A Fault Tolerant Model for a Functional Language Parallel Machine*

Xinfeng Ye,
Department of Computer Science,
University of Auckland,
New Zealand.

John A. Keane,
Department of Computation,
UMIST,
Manchester, UK.

Abstract

This paper describes a fault tolerant model for a functional language parallel machine. The model is transparent to the user and ensures successful execution of programs in the presence of hardware failure. The model is based on data replication. It takes advantage of the properties of the functional languages. The recovery scheme can be carried out simultaneously on all processors, and occurs while "normal" program execution is in progress. Thus normal execution suffers less performance degradation than with other approaches.

1 Introduction

Flagship is a parallel, graph reduction machine designed to support the execution of functional languages [9]. This paper describes an operating system approach to achieve fault tolerance on the Flagship machine. Normally, an operating system achieves fault tolerance by checkpointing the states of the processors in the system. The state of a parallel/distributed system supporting the execution of conventional languages must be restored to a consistent status in the presence of system failure. Therefore, for a conventional system, a relatively complicated mechanism that must ensure cooperation in the checkpointing of the states of the processors, is needed to ensure that the information stored for coping with system failure does not contradict itself [8].

Functional languages have the property of referential transparency [6]. Therefore, the system supporting the execution of the functional programs do not have the concept of "state". As a consequence, checkpointing is simply replication of data in the system, and this can be carried out without the cooperation of different parts of the system.

The approach in this paper takes advantage of the properties of determinism and referential transparency of the functional languages. Therefore, unlike a conventional system, state restoration after a failure is not needed in a declarative system (i.e. the system supporting functional languages). The fault tolerant model in this paper is transparent to the users. The objective of the model is to allow the execution of the program to be carried out in the presence of the failure of part of the hardware system. The model is based on data replication. The data lost due to system failure are replaced by their backups during recovery. Data replication is carried out by the system automatically. The information concerning the replicated data is maintained by the system. Therefore, the execution of the programs does not need to synchronise with the backing up of the data. Hence, compared with some previous approaches, the system should suffer less performance degradation from the efforts made to achieve fault tolerance. The recovery scheme in this paper can be applied in a real-time fashion in the sense that the recovery can be carried out while the execution of the program is in progress. Also, the recovery can be carried out simultaneously on all the PEs.

2 Model of Computation

Graph reduction is a computational model for functional programs [6]. It can be summarised:

1. A program can be represented as a graph.
2. Program evaluation proceeds by a sequence of simple steps called reductions. Each reduction performs a transformation (rewrite) of the graph according to the rules defined by the program.
3. Reductions may take place in parallel.

Flagship is a set of closely-coupled processor-store (PE-store) pairs. The PE-Store pairs are fully connected by a network. There is a single route from one PE to another. Therefore, the network has the order-preserving property. A program is complied into a graph. The reduction of the graph is carried out by rewriting its nodes. The graph is distributed among the PE-Store pairs. Each PE can directly access that part of the store to which it is closely-coupled. Access to a non-local store is achieved by sending a request message. Each PE performs reduction on the sub-graph contained in its own local store. Although the store is physically distributed, it is globally addressable by any PE. The graph is distributed dynamically as evaluation proceeds, thus nodes migrate to load balance the PEs during computation. Each PE conceptually has four processing units [2]:

1. The **Memory Management Unit** allocates memory in the store coupled to the PE.

*This work is supported in part by Auckland University under grant A18/XXXXX/62090/F3414029.

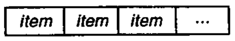

Figure 1: Packet Structure

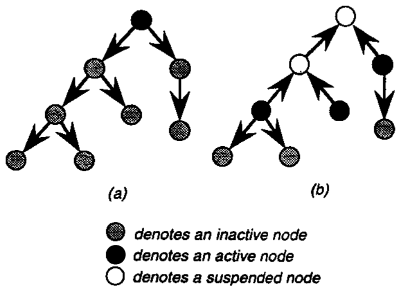

Figure 2: Graph Rewriting

2. The **Rewrite Unit** performs reduction.
3. The **Scheduler** manages the nodes which can be rewritten by the Rewrite Unit.
4. The **Network Interface** manages PE communication.

