc(d)$ is executed only by a processor if that processor is the only owner of d and is a consumer of d. In all other cases, the loading process will involve the communicators of the owner and consumer processors. The instruction $R\_C()$ is a modified version of multireceive instruction that allows multiple values of the same data item to be received and compared. In any given statement where a duplicated data d is used, the *executor* of a consumer processor will execute the instruction $R\_C()$ when it loads d. This causes the executor to wait for the communicator's response. The *communicator* of the consumer processor will receive two copies of d. It then compares the two copies, and if they match, it gives one copy to the *executor*. The communicator acts as a server, always waiting for an event. The event is one of the following:

1. $S(d, P)$: the executor wants to send a data d to a set of processors P;

2. $R(d, P)$: the executor requests receiving the data d from a set of processors P;

3. $A(d)$: a copy of d has arrived from another processor.

The communicator maintains a hash table, $Hash$, to store the received data temporarily. Each entry of $Hash$ contains a pointer pt which points to a queue. The queue is implemented by a linked list and each entry in the queue is either a pair of values (for a duplicated data item) or a single value (for a non-duplicated data item). To illustrate the former case, assume that a data item x is duplicated on processors P_i and P_j. If at a time during the execution, processor P_k needs to use x, it receives two copies of x, one from P_i and the other from P_j (see Figure 1 (a)). However, since the execution is asynchronous, it is possible that one processor proceeds faster than the others. Suppose P_i is ahead of P_j in the execution and before P_j gets the chance to send its first copy of x, P_i recomputes x and reaches another statement that uses x. If this statement is also executed on P_k, then P_i will send x again to P_k. When the communicator of P_k receives the new

x from P_i, it enqueues it at the end of the queue (see Figure 1 (b)). Only after the first copy of x from P_j arrives, could the communicator of P_k give the first value of x to its executor. After using the first value of x, the communicator deletes the head of the queue. See Figure 1 (c) and (d).

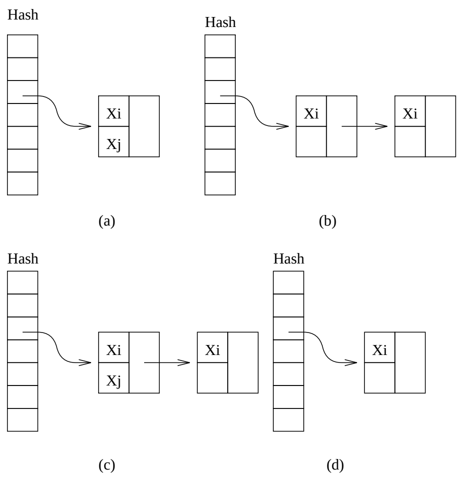


Figure 1: (a) The queue of P_k after receiving two copies of x; (b) P_i sends x twice; (c) The copy of the first x arrives from P_j: (d) After deleting the first x.

```
Algorithm 1. wait for all strategy:
Repeat
  Switch(wait(event))
  case S(d, P): $\forall p \in P, send(d, p)$
  case R(d, P): IF $(|Hash(d)| = |Owner(d)|)$ THEN
                    $delete(d, Hash)$
                    $return(d)$
                  ELSE $return(wait)$
  case A(d): $arrived = false$
             IF $(|Owner(d)| = 1)$ THEN
               $arrived = true$
             ELSEIF $(|Hash(d)| = 1)$ THEN
               IF $(d \neq Hash(d).pt)$ THEN
                 $broadcast(error)$
               ELSE $arrived = true$
             $insert(d, Hash)$
             IF $(arrived)$ THEN $Signal(d)$
until forever
```

Figure 2: The communicator algorithm for the check-before-use wait-for-all strategy

