8-1-1986

Distributed Recovery in Applicative Systems

Frank C. H. Lin

Robert M. Keller
Harvey Mudd College

Distributed Recovery in Applicative Systems

Frank C.H. Lin *
ESL, Inc.
Sunnyvale, California 95051

Robert M. Keller*
Quintus Computer Systems, Inc.
Mountain View, California 94041

Abstract: Applicative systems are promising candidates for achieving high performance computing through aggregation of processors. This paper studies the fault recovery problems in a class of applicative systems. The concept of *functional checkpointing* is proposed as the nucleus of a distributed recovery mechanism. This entails incrementally building a resilient structure as the evaluation of an applicative program proceeds. A simple rollback algorithm is suggested to regenerate the corrupted structure by redoing the most effective functional checkpoints. Another algorithm, which attempts to recover intermediate results, is also presented. The parent of a faulty task reproduces a functional twin of the failed task. The regenerated task inherits all offspring of the faulty task so that partial results can be salvaged.

Keywords: fault tolerance, error recovery, distributed systems, applicative systems, data flow architecture, functional language.

1. Introduction

An important feature of a multiprocessor system, including applicative multiprocessing systems, is the ability to sustain partial system failures. By an *applicative system* in this paper, we mean a partitioned-memory system such as Rediflow [8, 9, 18] which coherently executes *applicative*, or *functional*, programs.

The evaluation of an applicative program generates an implicit call tree. The result of the root task is the answer of the program. Every task in the call tree represents a partial result which is used by its parent task to compute other partial results. Because the semantics of applicative of languages has no notion of destructive modification, a parent task is capable of regenerating all of its child tasks based upon the argument and function information.

Many fault-tolerance techniques for general multiprocessor systems have been proposed [1]. Some of these schemes can be adapted to applicative systems. However, applicative systems possess some interesting characteristics, e.g., *determinacy*, that merit distinct fault recovery considerations [6, 7, 14].

In this paper, fault tolerance issues in a class of applicative systems are studied. We assume that any task can be executed by any processor and that tasks are dynamically assigned to execution processors at run time. A single processor failure is also assumed. A processor is assumed to be either faulty or fault-free. A faulty processor must voluntarily declare itself faulty, or otherwise be identified as faulty by other processors.

It is further assumed that if a processor fails, it will no longer transmit any valid messages. This assumption can be enforced by commanding a faulty node to keep silent and not to respond to any inquiry. Alternatively, a faulty node may answer an inquiry with an invalid message. Several techniques are available for a processor to determine node malfunctioning. Parity checking on the system bus or resident memory, illegal instruction trap, protection violation, or a subsystem breakdown may trigger the CPU reporting a processor failure. Duplication of processors within a node, called "passive node diagnosis" [12], is also a common technique for building self-checking nodes.

It is assumed that a processor makes its best effort to communicate with a destination node. If the destination cannot be reached due to a network problem, the unreachable node is considered faulty. Problems with the interconnection network may be detected via coding or timeout mechanisms.

Our approach exploits the determinacy property of applicative programs. A distributed checkpointing scheme, *functional checkpointing*, is proposed in the next section. As the evaluation of an applicative program proceeds, a distributed resilient evaluation structure is incrementally established across the network of processors. Any single processor breakdown is salvaged by the implicit redundant path of the robust structure. A simple rollback recovery algorithm, which basically discards all partial results, is discussed in section 3. In section 4, another recovery algorithm, *splice recovery*, is proposed to salvage as many intermediate results as possible. Tasks which are equivalent to those trapped inside the faulty processor are generated to replace the failed tasks. Partial results produced by the failed tasks are inherited by the recovery tasks.

2. Functional Checkpoints

Checkpointing is familiar in the fault-tolerant computing literature [1]. In a uniprocessor system, checkpointing is normally performed by storing machine state on nonvolatile devices periodically. Such a periodical checkpointing technique has been extended to multiprocessor systems [3, 5, 7, 15]. The basic idea is to virtually stop all computational operations while periodic global checkpointing takes place.

* Work reported herein was supported by a grant from the IBM Corporation, while both authors were at the University of Utah.

0190-3918/86/0000/0405 $01.00 © 1986 IEEE

Periodic global checkpointing may not serve the best interests of fault tolerant applicative systems. For example, nonvolatile storage for storing system states may not be necessary, if recovery of a faulty processor is accomplished outside the node. Checkpoint information may be stored on one or more peer processors. Furthermore, periodic global synchronization among a large number of processors is potentially inefficient [2].

