Deriving Optimal Checkpoint Protocols for Distributed Shared Memory Architectures

Lorenzo Alvisi[1]* and Keith Marzullo[2]**

[1] Cornell University, Department of Computer Science, Ithaca NY
[2] University of California at San Diego, Department of Computer Science and Engineering, La Jolla CA

Abstract. Uncoordinated checkpointing is one technique used to build processes that can recover to a consistent state after crashing. This technique requires each process to periodically record its state in a checkpoint. Furthermore, the threads executing on each process log any nondeterministic action that they take following the latest checkpointed state. When a process crashes, a new process, initialized with the appropriate recorded local state, is created in its place. The new process restarts executing, and whenever one of its threads confronts a nondeterministic choice, the thread references the log in order to reproduce the same action performed before the crash. Thus, uncoordinated checkpointing implements an abstraction of a resilient process in which the crash of a process is translated into intermittent unavailability of that process.
We give a specification of the consistency property "no orphan threads" in the context of multithreaded processes running on a shared memory multiprocessor. We also give a definition of optimality for uncoordinated checkpointing protocols given a memory coherency protocol. We then use this specification to derive an existing uncoordinated checkpoint protocol and show that it is optimal. This protocol assumes that once a process crashes, no further processes crash until the first process completes recovery.

1 Introduction

Uncoordinated checkpointing is one technique used to build processes that can recover to a consistent state after crashing. This technique requires each process

* This author was supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant No. CCR-9003440, DARPA/NSF Grant No. CCR-9014363, NASA/DARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

** This author was supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, Contract N00140-87-C-8904 and by AFOSR grant number F49620-93-1-0242. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

to periodically record its state in a checkpoint. Furthermore, the threads executing on each process log any non-deterministic action they take since the latest checkpointed state. When a process crashes, a new process, initialized with the appropriate recorded local state, is created in its place. The new process restarts executing, and whenever one of its threads confronts a non-deterministic choice, the thread references the log in order to reproduce the same action performed before the crash. Thus, uncoordinated checkpointing implements an abstraction of a resilient process in which the crash of a process is translated into intermittent unavailability of that process.

Uncoordinated checkpointing protocols require that the state of a recovered process be consistent with the states of the other processes. This consistency requirement can be expressed in terms of *orphan threads*, which are surviving threads whose state is inconsistent with the recovered state of a crashed process. Thus, checkpointing protocols implement the consistency property "no orphan threads" in the context of shared memory multiprocessors.

In this paper, we focus on the specification and implementation of this property in distributed shared memory systems in which the memory coherency protocol is implemented in software. We do not consider the problem of how threads periodically record their states. While this is an important problem (and is the part of the protocol that is indicated by the protocol's name), its cost is only incurred periodically, while the cost of ensuring that there are no orphans is incurred constantly. Hence, the metrics we use to define optimality do not refer to the cost incurred by the periodic recording of the states of processes.

We use the specification to derive an existing uncoordinated checkpointing protocol [7] for a particular memory coherency protocol and show that the protocol is optimal with respect to our metrics. This protocol makes the (perhaps unrealistic) assumption that once a process crashes, no further processes crash until the first process completes recovery. In fact, it was this unrealistic assumption that prompted the research that led to this paper. We had recently examined the related problem of uncoordinated checkpointing in distributed message passing systems [2] and realized that the specification could be modified to apply to distributed shared memory systems and then used to derive the protocol presented in [7]. It is then relatively straightforward to apply our results in the message passing domain to modify this protocol into one that can tolerate any pattern of process crashes and recoveries.

2 System Model

We assume a set $\mathcal{T}$ of threads. Each thread is associated with one of a set $\mathcal{P}$ of processes, and all of the threads of a given process run on the same processor. The state of a thread is the contents of the variables of its process and the value of the thread's program counter. Hence, threads of the same process can communicate through variables they share. In addition, threads can communicate through a set $\mathcal{O}$ of shared *objects* accessible by all processes in the system. Threads can create new objects and can read and write existing objects. Specific memory consistency policies can be enforced to regulate concurrent accesses.