The algorithm for the communicator is given in Figure 2. In response to the event $S(d, P)$, the communicator simply sends the data to all processors in P. If the executor requests receiving a data d, the communicator checks if both copies of d have arrived by comparing the number of values $|Hash(d)|$ in the head of the queue pointed by $Hash(d).pt$ with the number of owner, $|Owner(d)|$. If all copies of the data have arrived, the communicator gives the data to the executor; otherwise it returns a special value *wait* which causes the executor to wait for an event. In the event of a data arrival, the communicator checks if both copies have arrived and if so it compares the two values and broadcasts an error message if the values are not the same. The communicator uses a local boolean variable *arrived* to remember if all copies of a data item have arrived. In the algorithm shown in Figure 2, a consumer of a data, d, has to wait for all the copies of d to arrive before resuming execution. We call this strategy a wait-for-all strategy. Four types of time overhead may be identified with the fault-detection strategy:

```
Algorithm 2: wait for one strategy:
Repeat
  Switch(wait(event))
  case S(d, P): $\forall p \in P, send(d, p)$
  case R(d, P): IF $(|Hash(d)| > 0)$ THEN
                    IF $(|Hash(d)| = |Owner(d)|)$
                      Then $delete(d, Hash)$
                    $return(d)$
                  ELSE $return(wait)$
  case A(d): IF $(|Hash(d)| = 1)$ THEN
               IF $(d \neq Hash(d).pt)$ THEN
                 $broadcast(error)$
             $insert(d, Hash)$
             $Signal(d)$
Until forever
```

Figure 3: The communicator algorithm for the check-before-use wait-for-one strategy

1. *communication overhead* caused by sending multiple copies of data items to consumers;
2. *duplication overhead* caused by duplicating the execution of some statements;
3. *comparison overhead* needed to compare multiple copies of a data item; and
4. *synchronization overhead* caused by the possible delay in the execution of a processor, P_i, waiting for multiple copies of a data item. Such a wait may delay other processors waiting for the data computed by P_i.

The synchronization overhead is the most serious since it may have a ripple effect on the execution of all the processors. This overhead may be eliminated if we resume the execution of a consumer processor after it receives one copy of the data. The communicator of a consumer processor still waits for two copies of a duplicated data and performs the comparison, but it gives the value to the executor once it receives the first copy of the data item. This is called *wait-for-one* strategy. The algorithm is given in Figure 3.

Another improvement is to eliminate duplicate comparisons. Whether we use a wait-for-all or a wait-for-one strategy, the comparison is performed by all the consumers of a duplicated data item. The comparison and communication overheads may be reduced by comparing the two copies of a duplicated data item only once rather than twice. With only one processor, P_i, performing the comparison for a data item, d, a fault in one of the owners of d may escape detection if P_i itself is faulty, thus reducing the fault coverage. Noting that, in effect, P_i is testing the two owners of d, the detection of up to t simultaneous faults, for some t, may be accomplished if duplicated data is chosen such that each processor is tested by at least t different processors during the execution of the program.

In order to eliminate duplicate comparisons, we distinguish the two owners of a duplicated data as *primary* and *secondary* owners. Non-duplicated data have only primary owners. We also define a *primary* consumer of a data item x as the *primary* owner of y, where y is the left hand side of a statement that uses x. With these definitions, we may modify the compilation rules such that if x is duplicated, only the primary consumer of x compares its two copies. For this purpose, the SPMD compilation should be modified such that the primary owner of x sends its copy of x to all its consumers, while the secondary owner of x (if x is duplicated) sends its copy of x only to its primary consumer. This guarantees correct operation whether x is duplicated or not, and whether x has one or two consumers. Specifically, (a) if x is duplicated, its two copies will be compared; (b) a primary consumer of x will receive all the copies of x; and (c) a secondary consumer of x will receive only one copy of x.

Finally, the *duplication* overhead may be eliminated if we add a spare processor, P_s, to the system and designate that spare as the primary owner of all the duplicated data. With this modification, the execution time of a program is almost equal to the execution time without introducing the additional spare and the fault detection capability. Since the executor of a consumer processor waits for only one copy of a data item, it will not be delayed, even if P_s is delayed. All the processors except P_s execute at their own pace while P_s roves around executing selective statements from each other processor. There is some overhead due to the extra communication caused by the duplicated data. However, this overhead is limited by the fact that the send instruction is non-blocking.

The motivation for the check-before-use strategy is to reduce the communication overhead by sending data to its consumers when the data is used, thus following the original SPMD compilation model. This strategy, however, delays the detection of an error in the computation of a data until this data is used. It also requires that each processor in the system owns a duplicated data that is used somewhere in the program, in order for that processor to be tested. The Check-after-definition strategy overcomes these shortcomings at the expense of larger synchronization overhead.