We propose a distributed checkpointing strategy for applicative systems. The approach attempts to exploit the determinacy property of applicative programs.

By a *functional* checkpoint, we mean a recovery point for a function application in an applicative system. A partial state of the system is stored so that recovery of the function is possible. The partial system state used in a functional checkpoint is related to a *single* function only. Normally, a functional checkpoint does not have enough information to recover an entire node, not to mention recovering a system. The sole purpose of the partial state is just to back up a function application.

The idea of functional checkpointing is to disseminate the responsibilities of recovering a faulty node to processors which have immediate relationship with the faulty node. Complete recovery is done by collective efforts from various associated processors to retrieve the corrupted tasks.

2.1 Determinacy

Determinacy, or *referential transparency*, is the characteristic of applicative programs which makes them attractive for distributed execution. A program is called determinate if an identical answer always results from any function invocation for given arguments. In other words, a functional program is free from side effects.

Determinacy suggests that an appropriate time for a functional checkpoint is when a parent task spawns a child function. A task packet is formed for the new function and then waits for execution. The packet contains *all* necessary information, either directly or indirectly accessible, to activate the child task. Furthermore, determinacy insures that different activations of the same task packet will always yield the same result. Thus, even if a task is aborted during computation, a new invocation will not be contaminated by its predecessors.

2.2 Checkpoint Properties

Periodic checkpointing is a synchronous operation whereas functional checkpointing is **asynchronous**. Each processor holds the privilege and responsibility of checkpointing its offspring tasks. A processor may opt to arrange the checkpoints in a partial order such that more efficient recovery can be implemented (section 3). Checkpoint coordination between processors is not necessary.

Functional checkpointing can be implemented **implicitly**. As a child task is spawned to a new node, the parent task may retain a copy of the task packet. This retained copy is all that the parent needs to regenerate the child task, should the node evaluating the child task fail. Therefore, functional checkpointing can be fully embedded in the evaluation process.

3. Rollback Recovery

Using functional checkpointing as a framework, a simple rollback recovery mechanism can be devised. An applicative call tree is mapped onto a set of processors. Each processor may have an arbitrary number of tasks. When a processor fails, the call tree may be broken into pieces. However, the piece that contains the root task is always capable of regenerating all severed pieces.

Suppose that an applicative program has been spawned into the call tree as shown in Figure 1. For ease of discussion, tasks Ai (i = 1, 2) are mapped onto processor A, tasks Bi are executed in processor B, etc. Suppose that processor B fails. Then tasks Bi are destroyed. The call tree is thus fragmented into three pieces: {A1,C1,C2,C3,D3}, {A2,D1,D2,C4}, and {D4,D5,A5}.

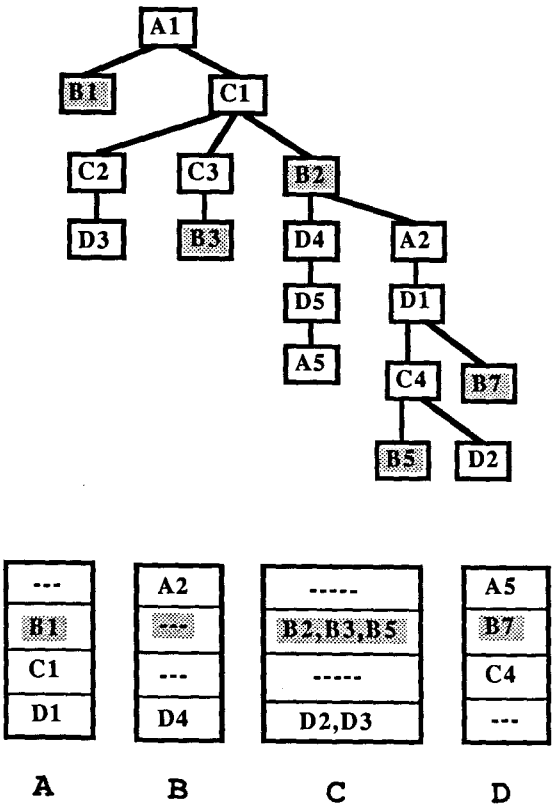

Figure 1: A call tree mapped onto processors A,B,C, and D, and corresponding distribution of checkpoints

Assuming the check point of an application is kept on the processor of its parent. Processor A contains the functional checkpoint for B1, processor C contains checkpoints for B2, B3 and B5, and processor D contains checkpoints for B7. To recover from the failure of B, the system needs to command processor A to respawn B1, and command processor C to regenerate B2 and B3. Task B2 will in turn generate new tasks which are equivalent to D4 and A2. Since an applicative program has no side effects, it does not require any *undo* operation, and hence there is no *domino effect* [13].