Since the state of an object may change as a result of create and write events, at any point during the execution an object is associated with a unique *version* v. Given an object o, we denote its current version by $o.v$, and the current value of its data by $o.data$.

For each write event $w(o)$ of object o, $w(o).t$, $w(o).o$, $w(o).v$, $w(o).data$ denote respectively the thread that executed w, the unique identifier of object o, the version of o that was created and the new value of $o.data$. Similarly, for each read event $r(o)$, we define the attributes $r(o).t$, $r(o).o$, $r(o).v$, $r(o).data$. In addition, we define the attribute $r(o).rsn$ to denote the order in which thread $r(o).t$ executed event $r(o)$ with respect to other read events that it executes: $r(o).rsn = \ell$ if $r(o)$ is the ℓ^{th} read event of an object executed by thread $r(o).t$. We call $r(o).rsn$ the *read sequence number* of event $r(o)$.

The execution of the system is represented by a *run*, which is an irreflexive partial ordering of events executed by threads. The events executed by the threads of a given process are totally ordered, but the events of threads of different processes may not be ordered [5]. For any two events e_i of thread i and e_j of thread j, we write $e_i \rightarrow e_j$ (read "e_i happens before e_j" or "e_j causally follows e_i") to denote the transitive closure of the relation defined by the following two conditions:

- threads i and j are of the same process and e_i was executed before e_j;
- e_i is the write event $w(o)$ and e_j is the read event $r(o)$ where $e_i.v = e_j.v$.

We say that the system is in a consistent state when, for all threads t_1, for every read event $r(o)$ of a version v of object o executed by t_1 there exists a thread t_2 that has previously executed a write event $w(o)$ such that $w(o).v = v$. Further restrictions on consistency may be imposed by the memory coherency protocol being used, but for our purposes such further restrictions can be ignored.

We assume that threads are piecewise deterministic with respect to object read events, in that the only non-determinism in a thread t arises from the fact that, for each object o, the version v of object o read by thread t is non-deterministic. Piecewise determinism can arise in one of two ways:

1. The thread scheduler for each process is nonpreemptive and deterministic. Otherwise, a thread t could be preempted by another thread t' which in turn could change the state of a variable, and hence the state of t. Since t' can preempt t at any point, the effect is a nondeterministic change in the state of t.
2. The thread scheduler is preemptive or nondeterministic, but all communication among threads, including threads of the same process, is through objects. Hence, while a thread can read and modify variables in its process, if these variables are shared they must be encapsulated as objects.

One can partition the execution of a thread into intervals. The beginning of the initial interval is defined by the initial state of the thread. Subsequent state intervals begin with the execution of a read operation on an object. Such an interval is called a *state interval.* Thus, given the first state of a state interval in the execution of thread t and the version of the object $r(o).v$ read by the event $r(o)$ that defines the beginning of the interval, the rest of the states in the interval are uniquely determined. The state interval that initiates with the event $r(o)$ is denoted $t[\ell]$ where ℓ, the *index* of $t[\ell]$, is equal to $r(o).rsn$. $t[0]$ represents the state interval that includes the states of t from its initial state to the state immediately before the execution of the first read event of an object.

We assume that processes fail independently according to the fail-stop model applied to processes [8]. When a process fails, then all the threads of that process also fail. We do not consider the case where a thread fails without its process failing, but the following formalism can be easily extended to include this possibility.

3 Derivation

A deterministic event can be defined by the pair of states $e_i = \langle s_i, s'_i \rangle$ such that when a thread executes e_i when its state is s_i, then the resulting state is s'_i. If e_i is nondeterministic, however, then the resulting state can be one of a set of possible states. For example, consider an asynchronous message passing system in which two processes p_1 and p_2 both have sent a message to p_3. A *receive* event executed by p_3 nondeterministically chooses one (or perhaps neither) of the two messages to deliver.

