A global checkpointing model for error recovery

by KRISHNA KANT
Northwestern University
Evanston, Illinois

ABSTRACT

The paper proposes a new concept for providing software fault tolerance in concurrent systems. It combines the traditional global-checkpointing mechanism with the recovery-block concept in order to come up with an easily implementable error-recovery mechanism. This mechanism involves smaller overhead in case of moderate to high process interaction than the schemes considered in the past, which are based upon the idea of local checkpointing. A model for computing the optimum checkpointing interval is also presented. A particular distribution is hypothesized for the coverage of the recovery and the behavior of the model studied in detail for this case.
A. INTRODUCTION

Global checkpointing (GCP) is a popular technique for ensuring that an unexpected system failure does not result in the loss of valuable information. The state of the entire system is saved at periodic intervals (known as checkpoints) so that in the event of a failure the system can be brought to a consistent state simply by resuming it to its last checkpoint. There is an obvious tradeoff between the checkpointing overhead and the amount of computation lost as a result of failure. Many models have been proposed in the literature for computing the optimum checkpointing interval.\(^2\)\(^3\)

The fundamental assumption in such a GCP-based approach to fault tolerance is that the system failure is caused by temporary faults such as transient hardware faults, operator error, erroneous input data, timing problems resulting from an unusual combination of circumstances, and so on. In such cases, a rollback and retry procedure would most likely correct the problem. However, if we consider the failures resulting from bugs in the operating software, such a scheme does not suffice. In order to obtain fault tolerance against software bugs, it becomes necessary to incorporate functional redundancy in software. A well-known mechanism for doing this is the recovery-block concept.\(^7\)

As proposed by Randell, the recovery-block (RB) concept uses local checkpointing (LCP), that is, the state saving and restoration is done on a per process basis. If the processes constituting a concurrent program only compete for resources but do not interact otherwise, LCP is clearly the preferred strategy, since the rollback and retry will be limited to only the failed process. However, if we have a system of interacting processes, the erroneous information may propagate from one process to the other before it is detected. This leads to two complications: (1) it is no longer sufficient to maintain only the last checkpoint; in fact, very old checkpoints may need to be kept in order to handle occasional very long rollbacks. (2) the "exact" identification of the points to which the processes of the system need be rolled back becomes extremely complex and costly. (See Kant and Silberschatz\(^8\) and Kim\(^9\) for more on LCP-based recovery in concurrent programs.) In this paper, we explore the possibility of using GCP instead of LCP to reduce the cost and complexity of recovery in concurrent programs.

B. RECOVERY BLOCKS WITH GLOBAL CHECKPOINTING

For this, we simply remove checkpointing from Randell's RB concept; that is, when a process enters an RB, it does not establish any checkpoint. Any failures, including those resulting from the inability to pass the acceptance tests, are handled using the conventional GCP scheme. Thus the checkpoints will be established at periodic intervals and their location would be unrelated to the entry and exit points of the RBs. It is worth mentioning that this use of GCP is not at the level of entire computer system but only at the level of an individual program, which may consist of several interacting processes. In what follows, "system" refers to only one such program.

An important consideration in the use of global checkpoints for error recovery is the coverage, that is, the probability that the rollback will undo the erroneous interactions between the processes. Since the checkpoint interval is fixed in advance rather than deduced from process interaction history, we run the risk of doing too little or too much rollback for a given case. If the rollback is insufficient, then the system will fail again during retry. This situation can only be handled by increasing the rollback span for the next recovery attempt.

We assume that every RB has at least \(N\) alternates, where \(N\) is the number of previous checkpoints (PCPs) we are willing to maintain throughout the execution of the program. The first \(N - 1\) of these PCPs are consecutive (i.e. PCP(1) is the last established, PCP(2) is the one before that, etc.), and the last one, PCP(\(N\)), corresponds to the starting system state. The checkpoint to be established next during normal execution will be denoted by NCP. If a failure occurs before an NCP, the goal would obviously be to rollback and restart the system so that the execution proceeds successfully until the establishment of this NCP. If, during reexecution, a failure occurs prior to this NCP, then the rollback span must be increased and execution attempted again as explained above. It is clear that the system must keep track of the location of NCP during retry.