A node consists of a number of items (see Figure-1). Before a node can be rewritten, child nodes needed in the rewrite must be evaluated. A node forces evaluation by activating the child nodes. Once a parent node activates its children, the parent is suspended. The parent is re-activated when all values are returned from its children. When a node is activated, it becomes an active node. Only active nodes can activate their children and be rewritten. Active nodes are maintained by the Scheduler on each PE. At the beginning of execution, only the root of the graph is an active node. Thus, node activation starts from the root and propagates down.

Figure-2(a) is the graph at the beginning of execution. Figure-2(b) is the graph during execution. It can be seen that, starting from active nodes, it is possible to reach all nodes through pointers. During the rewrite, some nodes will no longer be referenced and will be garbage-collected.

3 Failure Model

The concern of this paper is with the toleration of permanent failures in some of the PE-Store pairs of the Flagship machine, i.e. the situation where the failed PE-Store pairs cannot recover from their failure. This kind of failure is termed *PE failure.* The system assumptions are:

- **Reliable communication**: It is guaranteed that the messages are delivered correctly.
- **Fail-stop PE**: All failures are detected immediately, and result in the halting of the failed PE.
- **Time-out**: Failure to respond within a certain time means a PE is treated as having failed.

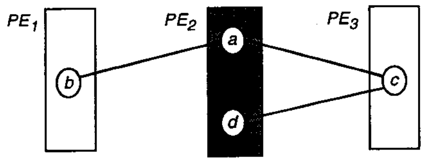

Figure 3: Recover Graph

To simplify the presentation, the situation that a PE fails when the system is recovering from a PE failure is not discussed. However, this situation can easily be coped with as explained in section 11.

4 Achieving Fault Tolerance

Fault tolerance is achieved via replication of the computational graph. The graph is distributed across several PEs. Hence, replication is achieved by copying the parts of the graph that reside on one PE to another PE. A node in the original graph is called a *primary* node, and its replication is termed a *backup* node. Backup nodes are passive during normal execution. They are only activated in the recovery if their corresponding primary nodes are in the failed PE.

When failure occurs, primary nodes in the failed PE are lost. However, lost nodes can be replaced by their backups which reside in another PE. In figure 3, a graph consisting of four nodes is spread across three PEs. If PE_2 fails, nodes a and d are lost. However, a and d can be replaced by their backups.

The properties of functional languages [6] allow *backup* nodes to be integrated into the graph. Due to the determinism property, nodes b and c return the same results when re-evaluated. Thus, nodes b and c can be adopted by the backup of a without being re-executed. Due to the referential transparency property, re-execution of nodes a and d does not make the system inconsistent. Thus, no state restoration is required when primary nodes (a and d) are replaced by their backups.

In order to find nodes lost in the PE failure, during the recovery, each node on the "healthy" PEs is checked to see if it has pointers to nodes in the failed PE. If so, the pointers are amended to point to the backups of the nodes in the failed PE.

5 Location of Replicated Data

A directional, logical ring consisting of all PEs can be formed. Each PE has both a unique *predecessor* and a *successor*. Each PE knows the *predecessor* and the *successor* of both itself and of every other PE. The data on a PE is replicated on its *successor* PE. Thus:

1. As each PE knows the failed PE's *successor*, it is easy to determine which PE holds the backups.
2. When a PE needs to replicate its data, it only involves the PE and its *successor*.

A bottleneck may occur during recovery because the *successor* of the failed PE has to inform other PEs of the addresses of the backups of lost nodes, so that pointers can be amended to point to the backups.

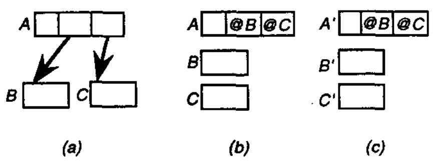

Figure 4: Storing the Replicated Packets

Another way of locating replicated data is to distribute the replicated data evenly across the other PEs. This would avoid the bottleneck described above. However, the number of new nodes which are generated during the rewrite of the graph is unknown. Therefore, in order to distribute the replicated data evenly, the location of the replicated data can only be determined during execution. In this case, in order to discover the location of the replicated data, a larger number of messages need to be sent during both normal computation and recovery [10]. Thus, the first approach is adopted here.