A *determinant* $\#e_i$ for a nondeterministic event e_i is information that, when given with an initial state s_i, uniquely determines a resulting state s'_i. For example, the determinant associated with the receipt of a message by p_3 above would identify which (if either) message is delivered. Determinants can be used during recovery to ensure that each recovering component deterministically reproduces the same sequence of states it entered before crashing. To do so, it is convenient for the determinant to identify the event it refers to. For instance, the determinant associated with the receipt of a message by p_3 above could also include the number of messages that were received by p_3 before executing e_i.

In a shared memory system, the nondeterministic events are the reading of objects. Hence, we associate a determinant $\#r(o)$ with each read event $r(o)$ that identifies the version of the object read. We use as the value of $\#r(o)$ the tuple $\langle r(o).o, r(o).v, r(o).t, r(o).rsn \rangle$, denoting that thread $r(o).t$ started a state interval with index $r(o).rsn$ by reading version $r(o).v$ of object $r(o).o$.

We define $r(o).depend$ to be the set of threads whose state either directly or indirectly reflects the state resulting from the execution of $r(o)$:

$$\begin{aligned} r(o).depend \stackrel{\text{def}}{=} \{ j \in \mathcal{T} \mid \exists \text{ an event } e \text{ executed by thread } j\text{:} \\ (e = r(o)) \vee (r(o) \rightarrow e)) \} \end{aligned}$$

A thread logs determinants in the address space of its process. We define $r(o).log$ to be the set of processes that have had logged $\#r(o)$ in their address spaces. Hence, if all the members of $r(o).log$ crash, then the determinant $\#r(o)$ is lost.

The function $P(t)$ denotes the process associated with thread t. We assume that this is a constant function—each thread is associated with a single process that does not change over time. The function $T(p)$ denotes the set of threads associated with process p. The functions $P^{\cup}(T)$ and $T^{\cup}(P)$ are the set-valued domain versions of $P(t)$ and $T(p)$ respectively:

$$P^{\cup}(\mathcal{T}) \stackrel{\text{def}}{=} \bigcup_{t \in \mathcal{T}} P(t) \qquad\qquad T^{\cup}(\mathcal{P}) \stackrel{\text{def}}{=} \bigcup_{p \in \mathcal{P}} T(p)$$

A thread is an orphan thread when its state depends on a determinant that is not logged. Let $\mathcal{C}$ denote a set of crashed processes. Then,

$$\begin{aligned} t \textit{ orphan thread of } \mathcal{C} \stackrel{\text{def}}{=} & \ (t \in \mathcal{T} - T^{\cup}(\mathcal{C})) \\ & \wedge \exists r(o) : ((t \in r(o).\mathit{depend}) \wedge (r(o).\mathit{log} \subseteq \mathcal{C})) \end{aligned} \tag{1}$$

Negating Equation 1 and quantifying over all t gives the following sufficient condition for there being no orphans created by the crash of a set of processes $\mathcal{C}$:

$$\forall r(o) : ((r(o).\mathit{log} \subseteq \mathcal{C}) \Rightarrow (r(o).\mathit{depend} \subseteq T^{\cup}(\mathcal{C}))) \tag{2}$$

Noting that $P^{\cup}(T^{\cup}(S)) = S$, it is straightforward to show that $A \subseteq B \equiv T^{\cup}(A) \subseteq T^{\cup}(B)$. Hence, we can replace the antecedent of Equation 2 with the equivalent expression $T^{\cup}(r(o).\mathit{log}) \subseteq T^{\cup}(\mathcal{C})$:

$$\forall r(o) : ((T^{\cup}(r(o).\mathit{log}) \subseteq T^{\cup}(\mathcal{C})) \Rightarrow (r(o).\mathit{depend} \subseteq T^{\cup}(\mathcal{C})))) \tag{3}$$

Universally quantifying Equation 3 over $\mathcal{C}$ yields the following sufficient property for there being no orphans:

$$\forall r(o) : (r(o).\mathit{depend} \subseteq T^{\cup}(r(o).\mathit{log})) \tag{4}$$