In order to keep track of which alternative of an RB is to be used for execution, we associate a counter ALTNO, initialized to 1 with each RB. We assume that all alternatives of an RB have been designed and coded independently and are approximately equally reliable. Thus we could use them cyclically, that is, if the current version of an RB (identified by ALTNO) is suspected of being faulty, we set ALTNO to \(\text{mod}(\text{ALTNO}, N) + 1\) for the next retry. The intended recovery algorithm for coping with a failure is as follows:

1. Set a checkpoint counter, \(K\), to 1. \(K\) will be used to keep track of how far back the system has been rolled back from the current state in terms of the number of previous checkpoints.
2. Rollback the system to \(K\)th previous checkpoint. Note that if \(K = N\), the system will be restarted from its initial

---

\(^2\) Kant and Silberschatz, op. cit.
\(^3\) Kim, Computer, February 1980.
\(^9\) Kim, Computer, February 1980.
state. For each RB that was entered since the establishment of this checkpoint until the point of failure, set ALTNO to \( \text{mod}(\text{ALTNO}, N) + 1 \). The purpose of this is to make sure that the failed part of the computation is retried using different alternatives. Discard any checkpoints that were established after PCP(\( K \)).

3. Restart the system and let it run until one of the following two things happens:
   - Execution proceeds successfully until the point where NCP should be established. In this case, all computation has been redone successfully and no further action is necessary until the next failure.
   - A failure occurs before the point for the establishment of NCP. Increment \( K \) and go to step 2.

C. ANALYSIS OF GCP MODEL

Our purpose here is to find the optimal checkpoint interval by minimizing the combined cost of checkpointing, recovery, and recomputation. However, an assumption is necessary before this cost could be defined meaningfully. Note that if a rollback to the starting state becomes necessary, an unbounded amount of computation may have to be redone after the rollback. If this recomputation is considered in computing the cost, the cost would be unbounded and no meaningful results can be obtained. Therefore, we consider the costs associated with only the first \( N - 1 \) rollbacks. If the probability of requiring rollback to the initial state is extremely small, the results would still be quite accurate and useful. We define the following quantities:

- \( \text{cps} \): Checkpointing span, that is, the number instructions executed between two successive checkpoints.
- \( \text{cpo} \): Checkpointing overhead, that is, the number of instructions (or its equivalent in terms of time overhead) required to establish a checkpoint. We assume that the use of a checkpoint in state restoration also involves the same amount of overhead.
- \( \text{PEj} \): Probability of successful execution until the next \( j \) checkpoints have been established given that the execution started in a consistent state.
- \( \text{PRj} \): Probability of successful recovery from failure when the rollback span is \( j \).
- \( \text{PERj} \): Probability of successful recovery and retry when a rollback span of \( j \) is used. It is easy to see that

\[
\text{PERj} = \text{PEj} \times \text{PRj}.
\]

- \( \text{RRC} \): Expected cost, in terms of number of instructions, of recovery and retry per failure.
- \( \text{cost} \): Fractional cost of checkpointing, recovery and retry.

Let \( \text{ncp} = \text{cps} + \text{cpo} \). The overhead of initial rollback and retry is \( \text{ncp} \). With probability \( (1 - \text{PERj}) \), a failure will occur during retry, thereby requiring more severe rollback. The overhead of the second rollback and retry will be exactly \( 2 \times \text{ncp} \) because PCP(1) must be reestablished during retry.

Continuing in this manner, and keeping in mind the assumption stated above, we come up with the following expression for \( \text{RRC} \):

\[
\text{RRC} = \text{ncp} \times \sum_{i=1}^{N-1} \frac{c^i}{i!} \prod_{j=0}^{i-1} (1 - \text{PERj})
\]

where \( \text{PER}_0 = 0 \).