3.2 Check After Definition Strategy

In the strategy presented in this section, the comparison of independently computed values of a data item is performed as soon as these values are computed. Specifically, immediately after a duplicated data item, x, is computed by two processors, the two processors exchange copies of x and compare them. The same four types of overhead discussed in the previous section apply to the check-after-definition strategy, and similar variations of the basic strategy may be used to reduce that overhead. Although the principle discussed here can be applied for general case, due to space limitation, we only discuss a wait-for-one check-after-definition strategy in which a spare, P_s, is used and is designated as the primary owner of duplicated data. Only, the primary owner of a duplicated data item x namely P_s, compares the two copies of x and flags an error if the two copies are different. Specifically, if P_s and P_i are the primary and secondary owners of x, then whenever x is computed, P_i sends its copy of x to P_s immediately and P_s performs the comparison.

In addition to P_i sending a copy of x to P_s for the purpose of comparison, each consumer of x should receive a copy of x at the time x is used. However, each consumer should receive only one copy of x since no comparison will be done before use. In order to guarantee that each consumer will receive one copy of x whether or not x is duplicated, the LOAD instruction should be modified such that:

1. For a non-duplicated data item, the owner sends this item to all consumers, and
2. For a duplicated data item, x, the primary owner of x sends it to its primary consumer, and the secondary owner of x sends it to its secondary consumer (if any). Note that the primary owner of x is always P_s, and if x has two consumers, its primary consumer is also P_s.

For the proper exchange of data after definition and before use, the semantics of LOAD and STORE should be modified as follows:

```
LOAD $d, t, PID$:
IF $(mypid \in Owner(d))$ THEN
  IF $(mypid \in PID)$ THEN $t \leftarrow Loc(d)$
  ELSEIF $(|Owner(d)| = 1)$ THEN
    $S(d, mypid \rightarrow PID)$
  ELSE $S(d, mypid \rightarrow PID - \{P_s\})$
ELSEIF $(mypid \in PID)$ THEN
  $R(d, mypid \leftarrow (Owner(d) - \{P_s\}))$
  Move $d, t$
```

```
STORE $d, f(t_1, \ldots, t_n)$:
IF $(mypid \in Owner(d))$ THEN
  $t \leftarrow f(t_1, \ldots, t_n)$
  IF $(mypid \neq P_s)$ THEN Move $t, Loc(d)$
  IF $(|Owner(d)| > 1)$ THEN
    $S(t, mypid \rightarrow P_s)$
    IF $(mypid = P_s)$ THEN
      $R\_C(d, P_s \leftarrow Owner(d))$
```

Whenever a data d is used in the program, a LOAD statement will be executed and whenever a data d is defined in the program, a STORE statement will be executed. The algorithm for the communicator is shown in Figure 4. In this strategy, the communicator needs to store only one copy of any duplicated data since the spare will do the comparison.

Algorithm 3: check-after-definition strategy:

```
Repeat
  Switch(wait(event))
  case S(d, P): $\forall p \in P, send(d, p)$
  case R(d, P):
      IF ($|Hash(d)| > 0$) THEN
        IF (($mypid = s$)&($|Hash(d)| = 2$)) THEN
          $delete(d, Hash)$
        IF ($mypid \neq s$) THEN $delete(d, Hash)$
        $return(d)$
      ELSE $return(wait)$
  case A(d):
      IF (($mypid = s$)&($|Hash(d)| = 1$)) THEN
        IF ($d \neq Hash(d).pt$) THEN
          $broadcast(error)$
        $insert(d, Hash)$
        $Signal(d)$
until forever
```

Figure 4: The communicator algorithm for the check-after-definition strategy

With the spare processor being the primary owner of any duplicated data, duplicate execution of statements and comparison of duplicate data are performed on the spare processor. Thus, the execution of the non-spare processors does not suffer from any *duplication* or *comparison* overheads. The non-spare processors also do not suffer from any *synchronization* overhead since their executions do not depend on that of the spare. The execution of the program, however, is not terminated until the spare terminates. Thus, duplicate data has to be chosen carefully such that the spare finishes its execution no later than the last non-spare processor.