Note that task C4 holds the checkpointing data for B5. Processor C may regenerate B5 when B fails. However, the recovery of B5 is not fruitful because antecedent task A2 cannot report its result to B2. Reactivation of B5 only increases the system overhead. Therefore, an efficient way to salvage a group of genealogical dependents is to redo only the most ancient ancestor and ignore the rest.

3.1 Level Stamps

Genealogical dependencies among tasks can be monitored by a simple level numbering scheme. Assume that the root task carries a null level number, a task at level one will bear a unique one digit identification. Tasks in subsequent levels are stamped by appending one more digit to the number of their parents. The term "digit" is used here generically and is not limited to a specific radix representation.

Since each task is associated with a unique level stamp, it is obvious that ancestor-descendant relationships can be observed by comparing stamps. Note that a level stamp is not a time stamp. Its uniqueness is guaranteed by the program structure. Stamping of tasks can be fully asynchronous.

3.2 Recovery Scheme

Each processor maintains a table of linked lists. The Nth entry of the table contains all topmost checkpoints from the host processor to processor N. Referring to FIgure 1, for example, when processor C spawns task B2 to processor B, C compares the level stamp of B2 with all checkpoints in entry B. If B2 is a descendant of an existing functional checkpoint, C does nothing. Otherwise, processor C makes a checkpoint for B2 in entry B.

When processor C identifies the failure of processor B, C simply reissues all the checkpointed tasks found in entry B of the table. By doing so, processor C fulfills its responsibility of recovering B. Other processors take similar actions to recover their descendant tasks being trapped in B. The complete recovery of a faulty processor is a collective effort from processors which have checkpointed applications on the failed processor.

During task evaluations, a processor is required to abort a task if new arguments of the task cannot be obtained due to failures of other processors. A task is also aborted if the result of the task cannot be forwarded to the parent task. The aborted tasks and their descendants may be recollected during garbage collection operations.

3.3 Dynamic Allocation and Recovery

The possibility of discarding intermediate results without extensive undo operations is a property of applicative programs. However, the ability to recover by simply reissuing checkpointed tasks depends on the availability of a dynamic allocation strategy, such as the *gradient model* approach [10].

Recovering tasks in a static allocation environment requires manipulations of some linkage information. For example, tasks being allocated to a failed processor have to be reassigned to other processors. Descendants of the reassigned task have to modify their return addresses accordingly. Furthermore, the balanced state derived from the static allocation method may not be maintained easily after a processor fails.

Dynamic allocation does not distinguish between tasks generated for recovery and original tasks. All tasks are treated equally during load-balancing activities. The parent-child linking information is dynamically produced. Hence, there is no need to update these linkages when the task is reassigned.

3.4 Orphan Tasks

Rollback recovery inevitably leaves a few orphan tasks after some recovery has taken place, e.g., task D4 in Figure 1 becomes an orphan when processor B fails. The problem is that a task might not know whether it is an orphan without expenditure of a considerable amount of system resources.

Returns from orphan tasks are theoretically harmless since they are forwarded to a faulty processor and no side-effect can be induced. However, the partial results produced by orphan tasks are in fact correct answers of their associated functions. Failure of a node does not contaminate these incomplete answers; it just breaks the linkage among them. These partial results are usable if the regenerated parent task knows where to retrieve them, or if the orphan tasks know the new address to which to forward their answers. The desire to salvage partial results motivates the design of the following recovery scheme.

4. Splice Recovery

Applicative systems facilitate evaluation of a functional program by dynamically unfolding the underlying structure of the algorithm and disseminating parallel tasks to many processing nodes. At any instant, task distribution in a system represents a snapshot of the program structure. Generation of a task creates a new substructure and establishes a linkage between the parent and children. Return packets from a child task normally eliminate the children that are no longer needed.

The simple rollback scheme cuts off the branch or branches originating from a faulty node and regrows new branches. The method basically abandons all intermediate results computed by the orphan tasks. This section suggests a different approach, *splice recovery* , which attempts to retrieve all possible intermediate results.