6 Storing Replicated Data

There are two ways of storing replicated data:

1. Store the backup nodes as identical to their primary ones. The advantage is efficiency in setting up backup nodes, since it is a simple copying process. However, the pointers in the backup nodes have to be changed when the backups replace the primary ones.
   Figure 4(b) shows how the graph in figure 4(a) is represented in the store (where *@X* denotes the address of node *X*). In figure 4(c) the backup node *A'* points to the primary nodes *B* and *C* rather than to the backup nodes *B'* and *C'*.
2. Make the pointers within the backup nodes point to other backup nodes when the backups are set up. Thus, the pointers in the backup nodes do not have to be changed during the recovery. In this case the example in figure 4(c) would have node *A'* pointing to the backup nodes *B'* and *C'*.

The extra complication in approach 2 significantly affects performance during normal computation [10]. Because the pointers in a node may point to a node on a different processor, backing up may involve more than one processor. In the example in figure 4(b), if *A* and *B* reside on different PE-store pairs then when *A* is backed up, the address of *B'* must be stored in *A'*. Hence the processor which holds *B'* must be consulted. Therefore, the first approach is adopted in this paper.

7 Data Replication

Before execution, the graph is backed up, i.e. all the nodes on each PE are copied to its *successor*. During execution, backups will be set up for newly created nodes, and some backups will be updated. The reason for updating the backups is to avoid the backup nodes being out-of-date.

For example, in figure 4, if node *B* is garbage-collected then the pointer *@B* in *A'* becomes invalid. Therefore, the backup nodes must be up-to-date enough so that they do not refer to nodes which no longer exist. Otherwise, the recovery cannot be carried out. The setting up and updating of the backup nodes can be carried out according to the following:

1. If a node is rewritten into another node, then the new node is backed up.
2. If the content of an item in a node is changed, then the backup of the node is updated.
3. When a node migrates from one PE to another PE, the backup of the node will be set up on the *successor* PE of the destination PE of the node.

8 Managing Replicated Data

During recovery, pointers to lost nodes are amended to point to the backups of the lost nodes. A *Mapping Table* on each PE, maintained by the Memory Management Unit (MMU), maps primary nodes to backups. When a primary node in a PE needs to be backed up, a backup request is sent to the *successor* of the PE. The request is parameterised by the primary node's address. The MMU of the *successor* allocates the store for the backup node and makes an entry in the Mapping Table to store the addresses of the primary node and the backup. When the backup of a primary node needs to be updated, a look-up is made on the Mapping Table to find out the address of the backup.

A computational graph does not contain any information concerning the mapping between primary and backup nodes. This information is maintained at the system level in the Mapping Tables. Hence, creation of a new node can occur without waiting for the setting up of the backup of the new node. Thus, the setting up of backup nodes does not interfere with "normal" execution, and there is little performance degradation due to fault-tolerant activity.

It can be seen that the setting up of the backups is slower than the execution of the program. This, however, does not affect the correctness of the model [12]. A Mapping Table needs to be maintained on each PE. Greenberg [2] points out that the MMU is idle for a large percentage of the total time. This suggests that the overall performance is unlikely to degrade as the Mapping Table can be maintained by the MMU.

9 The Recovery Scheme

During recovery the *successor* of the failed PE is called the *backup* PE. The PE which detects a PE failure is called the *initiator*. The PE which declares termination of the recovery process is called the *detector*. The aim of the scheme is to replace nodes which are lost due to PE failure. The parts of the graph which are not in the failed PE can carry on computation while the recovery is in progress.

At any time, each PE is in one of the six states:

- During normal execution, all PEs are in the **execution** state.
- When a PE fails, the *initiator* instructs all PEs except the failed PE to change from **execution** to **briefing**.
- When all PEs are in **briefing** state, the *initiator*

instructs all PEs to change to **initialisation**.

- All messages in the network are pushed to their destinations in the **initialisation** state. Following this, the *initiator* instructs all PEs to change to **recovery**.
- The recovery process starts once a PE enters the **recovery** state. Recovery processing starts from the active nodes and the backups whose primary nodes are in the failed PE. The recovery process will be propagated along the pointers to the whole graph. During the recovery, all references to nodes in the failed PE are changed to be references to their backups. Once recovery is complete, the *detector* instructs each PE to change to **informing** state.
- When all PEs are in **informing**, the *detector* instructs each PE to change to **backing up**. In this state the backups of all nodes are set up again.
- When backing-up is completed on all PEs, the *detector* instructs each PE to change to **execution** state, and the "normal" execution resumes.

The operations at each state are explained below.