Let \( \text{ier} \) be the instruction execution rate of the machine. Then the number of checkpoints established per second is \( (\text{ier}/\text{ncp}) \). Therefore,

- Primary failure rate \( (1 - \text{PER}) \times (\text{ier}/\text{ncp}) \)
- Primary checkpointing cost per second \( = \text{cpo} \times (\text{ier}/\text{ncp}) \)

Thus, the total cost of checkpointing, recovery, and retry (TC) is

\[
\text{TC} = [\text{cpo} + (1 - \text{PE}) \times \text{RRC}] \times (\text{ier}/\text{ncp})
\]

whence

\[
\text{cost} = \frac{\text{TC}}{\text{ier}} = \frac{\text{cpo}/\text{ncp} + (1 - \text{PE}) \times \sum_{i=1}^{N-1} \frac{c^i}{i!} \prod_{j=0}^{i-1} (1 - \text{PERj})}{\text{ier}/\text{ncp}}
\]

Now we can minimize cost and compute the optimum \( \text{cps} \). First, however, we must obtain an expression for \( \text{PER}_j \).

C.1 Calculation of \( \text{PER}_j \)

First note some general properties of \( \text{PE}_j \) and \( \text{PR}_j \). Assuming that the occurrence of software and hardware faults is uncorrelated, \( \text{PE}_j \) would be the product of \( \text{PES}_i \) and \( \text{PEH}_i \), which are the probabilities that the software and hardware failures do not occur until the establishment of next \( j \) checkpoints. Let \( P \) be the probability that no failure occurs during the computation performed between successive checkpoints. Then \( \text{PES}_i = P^{*i} \) assuming that the code used for establishing checkpoints and using them for state restoration is free of software faults. The probability \( P \) primarily depends upon three parameters: (a) the checkpoint span \( \text{cps} \), (b) the "quality" of the software, and (c) its error-detection capability. The last two of these are very difficult to quantify, although several metrics for them have been proposed in the literature. Here we shall simply work with a parameter \( ifp \), which is defined to be the probability of failure per instruction of the user code. Then

\[
P = (1 - ifp)^{\text{cps}}.
\]

Note that if fault corrections were taken into account, \( ifp \) would change as a function of the number of failures experienced. We could use Musa’s execution-time model to compute this change. Here we assume that \( ifp \) is a constant. Thus,

\[
\text{PES}_i = (1 - ifp)^{i \times \text{cps}}.
\]

\( \text{PEH}_i \) can be computed using the classical hardware-reliability
A Global Checkpointing Model for Error Recovery

To make our analysis independent of the execution speed of the machine, we shall work in terms of the number of instructions executed rather than the execution time. We assume a constant hazard rate (in the units per instruction) denoted by \( \lambda \). Then \( PEH_i \) must be an exponential function of the number of instructions executed. Therefore,

\[
PEH_i = \exp(-\lambda \cdot ncp \cdot j),
\]

and

\[
PE_j = PE_j \cdot PEH_i = (1 - ifp)^{i\cdot a} \cdot \exp(-\lambda \cdot (cps + cpo) \cdot j).
\]

Using the fact that \( ifp \) must be extremely small in any practical case, we have:

\[
PE_j = \exp(-j(a \cdot cps + b))
\]

where

\[
a = \lambda \cdot ifp, b = \lambda \cdot cpo.
\]

We assume that correct recovery can always be performed by rolling the system back to its starting state. Thus \( PR_j = 1 \) for \( j = N \). It is also clear that \( PR_0 = 0 \) because no recovery is possible without rollback. Furthermore, \( PR_j \) is expected to be a monotonically increasing function of \( j \). We hypothesize that it approaches 1 exponentially as a function of \( j \). Thus

\[
PR_j = 1 - \exp(-j/\alpha), 0 \leq j < N,
\]

where \( \alpha \) is a control parameter and must be nonnegative. It can be noted that smaller values of \( \alpha \) would be desirable. Obviously, \( \alpha \) depends upon the extent to which erroneous information can propagate before it is detected. This aspect of the system behavior is primarily controlled by two parameters, (a) the average number of instructions per RB and (b) the extent of interactions between the processes of the system.

A reasonable measure for the latter is the fraction of interprocess communication. We denote average RB size by \( rbs \) and the interprocess communication fraction by \( ipcf \).

Then \( \alpha \) can be written as a function of two arguments, that is, \( \alpha(rbs, ipcf) \). We claim the following:

1. \( \alpha(rbs, ipcf) \) is very large for \( rbs >> cps \) and \( 0 \leq ipcf < 1.0 \)
2. \( \alpha(rbs, ipcf) \) is very small for \( rbs << cps \) and \( ipcf < 1.0 \)
3. \( \alpha(rbs, ipcf) \) increases monotonically with \( rbs \) and \( ipcf \).