4.1 Resilient Evaluation Structure

The splice approach is to continuously establish a *resilient evaluation* structure during program computations. A resilient structure is one containing redundant information which allows a system to rebuild the original structure after a failure has been identified. By rebuilding the structure, the system may salvage many partial results.

We have seen that when a processor fails, an applicative call tree may break into several pieces. The idea of splice recovery is to provide necessary bridging information such that broken pieces can be put together again. When a parent discovers the failure of a child task, the parent task generates a twin task of the faulty child. This twin task **inherits** all offspring of the faulty task with the help of the grandparent pointer.

A grandparent pointer of a task is a pointer from the task to its ancestor in the grandparent *processor*. For example, the grandparent pointer of task B3 in Figure 1 points to task A1, and task D4 has a grandparent pointer to C1 (Figure 2).

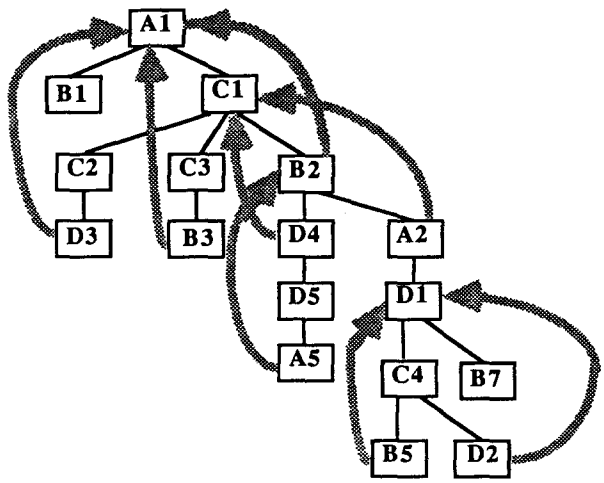

Figure 2: Grandparent pointers

Assuming as before that processor B fails, processor C may start recouping the loss of B2 as soon as C realizes that node B is dead. A twin task of B2, say B2', is created by the parent C1 to inherit tasks D4 and A2 (Figure 3). A full emulation of task B2 would require task B2' to possess physical binding information between B2 and D4, and between B2 and A2. Unfortunately, this information must be embedded not only inside the faulty node, but also within every descendant processor. Changing the return addresses of every descendant task at various sites could be very tedious.

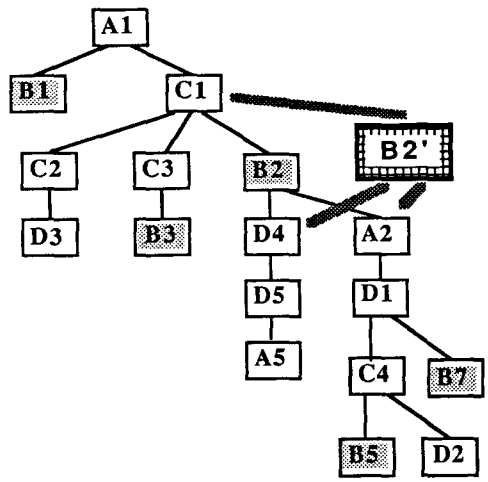

Figure 3: Task B2 is inherited by task B2'

Instead of fully emulating a faulty task, we opt to make B2' inherit descendant tasks of B2. Suppose that when D4 tries to return the evaluated answer to parent B2, it detects that node B is dead. The algorithm commands D4 to forward the result to grandparent C1. Processor C receives these unexpected partial answers from grandchildren and asserts that the parent of these grandchildren is faulty. Then, processor C forms the recovery task B2' by duplicating the task packet of B2.

If processor C has already reproduced B2' when the return from D4 arrives, task A simply forwards what it has received to step-child B2'. The role of a grandparent node in this recovery scheme is two-fold: it reproduces the dead task and it transports the orphan results to their step-parent when these returns become available. Having the grandparent relay partial results eliminates the problem of updating return addresses in every orphan task.

A recovered task, like any other, starts executing its function code as soon as it is committed to a physical processor. When it encounters a function call, it forms a task packet and spawns the child. However, offspring of a recovered task may or may not have been demanded by the preceding faulty task. Let P represent a faulty task, and C a child task of P. Let P' be the recovery task for P. C' is generated by P' and is the equivalent of C, as suggested in Figure 4.