Equation 4 can be weakened while still ensuring that there are no orphans by constraining the number of failures. Assuming that no more than f processes can fail at any time, once $\#r(o)$ has been logged at more than f processes, no failures can cause $\#r(o)$ to be lost: we say that $\#r(o)$ has become *stable*. Hence, the following property also asserts that there can be no orphans:

$$\forall r(o) : ((|r(o).\mathit{log}| \leq f) \Rightarrow ((r(o).\mathit{depend} \subseteq T^{\cup}(r(o).\mathit{log})) \tag{5}$$

Protocols that satisfy Property 2—and hence Property 5—are commonly called *checkpoint protocols*. However, none of the properties presented above mentions checkpointing. Checkpointing is an (albeit important) optimization that expedites the recovery of a crashed process and allows for the garbage collection of logged determinants. Hence, while any implementation of a checkpoint protocol must address checkpointing, we ignore it in this paper, and assume that it can be added to any reasonable checkpoint protocol. Furthermore, we use the term *read logging protocol* instead of checkpoint protocol because it emphasizes what we believe to be the important trait of the technique.

4 Optimality

Memory coherency protocols of distributed shared memory systems exchange messages among processes in order to disseminate versions of objects to the sites that wish to read them. A read logging protocol is associated with a given memory coherency protocol, and any cost in performance imposed by the protocol is in addition to the memory coherency protocol. Hence, we define a read logging protocol as being optimal if it adds no cost over that imposed by the memory coherency protocol in terms of three metrics.

The first metric concerns rollbacks.

1. Consider a non-fault-tolerant memory coherency protocol Π run by a set of processes $\mathcal{N}$, and suppose that a set $\mathcal{C} \subseteq \mathcal{N}$ of processes crash. After the failure, the remaining threads have their state intact, but the system will not in general be in a consistent global state: some threads will have read objects that were written by threads that are now crashed. Let Π_c be protocol Π combined with a read logging protocol, and suppose again that the same set $\mathcal{C}$ of processes crashes at the same point in time. We say that c introduces r rollbacks if c requires r correct process to roll back the states of their threads in order to build a consistent state upon recovery. Clearly, a lower bound for r is 0, and several existing protocols show that this bound is tight. Hence, an optimal read logging protocol introduces no forced rollbacks.

The other two metrics are concerned with the messages sent by the memory coherency protocol. Consider a failure-free run r of a system that uses a given memory coherency protocol and a failure-free run r_c of the same system that uses the same memory coherency protocol but with a read logging protocol. We say that the r and r_c are *equivalent* if the sequence of application events executed by each thread in both runs are the same. Hence, r_c may contain different events related to memory coherency and read logging than r contains, but the application sees no difference when there are no failures.[3] With this definition in mind, we define the remaining two metrics:

2. Consider the interval of time between when a thread initiates a read operation on an object and the time that the thread obtains the data associated with the object. During each such interval the thread may need to deliver messages (in particular, those containing the value of the object). Let r be a run of a system with a memory coherency protocol and r_c be the equivalent run with the memory coherency protocol and a read logging protocol. If the read logging protocol is optimal, then the number of memory coherency messages sent in r and delivered in each such interval is the same as the number of memory coherency plus read logging messages sent in r_c and delivered in the equivalent interval. Hence, an optimal read logging protocol doesn't force a non-faulty thread to wait longer than it would were the read logging protocol not present.

[3] If there *are* failures, then the application can see a difference, since without the read logging protocol the crashed threads will not recover.

3. Consider equivalent failure-free runs r and r_c. If the read logging protocol that generated r_c is optimal, then the number of messages sent in r and r_c are the same. Hence, an optimal read logging protocol doesn't cause additional messages to be sent.