The first claim is based on the fact that no recovery is possible if the average RB size is much larger than the checkpoint span. The second claim is based on the fact that a single rollback would be sufficient for correct recovery if \( rbs \) is much smaller than \( cps \) and the interactions between processes are very infrequent. The justification for the third claim should be obvious. We assume that the contribution to \( \alpha \) from \( rbs \) is proportional to the fraction \( rbs/cps \). The contribution due to \( ipcf \), however, is expected to increase with \( ipcf \) because error propagation due to process interactions generally increases very fast with the level of interaction. We hypothesize that \( \alpha \) is of the following form:

\[
\alpha(rbs, ipcf) = C \cdot (rbs/cps) + D \cdot ipcf^a
\]

where \( C, D, \) and \( a \) are some positive constants that depend on various characteristics of the system under consideration, such as the quality of acceptance tests, the number of processes, and the complexity of interprocess communication. For example, if the processes show a very complex interaction pattern, the constant \( u \) is expected to be rather large, thereby making recovery difficult even when the parameter \( ipcf \) is fairly small.

Since the checkpoint span, \( cps \), is the only parameter of interest in the calculation of the optimum checkpoint interval, we can rewrite \( \alpha \) as follows:

\[
\alpha = c/cps + d
\]

where \( c = C \cdot rbs \) and \( d = D \cdot ipcf^a \). Let \( x = a \cdot cps + b \). Then \( \alpha = (a \cdot c - b \cdot d + d \cdot x)/(x - b) \). Also let \( y = \exp(-1/\alpha) \).

Then,

\[
PE_j = \exp(-jx); PR_j = 1 - y^j; PER_j = \exp(-j) (1 - y^j)
\]

\[
cost = cpo/ncp + (1 - \exp(-x)) \sum_{i=1}^{\infty} i \cdot \left( \prod_{j=0}^{i-1} (1 - \exp(-jx (1 - y^j))) \right)
\]

The expression for cost can be further simplified as

\[
cost = a \cdot cpo/(x + a \cdot cps - b) + (1 - \exp(-x)) \cdot f(x)
\]

where

\[
f(x) = 1 + (1 - (1 - y) \exp(-x)) \cdot [2 + (1 - (1 - y^2) \times \exp(-2x) \cdot [3 + \ldots]]
\]

C.2 Computation of the Optimal Checkpoint Interval

For \( N = 2 \), we only need to consider the first term in \( f(x) \). Since \( x << 1 \), we can approximate \( \exp(-x) \) by \((1 - x/2)^2\) and compute the value that minimizes cost. The result is:

\[
x = (1 - \beta/2) - [(1 - \beta/2)^2 - 2(\sqrt{a \cdot cpo - \beta})]^{1/2},
\]

where \( \beta = a \cdot cps - b \). The model for \( N > 2 \) can now be solved iteratively using the above value as the first approximation.

The results for the case \( N = 4 \) are shown in Figures 1–3. The parameters chosen are as follows:

- Hardware mean computation between failures \((1/\lambda) = 10^9\)
- Checkpointing overhead \((cpo) = 1,000 \) instructions per checkpoint
- \( C = 3 \), makes \( PR_j = 50\% \) at \( ipcf = 0 \) and \( RB\)-size = \( cps/2 \)
- \( D = 30 \) and \( u = 3 \), which makes \( d = 1.0 \) at \( ipcf = 0.32 \).

From the collection of the Computer History Museum (www.computerhistory.org)
The figures show optimum \(cps\) (CPS) and optimum cost (COST) as a function of \(ifp\), \(rbs\), and \(ipcf\). As expected, CPS decreases and COST increases with increasing \(ifp\). It is interesting to note that for \(ifp > \lambda\), that is, when the effect of hardware failures is negligible, COST is directly proportional and CPS is inversely proportional to the square root of \(ifp\). Since COST represents the fraction of instructions that are "useless," it must be \(< 1\) for a practical case. This means that for the above parameter values, the case \(ifp > 10^{-6}\) would be highly undesirable.