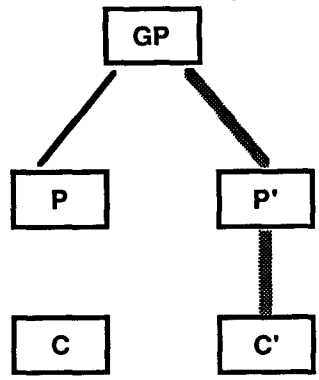

Figure 4: Tasks in splice recovery model

The relationship between child task C and its clone C' has the following possibilities: (Figure 5)

(1) C has never been invoked;
(2) C will never complete;
(3) C completes before P dies;
(4) C completes after P dies, but before P' is invoked;
(5) C completes after P' is invoked, but before C' is invoked;
(6) C completes after C' is invoked;
(7) C completes after C' has completed;
(8) C completes after P' has completed.

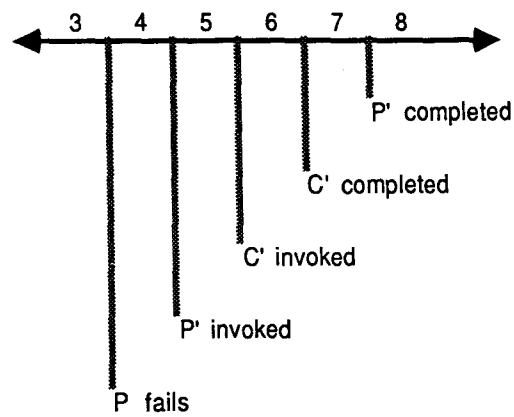

Figure 5: All possible orderings with respect to completion of C

Case 1 or 2: C has never been invoked or C will never complete. In either case, no result of C is produced. Task C is practically nonexistent and will be garbage collected. Only C' may produce an answer.

Case 3: C completes before P dies. Task C may have already finished the computation and returned the answer back to parent P before P breaks down. The result of task C is stored inside the parent P. When P fails, the system loses all partial results which have been saved in P. The recovery task P' must recalculate C by activating task C'.

Case 4 and 5: an old result comes before the new invocation. Task C finishes computation after the parent P dies. C sends its result to the grandparent task which transfers the result to the step-parent P'.

The difference between cases 4 and 5 is that in case 4, the grandparent has to reproduce P' first. When child task C' is executed by task P', P' will not spawn C' because the answer is already there. There is still only one result C' in the system.

Case 6: C completes after C' is invoked. Suppose that P' has already spawned C' when the result from C arrives at P'. Theoretically, the result from C and the would-be answer from C' are identical. Therefore, parent P' takes the answer from C and proceeds with the execution. The addition of C' may produce a duplicate answer to P'. Since they are identical, the second copy is simply ignored.

Case 7: C completes after C' has completed. This is the reciprocal situation of case 6. Note that due to the asynchrony nature of task evaluations, late invocation of an identical task may yield a result faster than the earlier invocation.

Case 8: old result arrives after everything is completed. The processor which contained P' may no longer recognize the arrived answer. The result is discarded.

4.2 Protocol for Splice Recovery The main idea of the splice recovery method is to build a resilient structure along with a program evaluation. The redundant information must be in place long before a recovery is initiated. This section describes the usage of these redundancies from the view of a single processor.

```
LOOP
  CASE received packet OF
  forward result:
      Interpret the level stamp.
      CASE level stamp OF
        child: Place data at the location indicated by the
               level stamp. If a task can be continued,
               resume the task.
        grandchild: Create a step-parent for the grandchild
                    if there isn't one already.
                    Transfer the result to its step-parent.
        others: Ignore the packet
      ENDCASE

  fetch data: If the location has been evaluated, forward the
              data. Otherwise, DEMAND_IT.

  task packet: Execute the task. DO each instruction
       If an unevaluated function encountered,
       DEMAND_IT.
       If cannot proceed, suspend the task.
       UNTIL completion. Send the result to the parent.
       If the parent is dead,
            notify the grandparent and
            send the result to the grandparent.

  error-detection: Find the topmost offspring of all
       branches, respawn all of these apply tasks.
       Establish transport mechanism for relaying
       partial results.
  ENDCASE
ENDLOOP.
```

The routine *DEMAND_IT* is the fundamental evaluating process of an applicative program. We elaborate the algorithm in the following form.

```
DEMAND_IT:
      Create a task packet.
      Level-stamp the task packet
      Attach parent and grandparent identifications
        to the task.
      Queue the task packet to load balancing manager.
      Functional checkpoint the packet.
End DEMAND_IT.
```

As a rule of thumb, if a processor receives a packet and cannot find a proper rule to handle it, the processor simply ignores the received message. Note that the overhead of splice recovery protocol is small. By using the level stamps as tags for a program structure, the apparent overhead for establishing a resilient structure is a physical identification of grandparent node which may be just an integer.