1. When a PE is in the **execution** state, the program on the PE can be executed normally.
2. When the *initiator* instructs the PEs to enter **briefing**, it informs each PE of the identity of the failed PE. Thus, the *backup* PE becomes aware of its responsibility. To handle more one PE simultaneously discovering a PE failure, the PEs are ordered based on their identifiers. A PE gives up its *initiator* status once it receives a state-changing message from a more senior PE. Hence, the most senior PE that discovers failure becomes the *initiator.*
3. The **initialisation** stage involves network flushing, and forbidding node migration. Network flushing is needed because of node migration. Consider the following. Node a is being sent from PE_i to PE_j before a PE failure. If the transmission of a has not been completed by the end of the recovery, then a will not be processed by the recovery process. If a contains some references to the nodes which reside in the failed PE, then these references are invalid after the recovery. Network flushing can be carried out by letting each PE send special messages to all other PEs, and the *initiator* instructs the switching elements of the network which are connected to the failed PE to send the special messages on behalf of the failed PE. The flushing is completed when each PE receives the special messages from all other PEs. This technique takes advantage of the order-preserving feature of the network. During network flushing, the nodes bound for the failed PE are sent back to their original PEs. Node migration is forbidden during the initialisation stage because a migrating node may not reach its destination before the recovery is completed.
4. In the **recovery** stage, all pointers are checked such that if they point to a node in the failed PE, then a request is sent to the backup PE to find the new value for the pointer from the Mapping Table.

From the discussion in section 2, it can be seen that any part of the graph can be reached from the active nodes. However, the existence of the failed PE complicates the issue. This problem is illustrated in figure 5(a). It can be assumed that (a) B is an active node, (b) B, A and C are on different PEs, (c) the PE where B resides fails. Nodes A and C cannot be reached during the recovery, because B is in the

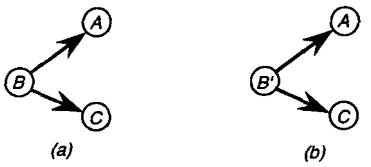

Figure 5: Problems with data on the failed PE

failed PE and cannot be accessed. Hence, the recovery should also be started from the backups of the nodes in the failed PE. From figure 5(b) it can be seen that A and C can be reached from B' which is the backup of B. Hence, the recovery is started from (a) the active nodes on each PE and (b) the backups of the nodes in the failed PE.

Once a PE enters the **recovery** state, nodes which have been processed by the recovery process can be migrated. Also, only nodes which have been processed by the recovery process can be rewritten by the Rewrite Unit. The advantage of this is that any node which is generated by the rewrite does not need to be processed by the recovery process. The reasons being (a) the pointers in a newly generated node can either be inherited from the node which is rewritten or allocated by the Memory Management Unit, and (b) only the nodes which have been processed by the recovery process can be rewritten. Therefore, a new node has no references to the nodes in the failed PE. Completion of recovery is detected by the algorithm in [11].

5. The **informing** state is introduced to make the PEs ready to receive messages from the subsequent stage of the recovery.
6. Since backups are identical to primary nodes, some backup nodes may contain pointers to the nodes in the failed PE. For example, in figure 4, if (a) A and B are on different PEs, and (b) B is in the failed PE, then A' contains a pointer to a node (i.e. B) in the failed PE. To tolerate future PE failure, in the **backing up** state, the current graph is replicated, and all old backups are discarded. The replication is carried out by sending the data on a PE to its *successor* PE. When each PE has completed the backing-up, the *detector* instructs each PE to enter the **execution** state. As a consequence, program execution returns to normal.

10 Comparison with Other Schemes

The schemes in [1] and [4] are designed to handle PE failure in a declarative system. In those schemes data are not lost during the PE failure because it is assumed that the store containing the data can still be accessed. Hence, data are not backed up. In contrast, this assumption does not hold for the Flagship closely-coupled PE-store pairs and thus the data in the failed PE cannot be retrieved. As a result, data are backed up in the model presented here.

In Grit's [3] and Lin's [5] schemes, in order to salvage the intermediate results, each node must keep some information about its ancestors or children. The scheme in this paper does not require nodes to keep any information about their ancestors or children. As a consequence, the execution of the program does not need to synchronise with the efforts made to achieve fault-tolerance. Hence, the scheme in this paper seems to be more efficient.

In Sharma's scheme [7] failure causes the re-execution of the computational graph. In contrast the scheme in this paper is based on node-by-node recovery. Therefore, instead of re-executing the whole branch of the computational graph whose root is in the failed PE, only the nodes in the failed PE need to be re-executed. This means that the failure does not affect the computation as much as Sharma's scheme.