Both CPS and COST show interesting behavior with respect to \(c (= C*rbs)\). First consider the case \(ipcf = 0\). For \(c < < cps\), both COST and CPS are insensitive to increase in \(c\) but rise at an increasing rate for larger values. Although COST maintains its increasing trend, CPS peaks at around \(c = cps/2\) and then falls rapidly. Such behavior is expected because as RB size increases correct recovery becomes increasingly costly. The reason for the peak in CPS is that the probability of correct execution (or reexecution) decreases as \(cps\) increases; therefore, even though a larger \(cps\) would make correct recovery more likely, the success of correct reexecution becomes more doubtful. Thus we cannot keep increasing \(cps\) indefinitely to cope with larger \(rbs\) values. As the interaction between processes increases, the recovery cost rises at an increasing rate and smaller \(cps\) is required in order to hold COST down. This also explains why the peak in CPS becomes smaller and shifts to left as \(ipcf\) increases. The results for other values of \(N\) (not included here) are very similar.

D. COMPARISON OF LOCAL AND GLOBAL CHECKPOINTING

As mentioned before, the motivation for introducing GCP is to reduce the cost and complexity of performing backward recovery in concurrent programs. When comparing GCP against LCP, we must consider two aspects: (1) the complexity and overhead of the mechanism itself, and (2) the cost of all computation that had to be discarded or redone. It is clear that the recovery mechanism used by the GCP scheme is very simple and would result in very little time and space overhead. On the other hand, the LCP scheme requires maintaining both a very large number of local checkpoints for each process and the complete history of the interprocess interaction during which they were established. Moreover, the determination of rollback points requires either a search through this long history (as in Kant and Silberschatz) or an incremental update of information regarding it (as in Kim). Thus the GCP wins in this respect. The argument tilts in favor of LCP when we consider the second aspect, because LCP determines the correct rollback points on the basis of process interaction history rather than by a trial and error method. Thus LCP would be expected to usually involve less recomputation than GCP. However, it should be noted that even in LCP we do not know exactly where the problem lies (if we did, it should have already been removed!) and the algorithm for determining rollback points will usually require significantly more rollback than necessary.
It is clear from these observations that neither of the two approaches can be claimed to be always superior to the other. However, it can be argued that they are complementary if we consider the full range of interprocess interaction levels. Since all processes are rolled back to a common point in the GCP scheme, its overall cost would be higher than that for the LCP scheme for a system of processes that rarely interact. Furthermore, if the process interactions are rather infrequent, the overhead of maintaining and searching interaction history of the processes will be reasonably small. Thus LCP is an attractive scheme at low process-interaction levels. GCP appears to be a better scheme at moderate to high process-interaction levels, since the cost of maintaining and searching interaction history rises sharply with interaction level. A combination of the two schemes may also be used to advantage in certain situations. For example, we may carry out the determination
of rollback points for all processes assuming the LCP scheme but actually use GCP for rollbacks. In this case, the global checkpoint to which the system must be rolled back will be the one established prior to the earliest point in time to which some process must be rolled back according to our rollback-point determination algorithm.

E. CONCLUSIONS

In this paper, we have presented a global checkpointing scheme as an alternative to the local checkpointing inherent in the RB construct proposed by Randell. The basic purpose of introducing GCP is to simplify backward recovery in concurrent programs at the risk of discarding some computation unnecessarily. As noted above, this scheme is most suited for concurrent programs with moderate to high process-interaction levels. In this sense, the scheme complements the LCP scheme, which was designed primarily for sequential programs.

Several models for finding the optimum checkpoint interval have been proposed in the literature.2, 3 However, these models only consider recovery from transient failures where a rollback to the last checkpoint is sufficient for recovery. Our model deals with both transient hardware failures and the failures caused by software imperfections. (We do not consider permanent hardware failures, because the nature of recovery is very different in those cases). The model does take into account crucial system parameters such as recovery block size, the complexity and extent of process interaction, and so on. The software reliability was accounted for by a single parameter ifp, the instruction failure probability. An interesting extension would be to use some of the software quality metrics proposed in the literature to get a better characterization of software failure. Many other extensions are possible, such as the consideration of several classes of hardware and software faults, taking into account the effect of system load on failure rates, and so on.

REFERENCES