4.3 Correctness of the Recovery Scheme

The successful evaluation of an applicative program is signaled by the completion of the root task. A necessary condition for completing the root evaluation is to satisfactorily compute all immediate descendants of the root. This observation, when applied recursively, implies that all tasks must be evaluated correctly.

In order to guarantee a task can be evaluated, the task has to be generated in the first place. This means that every task is flawlessly reproducible even if some processor may fail during the evaluation. *Reproducibility* of tasks is the main criterion for a resilient applicative system.

4.3.1 Reproducibility

If the failed processor contains the root of a task tree, the regeneration of the root does not come naturally with recovery schemes. The user must restart the program, or a preevaluation functional checkpoint needs to be implemented.

One simple method to generate a preevaluation checkpoint is to create a super-root which acts as the parent processor of all user programs. When a user program is initiated, the super-root checkpoints the program so that a duplicate copy of the program can be found in the system should the root fail.

With this modification, every task in an applicative program has a parent. A parent task is capable of generating and regenerating any immediate child task as long as the parent is informed by some error detecting mechanism. This satisfies the reproducibility requirement of a correct recovery algorithm.

4.3.2 Residue Effects

Without loss of generality, evaluation of an applicative tree can be typified by scrutinizing the spawning process of a three-task sequence. Figure 6 shows the state transition diagram of spawning and reduction of task G. Task G spawns task P which subsequently spawns C. Note that states b and d are transient. The existence of transient states is a result of the dynamic load balancing method.

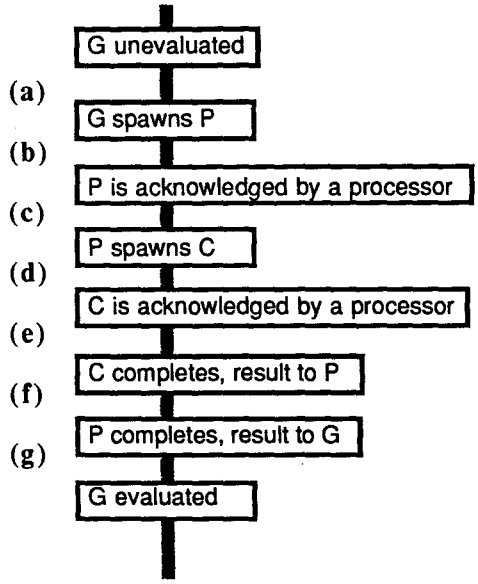

Figure 6: State transition diagram for evaluating G

The pointers being produced and reduced among tasks of each state are depicted in Figure 7. The pointer from P to its grandparent and the pointers to P from its grandchildren are omitted for clarity. Assume that task P fails during the evaluation of G, residue effects may affect any one of the related tasks at any stage of the state transition. A residue-free fault tolerant measure must assure that tasks G and C are not affected by the failure of P from state a through state g.

The failure of P obviously has no effect in state a. In state b, failure of the processor which absorbs P means that parent task G will not receive a positive acknowledge from task P. As a result, processor G times out and reissues a new task P. The system acts as if the first invocation of P did not take place.

In state c, task G receives an acknowledge from P and establishes a parent-to-child pointer to P. The new pointer may provide additional fault detection capability, but the impact of the failure remains similar to that of state b.

Residual effects, resulting from the failure of P, may happen at state d and successive states. Parent task G is left in the same situation as if it were in state b or c. However, there is a child task C lingering around the system. Task C may be stranded due to incomplete information from parent P. In this case, C commits suicide and the recovery from G is free from residual effects, or task C may complete the evaluation and try to return the result. C sends the result to G after failing to communicate with parent P. The case analysis in section 4.1 applies here.

State e has exactly the same recovery condition as state d, since the transient state d becomes state e as soon as task C finds an idle processor. The discussions on state d can be applied here and will not be repeated. State f is similar to state c as far as recovery is concerned.