These two metrics combined imply that an optimal read logging protocol uses the memory coherency protocol's messages to carry any information it needs to ensure the ability to recover from crashes. This means that the messages will in general be larger in r_c than in r. One could instead send additional messages with a smaller increase in the size of messages, but in general there is no benefit in doing so. Of course, a message may become large enough that the underlying network will need to break it into separate packets thereby significantly increasing the overhead of the read logging protocol. In this case it may become advantageous to consider read logging protocols that send additional messages without introducing additional blocking.

In practice, there will be other costs associated with the use of a checkpointing protocol that are very hard to quantify. For example, the size of an object may increase because of the additional attributes associated with it. This in turn could lead to a different memory layout and therefore a different memory reference pattern. Thus, a protocol that is optimal with respect to the above metrics may not perform better than a non-optimal protocol that is more careful with its impact on memory layout.

An example of a read logging protocol that meets the above definition of optimality with respect to the Midway memory coherency protocol is given in [7]. This protocol is described in detail in the next section.

5 The DiSOM Protocol

In this section, we illustrate the formalism developed in Section 3 by applying it to a particular memory coherency protocol (*entry consistency*) with the assumption that $f = 1$. We then derive an existing protocol [7] used in the DiSOM system [4]. The entry consistency memory model was first introduced in the Midway system [3]. In this model, programmers is given the responsibility of explicitly structuring their applications in order to ensure that processes view a sequentially consistent memory [6]. In exchange, the weaker consistency requirement imposed by the model allows for good performance. The features of the entry consistency model can be summarized as follows:

- All access to shared objects must be enclosed between explicit *acquire* and *release* commands.
- An object can be acquired for either reading or writing. In the latter case, the thread that acquires the object is also allowed to read it.
- Whenever a thread releases an object it acquired for writing, it produces a new version of the object.
- A thread that executes an *acquire* command blocks until the command completes successfully.

– The model supports a multiple-reader, single-writer policy.

Hence, the versions of an object are totally ordered. DiSOM defines the notion of object ownership: a process p is the *owner* of an object o if the last version of object o has been produced by one of the threads running on p.

The system assumptions made in Section 2 are also made in [7]: threads are piecewise deterministic and processes fail in a fail-stop manner. The failure of a process can lead to the loss of a suffix of the versions of objects. The correctness condition for recovery given in [7] is:

> A system state is *consistent* if all threads, holding objects, hold the last versions of those objects and no thread has acquired a version of an object that was lost due to a failure.

Protocol Derivation. Since an object can be read only when it is acquired, and it can be read both when it is acquired for read and for write, recording the determinant of the read events is equivalent to recording the determinants of the *acquire* events. Hence, Property 5 becomes:

$$\forall o : ((|a(o).log| \leq 1) \Rightarrow ((a(o).depend \subseteq T^{\cup}(a(o).log) \qquad (6)$$

where *acquire* events have substituted read events. The attributes of an *acquire* event $a(o)$ are the same as a read event: $a(o).o$ denotes the object, $a(o).t$ denotes the thread acquiring o, $a(o).v$ denotes the version of the acquired object, $a(o).data$ denotes the data of the acquired object, and $a(o).asn$ denotes the acquire sequence number (analogous to the read sequence number for a read operation).

A version v of an object o can be represented by the pair $\langle tid, lt\rangle$ where $tid[lt]$ is the state interval in which thread tid executed the *release-write* operation that created version v.[4]. This is a unique version for o because, if tid were to create another version of o, it would first have to re-acquire o and perforce start a new state interval. Following [7], we call the pair $\langle tid, lt\rangle$ the *execution point* ep_{prd} of thread tid when it created v. We denote this execution point as ep_{prd} for this version of o.

Analogous to ep_{prd}, one can define an execution point ep_{acq} for each *acquire* operation $a(o)$. Hence, $ep_{acq} = \langle a(o).t, a(o).asn\rangle$.

The determinant $\#a(o)$ is the tuple $\langle a(o).o, a(o).v, a(o).t, a(o).asn\rangle$. This can be rewritten more compactly as $\langle a(o).o, ep_{prd}, ep_{acq}\rangle$ where $ep_{prd} = a(o).v$.