3.3 Fault Detection for Regular Loops

In general, the problems of selecting the data to be duplicated and the processors on which duplicated data is to reside are nontrivial. On the one hand, we would like to cover as many faults as possible and on the other hand, we do not want to cause too much overhead.

The reason for the wide acceptance of the SPMD model is that data distributions that lead to efficient SPMD execution may be found for parallel loops operating on arrays of data, and that such loops usually comprise the core of large scientific programs.

For a large class of regular loops (loops in which the data dependencies are independent of the loop indices), not all processors can be active all the time due to data dependencies. For these loops, we can duplicate the data in such a way that the idle processors are exploited for the purpose of fault-detection. In [8], we have developed some efficient fault-detection schemes that exploit the data dependencies of a regular loop. Following is an example of such a loop:

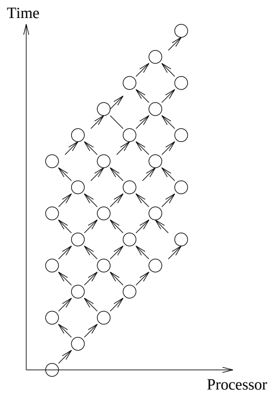


Figure 5: Execution pattern.

```
Integer A[1:M, 1:N]
Distribute A[i,j] = j

Do i = 1, M, 1
  Do j = 1, N, 1
    A[i,j] = A[i, j-1]*A[i-1, j+1]+C
  Enddo
Enddo
```

where the directive *Distribute A[i,j]=j* indicates that the data element $A[i,j]$ is owned by processor P_j. Under this data distribution, the execution pattern of the above loop with $M = 5$ and $N = 6$ can be plotted as in Figure 5, in which circles represent the executions of statement instances and arrows indicate data dependencies. As can be seen, processors have to be idle for some time slots during the execution of the loop. These idle time slots can be used for fault detection purposes. To detect permanent faults, we only need to duplicate the last statement instance of each processor. This can be achieved through an additional directive: *Distribute A[M,j] = (j-1), for j* > 1. See Figure 6 (a) for the duplicated execution pattern, in which the shaded circles represent those duplicated executions of statement instances. In order to detect transient fault, we have to duplicate all statement instances through the directive: *Distribute A[i, j] = (j-1), for j* > 1 (see figure 6 (b)).

4 Experiment Results

In the previous sections, different duplication strategies were described. The different strategies represent different tradeoffs between fault coverage and overheads. In this section, we present some results that empirically and experimentally estimate some of the overheads associated with duplicated execution.

In order to separate the different overheads associated with the duplication strategies, we first conducted an experiment to measure the communication and comparison overheads O_{cc}. The experiment, which was conducted on the Intel IPSC/2 multiprocessor, aimed at estimating the time overhead spent in the communication and comparison of duplicated

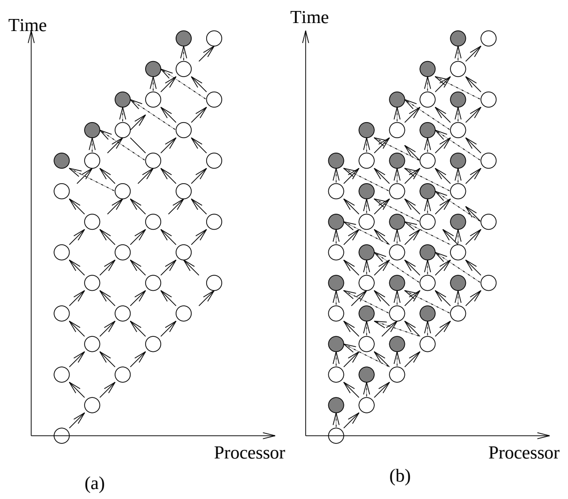


Figure 6: (a) Data duplication for permanent fault; (b) Data duplication for transient fault.

data. Specifically, let σ be a computation executing on a processor requiring data communication with other processors, and let τ and τ_r be the time to execute σ in a non-duplicated and a duplicated environments, respectively. That is, τ_r takes into account receiving and comparing two copies of any incoming data as well as sending two copies of outgoing data. We measured the effect of the granularity of σ on the overhead $O_{cc} = \frac{(\tau_r - \tau)}{\tau}$.