11 Conclusions

In this paper a fault tolerant model for a parallel graph reduction machine designed to execute functional languages has been described. The model is transparent to the programmer. That is, the model hides the partial failure of the system from the programmer and guarantees the correct execution of the programs in the presence of partial system failure.

Recovery for functional languages is easier than for conventional languages. This is because functional languages are deterministic and referentially transparent. Thus, the repeated execution of the whole, or part of a functional program, always gives the same result. Hence, the backing-up of the functional programs can be carried out on any part of the system without the co-operation of other parts.

The backing-up of the nodes is "lazy" in the sense that the backup of each node is identical to the node. This strategy simplifies the setting up of the backup nodes. As a consequence, it minimises the effect on "normal" computation in the absence of failure.

Although the backing up of the nodes increases network traffic, the simulation and discussion in [2, 10], suggest that the backing up will not significantly reduce performance.

The mapping between a node and its backup is achieved at the system level via a Mapping Table. This means that primary nodes do not need to know the locations of their backups. Therefore, the rewrite of a primary node is not delayed by the setting up of its backup.

The recovery can be started on all PEs simultaneously. Therefore, the recovery can be carried out efficiently.

The recovery can be carried out while the execution of the program continues. Hence, it can be argued that the bottleneck described in section 5 may not seriously affect performance. This is because a PE can rewrite the nodes which can be rewritten while waiting for the addresses of the backup nodes to be looked up on the backup PE. However, this depends on the data dependency amongst the nodes. If most of the active nodes have references to the nodes which are in the failed PE, then the bottleneck remains. This is because only active nodes which have been processed by the recovery process can be rewritten. Thus, the PEs cannot rewrite the nodes containing references to nodes in the failed PE until the addresses of the backups of the nodes in the failed PE are returned from the backup PE.

The case that more than one PE fails simultaneously or a PE failure occurs while the machine is recovering from a failure, can easily be coped with by setting up more than one backup for the data in the system. This is because, by setting up multiple backups, it is always possible to obtain the backups of the data on a PE in the presence of multiple PE failure.

References

[1] Contessa A., An approach to fault tolerance and error recovery in a parallel graph reduction machine: MaRS - a case study, Computer Architecture News, 16(3), pp.25-32, ACM, 1988.

[2] Greenberg M.I., An investigation into architectures for a parallel packet reduction machine, Technical Report, UMCS-89-8-1, Department of Computer Science, Univ. of Manchester, 1989.

[3] Grit D.H., Towards fault tolerance in a distributed applicative multiprocessor, pp.272-277, Proc. 14th International Symp. Fault-Tolerant Computing, IEEE-CS, 1984.

[4] Hughes J.L.A., Error detection and correction techniques for dataflow systems, pp.318- 321, Proc. 13th International Symp. Fault-Tolerant Computing, IEEE-CS, 1983.

[5] Lin F.C.H and Keller R.M., Distributed recovery in applicative systems, pp.405-412, Proc. of 1986 International Conference on Parallel Processing, IEEE-CS, 1986.

[6] Peyton Jones S.L., The implementation of functional programming languages, Prentice-Hall International, 1987.

[7] Sharma M. and Fuchs W.K., Applicative architectures for fault- tolerant multiprocessors, pp.475-493, in Concurrent computations: algorithms, architecture and technology, Tewsbury S.K., Dickson B.W and Schwartz S.C., Plenum Press, 1988.

[8] Strom R.E. and Yemini S., Optimistic recovery in distributed systems, ACM Trans. Computer Systems 3(3), pp.204-226, 1985.

[9] Watson P. and Watson I., Evaluating functional programs on the Flagship machine, pp.80-97, Lecture Notes in Computer Science, Vol. 274, 1987

[10] Ye X., An investigation into the fault tolerant models for a parallel graph reduction machine, PhD Thesis, Department of Computer Science, University of Manchester, 1991.

[11] Ye X. and Keane J.A., Token scheme - an algorithm for distributed termination detection and its proof of correctness, pp.357-364, Information Processing 92, Volume I, J van Leeuwen Eds. North-Holland, 1992.

[12] Ye X., The Modelling and the Verification of a Fault-tolerant Functional System, pp.755-764, Proc. of the Sixteenth Australian Computer Science Conference, 1993