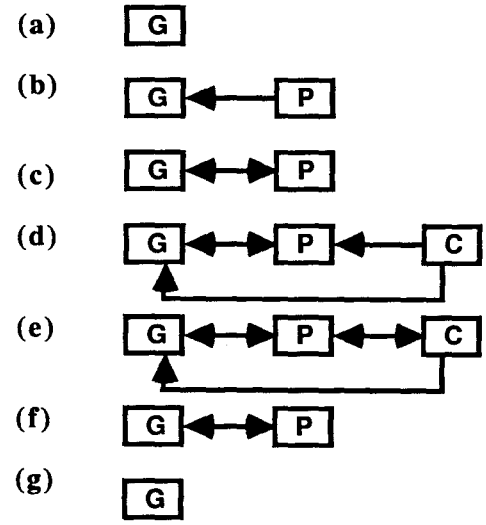

Figure 7: Available pointers among tasks

5. Discussion and Related Research

5.1 Robust Storage Structures

Using a resilient structure for fault-tolerant computing is not a new idea. Waldbaum [19], and Taylor, et al. [16, 17] proposed a robust storage structure to ensure data integrity in uniprocessor or shared memory machines. Item count, identifier field, and/or additional pointers are commonly added for error detection and recovery purposes. This paper extends the concept of resilient structure to a distributed applicative evaluation graph.

Conceptually, an evaluation structure is quite different from a storage structure. A storage structure is an object manipulated by programs, while an evaluation structure is a program itself. Furthermore, most techniques developed for resilient storage structure seem to be impractical in distributed systems. For example, item count of a linear list is a convenient way for checking broken links in a shared memory machine. To maintain a correct item count and verify it regularly in a network of processors would require significant traversing overhead.

5.2 Multiple Faults

Both the rollback and splicing recoveries use functional checkpoints to tolerate hardware failures. Although single node failure is assumed throughout the discussion, it is obvious that rollback recovery is not limited to tolerate only

single node failure. The difference between multiple faults and single fault in the rollback algorithm is the placement of the recovery border in the evaluation graph.

Splicing recovery can handle some combinations of multiple faults gracefully. For example, multiple failures on different branches of a structure do not disturb the recovery algorithm at all. Separate recoveries take place at different parts of the program in parallel. However, if both the parent and grandparent processors of a task fail simultaneously, the orphan task would be stranded. It is noted that the resilient structure concept can be further extended to include pointers to the great grandparent and beyond to tolerate multiple failures on one branch of the graph.

5.3 Hardware Redundancy

In a hardware redundant fault tolerant system, several redundant machines execute an identical program on replicated data objects. An applicative system can emulate hardware redundancy by simply replicating the task packets. Eventually, a task is executed by several processors at random times. The results are sent back to the originating node asynchronously. The originating node compares these results and selects a majority consensus as the correct answer.

A fundamental difference between applicative replicated task redundancy and pure hardware redundancy is that applicative systems execute redundant tasks asynchronously, while most hardware redundant systems employ synchronous operation. Asynchronous operations are subject to timing delays because a node has to wait for the return from slower processors. But a node does not have to wait for the slowest answer if it has received the identical results from the majority of replicated tasks. Replicating tasks provides a means of emulating hardware redundancy in applicative systems. The user may specify certain critical sections of a program for such a highly reliable operation.

5.4 Related Research

Fault tolerant problems in data-driven systems have been studied [6, 7, 11, 14]. Misunas proposed a triple modular redundancy implementation of a dataflow machine [4, 11]. Three complete copies of the program are stored in the memory. Copies of each instruction are carefully distributed so that each copy is executed by a different processor and utilizes different communication paths. Thus, the failure of any single block affects at most one copy of the program.

Hughes [7] described a variation of periodic checkpointing, where a host processor periodically stored the whole system state. Also discussed was a recovery technique, node-by-node correction, which used a control unit of the system as a monitoring device. Erroneous packets were recomputed and re-sent.

Srini [14] suggested a node reassignment algorithm for error recovery purposes. The algorithm depends on a global system memory for collecting and communicating recovery messages. The checkpointed node state is stored in the global memory.

Grit [6] proposed a structural recovery method where each node in the system is limited to spawning child tasks to its immediate neighbors. At system initialization time, a node receives a list of recovery sites for each of its immediate neighbors. When a node fails, a neighbor notifies the recovery site. The recovery node polls all possible parent and child nodes of the failed processor and tries to reconstruct the lost task.