We considered a parallel multiplication of a $k \times k$ matrix, A, with a vector, X. The first $k/2$ rows of A and X are stored on a processor P_1 and the last $k/2$ rows on a processor P_2. The computation, C_1 (executing on P_1) first sends $X_1 \ldots X_{k/2}$ to P_2, receives $X_{k/2+1} \ldots X_k$ from P_2 and then performs the multiplication. The computation, C_2, executing on P_2 is similar in nature. In a duplicated environment, we executed C_1 on two processors, P_1 and P_3 and executed C_2 on two processors, P_2 and P_4, and we modified C_i, $i = 1, 2$, such that it sends duplicated data to P_i and P_{i+2}, and receives and compares duplicated data from P_i and P_{i+2}. Figure 7 shows the overhead resulting from such a duplication. As expected, the overhead decreases when the granularity, k, of the computation increases. Figure 7 also shows the overhead for a similar experiment where the computations are sorting and vector normalization.

In the above experiment, additional processors were provided to perform the duplicate computations. When no additional processors are provided, the execution of the primary computation may be delayed further. In Section 3.3, it was argued that, for some parallel loops with data dependencies, it is possible to duplicate execution when processors would be otherwise idle. This argument, however, was based on the assumption that the execution times of all instances are the same and that this time includes the time for data communication. In order to estimate the overhead in the absence of these simplifying assumptions,

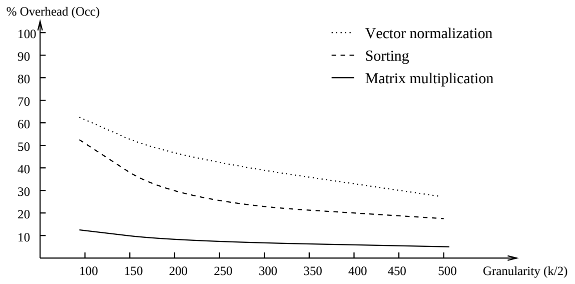


Figure 7: Overhead of Fault Detection on Intel Hypercube.

we have implemented a parallel SPMD interpreter to interpret intermediate language programs annotated with data ownership functions. The goal of this interpreter is to emulate the performance of the code generated by an actual SPMD compiler when such compiler is augmented with fault detection capabilities.

To emulate the execution of a program, G, on a multiprocessor, N copies of the interpreter are initiated on N processors and each is provided with a copy of G. The N interpreters, then, execute the different statements and communicate data among themselves following the SPMD owner computes execution rules. The interpreters are provided with information regarding the average times to execute different instructions and to send and receive messages. In order to account for variations in the execution time of memory and communication operations, the time for these operations are chosen randomly from a specified range. Each interpreter keeps a local simulated time and local times are aligned by time-stamping messages with the local time of their senders. In the results presented here, a communication instruction is about 10 times slower than a floating point instruction which includes memory fetch and store. Statistical variations of $\pm 25\%$ are assumed.

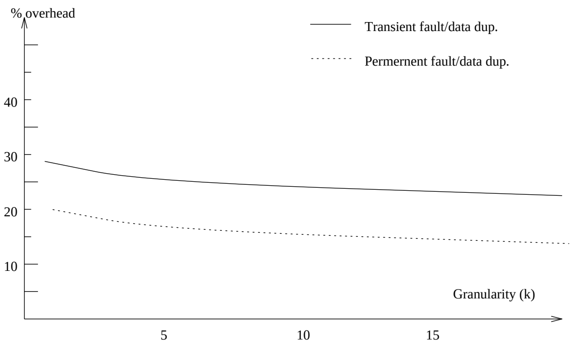
We have hand coded the parallel loop shown in Figure 8a using our intermediate language. The inner loop represents an instance with granularity k which communicates k data items and executes k statements. Figure 8b shows the overhead resulting from permanent and transient fault detection on ten processors. Although the duplicated instances are executed on processors that would be otherwise idle, the results show that the communication, comparison and synchronization overheads cause the duplicated execution to take longer than the non-duplicated one. However, the total overhead caused by adding the fault detection decreases with the granularity of the loop instance, k.