Consider a thread t' that issues an acquire request $a(o)$, and suppose that the last version of o has been produced by thread t''. We implement Property 6 as follows:

1. If $P(t') \neq P(t'')$, then thread t' piggybacks ep_{acq} on the request $a(o)$ directed to t''. When it is ready to process the request, t'' combines the piggybacked value of ep_{acq} with the values of $a(o)$.o and ep_{prd} to build the determinant

[4] Recall that a state interval is denoted by a thread identifier tid and an index ℓ (Section 2)

$\#a(o)$. The determinant is then logged in the address space of process $P(t'')$, that becomes a member of $a(o).log$. Notice that when t'' is ready to respond to the request issued by thread t', $a(o).log$ contains only process $P(t'')$. In order to satisfy Property 6, t'' piggybacks ep_{prd} to its response. As soon as it receives ep_{prd}, and before processing the rest of the response, thread t' builds the determinant $\#a(o)$ and logs it in the address space of process $P(t')$. Since now $|a(o).log| > 1$, $\#a(o)$ has become stable and does not need to be further piggybacked.
2. If $P(t') = P(t'')$, then by the time thread t' has obtained from t'' the last version of object o, the determinant $\#a(o)$ is logged only in the address space of process $P(t')$. To implement Property 6, t' needs then to piggyback $\#a(o)$ on the next message used by the memory coherency protocol.

Since the protocol satisfies Property 6, no orphan threads are created. Furthermore, no additional messages are sent and no additional blocking is introduced. Hence, this protocol is optimal with respect to the entry consistency protocol presented in the paper.

The logged information allows the threads of a process that is recovering after a crash to reacquire the same object versions that it acquired during the execution preceding the crash. When a process p starts recovering after crashing, it collects from the surviving processes in the system those object versions that were accessed by any of its threads before the crash. At the same time, p collects the determinants of all the acquire events performed by its threads. Whenever a recovering thread of p issues an *acquire* request for an object o, the recovery protocol responds by providing the thread with the version of o recorded in the appropriate determinant. Notice that since threads are piecewise deterministic, this procedure ensures that during recovery they will regenerate the same versions of objects generated before p crashed. After recovery is complete, all object versions acquired by surviving processes will have been regenerated, and the system will be in a consistent state.

6 Discussion

In [1, 2] we have presented a derivation of message logging protocols based on the specification "no orphan processes". The derivation and resulting protocols are different from what we present in this paper because the system model is considerably different. However, much of the work that is presented in [2] can be applied to the distributed shared memory model presented here. In particular, we have developed a set of optimal read logging protocols that use the techniques presented in [1] to tolerate any number of concurrent failures.

We have not yet derived read logging protocols for memory coherency protocols other than entry consistency. However, we believe that it should be possible to apply our approach to other memory coherency models. Any DSM system that uses software protocols for memory coherency must, at some point, be able to transfer a segment of memory that was written by one processor into the address space of another processor. Because this data must be sent in a message, we can satisfy Property 5 by piggyback on this message the non-stable determinants of any previous read operations. Doing so should lead to optimal read logging protocols, independent of the specifics of the memory coherency protocol.

Acknowledgements. This research benefited greatly from discussions we had with Brian Bershad, Paolo Guedes and Fred Schneider.

References

1. Lorenzo Alvisi and Keith Marzullo. Optimal Message Logging Protocols. Cornell University Department of Computer Science Technical Report TR 94-1457, September 1994.
2. Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. In *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*. IEEE Computer Society, May 1995.
3. B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Proceedings of the 93 COMPCON Conference*, pages 528–537. IEEE, February 1993.
4. P. Guedes and M. Castro. Distributed shared object memory. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 142–149, October 1993.
5. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
6. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.
7. N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the Thirteenth Symposium on Principles of Distributed Computing*. ACM SIGACT/SIGOPS, August 1994.
8. Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.