6. Summary

This paper discusses the reliability aspect of applicative multiprocessor systems and suggests means for fail-soft treatment. The concept of functional checkpointing is proposed. Unlike conventional checkpoint schemes, functional checkpointing is concise, distributed and asynchronous. Two fault recovery techniques based on the notion of functional checkpointing are proposed. The thrust of these recovery models is to minimize the overhead while the system is in a normal, fault-free operation.

The simple rollback recovery method attempts to reconstruct the faulty section of the program structure by redoing the functions from the most efficient parent task or tasks. In other words, the recovery starts from the most recent functional checkpoints. The scheme is simple and has very little overhead in a normal operation. But, if a fault happens at a later stage of the evaluation, the rollback recovery may be costly.

The splice recovery scheme also uses the most recent functional checkpoints for error recovery as in the rollback method. In addition, the splicing scheme tries to salvage as much intermediate partial results as possible. The salvage is made possible by a backward grandparent linkage along with a program graph stamping mechanism. This approach enables the parent tasks of a faulty processor to regenerate the corrupted substructure of the program and splice the recovery results into the framework preceding the failure.

7. References

[1] T. Anderson and P.A. Lee. Fault Tolerance: Principles and Practice. Prentice Hall International, 1981.

[2] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. Computing Surveys 15(1):3-43, March, 1983.

[3] G. Barigazzi and L. Strigini. Application-transparent setting of recovery points. In Proc. of 13th Int'l Conf. on Fault-Tolerant Computing, pages 48-55. IEEE, June, 1983.

[4] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data-flow processor. In Proc. 2nd Annual Symposium on Computer Architecture. IEEE, 1974.

[5] M.J. Fischer, N.D. Griffeth and N.A. Lynch. Global states of a distributed system. IEEE Trans. on Software Engineering SE-8(3):198-202, May, 1982.

[6] D.H. Grit. Towards fault tolerance in a distributed applicative multiprocessor. In Proc. of the 14th Int'l Conf. on Fault Tolerant Computing, pages 272-277. IEEE, June, 1984.

[7] J.L.A. Hughes. Error detection and correction techniques for dataflow systems. In Proc. of the 13th Int'l Conf. on Fault-Tolerant Computing, pages 318-321. IEEE, June, 1983.

[8] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In AFIPS, pages 613-622. AFIPS, June, 1979.

[9] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. IEEE Computer 17(7):70-82, July, 1984.

[10] F.C.H. Lin and R.M. Keller. Gradient model: A demand-driven load balancing scheme. In Proc. 6th Int'l Conf. on Distributed Computing Systems, IEEE, Cambridge, Massachusetts, May, 1986.

[11] D.P. Misunas. Error detection and recovery in a data-flow computer. In Proc. 1976 Int'l Conf. on Parallel Processing, pages 123-131. IEEE, August, 1976.

[12] K.N. Oikonomou and R.Y. Kain. Abstractions for node level passive fault detection in distributed systems. IEEE Trans. on Computers c-32(6):543-550, June, 1983.

[13] B. Randell. System structure for software fault tolerance. IEEE Trans. on Software Engineering SE-1(2):220-232, June, 1975.

[14] V.P. Srini. Node reassignment in a dataflow system. In Proc. 4th Int'l Conf. on Distributed Computing Systems, pages 15-27. IEEE, San Francisco, CA., May, 1984.

[15] Y. Tamir and C.H. Sequin. Error recovery in multicomputers using global checkpoints. In Proc. of the 13th Int'l Conf. on Parallel Processing, pages 32-41. IEEE, August, 1984.

[16] D.J. Taylor, D.E. Morgan and J.P. Black. Redundancy in data structures: Improving software fault tolerance. IEEE Trans. on Software Engineering SE-6(6):585-594, November, 1980.

[17] D.J. Taylor, D.E. Morgan and J.P. Black. Redundancy in data structures: Some theoretical results. IEEE Trans. on Software Engineering SE-6(6):595-602, November, 1980.

[18] S.R. Vegdahl. A survey of proposed architectures for the execution of functional languages. IEEE Trans. on Computers C-33(12):1050-1071, December, 1984.

[19] G. Waldbaum. Audit programs - A proposal for improving system availability. Res. Rep. RC2811, IBM, February, 1970.