5 Conclusions

The main advantages of the fault-detection approach presented in this paper is that no specialized hardware is needed and the overhead introduced by the fault detection mechanism can be reduced through compiler techniques. By applying data analysis, the

```
Real C, A[0:M,0:N+1]
Distribute A[i,Block(k)]
Do i=1,M,1
   Do j=1,N-k, k
      Do l=j,j+k,1
         A[i,j]=A[i,l-1]*A[i-1,l+1]+C
      Enddo
   Enddo
Enddo
```

(a)



(b)

Figure 8: Overhead of Fault Detection with different Granularities using the Simulation Environment.

compiler can take advantage of spare capacity of the system for fault-detection purposes. Moreover, the degree of replication and the method used for replication may be chosen to be different for different parts of the program.

The fault model considered in this paper assumes that a faulty processor produces wrong results. Extending the fault detection technique to detect fail-stop processors is straight forward. Each time an executor on a processor wants to receive a data, the communicator starts a clock and waits for a predefined period of time. If the requested data does not arrive within the predefined period of time, the communicator signals error messages to other processors. This extension could be used to also detect network faults that cause a message to be sent to a wrong destination.

Unlike the software approaches that replicate a whole task onto more than one processors and compare the final results produced by those processor to achieve fault detection, our approach performs fault detection at the statement level, which results in better efficiency. Although, the problem of choosing the data to be replicated is not simple, it is possible to solve that problem for the important case of parallel loops operating on arrays of data.

References

[1] N. J. Alewine, S. K. Chen, C. C. Li, W. K. Fuchs, and W. M. Hwu, "Branch Recovery with Compiler-Assisted Multiple Instruction Retry," *The 22nd Annual International Symposium on Fault-Tolerant Computing,* pp. 66–73, 1992.

[2] V. Balasubramanian and P. Banerjee, "Compiler-Assisted Synthesis of Algorithm-Based Checking in Multiprocessors," *IEEE Trans. on Computers,* Vol. 39, pp. 436–446, April 1990.

[3] D. M. Blough and A. Nicolau, "Fault Tolerance in Super-Scalar and VLIW Processors," *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems,* pp. 193–200, 1992.

[4] M. A. Breuer and A. A. Ismaeel, "Roving Emulation as a Fault Detection Mechanism," *IEEE Trans. on Computers,* Vol. c-35, pp. 933–939, November 1986.

[5] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing,* Vol. 2, pp. 151–169, 1988.

[6] A. T. Dahbura, K. K. Sabnani, and W. J. Hery, "Spare Capacity as a Means of Fault Detection and Diagnosis in Multiprocessor Systems," *IEEE Trans. on Computers,* Vol. 38, pp. 881–891, June 1989.

[7] C. Gong, R. Gupta, and R. Melhem, "Compilation Techniques for Optimizing Communication in Distributed-Memory Systems," *Proc. International Conference on Parallel Processing,* 1993.

[8] C. Gong, R. Melhem, and R. Gupta, "Automatic Transformation of Programs for Fault Detection on Distributed-Memory Multiprocessors," *Technique Report RT-94-12,* CS Department, University of Pittsburgh, Jan. 1994.

[9] R. Gupta, "Compiler Optimizations for Distributed Memory Programs," *Proc. of the Scalable High Performance Computing Conference, Williamsburg, Virginia,* 1992.

[10] C. Koelbel, P. Mehrotra, and J. V. Rosendale, "Supporting Shared Data Structure on Distributed Memory Architectures," *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming,* pp. 177–186, 1990.

[11] M. J. Quinn and P. J. Hatcher, "Compiling SIMD Programs for MIMD Architectures," *Proceedings of International Conference on Computer Languages,* 1990.

[12] L. A. Shombert and D. P. Siewiorek, "Using Redundancy for Concurrent Testing and Repairing of Systolic Arrays," *The Seventeenth International Symposium on Fault-Tolerant Computing,* pp. 244–249, 1987.

[13] S. Tridandapani and A. K. Somani, "Efficient Utilization of Spare Capacity for Fault Detection and Location in Multiprocessor Systems," *International Symposium on Fault-Tolerant Computing,* pp. 440–47, 1992.