An architecture for tolerating processor failures in shared-memory multiprocessors

Michel Banâtre, Alain Gefflaut, Philippe Joubert, Peter Lee, Christine Morin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Architecture For Tolerating Processor Failures In Shared-Memory Multiprocessors

Michel Banâtre, Alain Gefflaut, Philippe Joubert,

Pete Lee, Christine Morin

N° 1965

mars 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

INRIA
RENNES

An Architecture For Tolerating Processor Failures In Shared-Memory Multiprocessors

Michel Banâtre*, Alain Gefflaut*, Philippe Joubert*,
Pete Lee**, Christine Morin*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet LSP

Rapport de recherche n ° 1965 — mars 1993 — 35 pages

Abstract: In this paper, we focus on the problem of recovering processor failures in shared memory multiprocessors. We propose an architecture designed for transparently tolerating processor failures. The Recoverable Shared Memory (RSM) is the main component of this architecture which provides a hardware supported backward error recovery mechanism. This technique copes with standard caches and cache coherence protocols and avoids rollback propagation.

The performance of the architecture during normal execution is evaluated and compared with that of existing fault tolerant shared memory multiprocessors. The performance study has been conducted by simulation using address traces collected from real parallel applications.

Key-words: shared memory multiprocessor, fault tolerance, stable storage, backward error recovery.

(Résumé : tsvp)

This work has been partly supported by the FASST Esprit project. This paper will also appear in a research report from the department of Computing Science of the University of Newcastle.

*{Michel.Banatre}{Alain.Gefflaut}{Philippe.Joubert}{Christine.Morin}@irisa.fr
**Department of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, UK. Email : P.A.Lee@newcastle.ac.uk

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Proposition d'une architecture à mémoire partagée tolérant les défaillances des processeurs

Résumé : Nous nous intéressons dans ce rapport au problème de la récupération des défaillances des processeurs dans une architecture multiprocesseur à mémoire partagée. L'architecture offre un mécanisme matériel de récupération arrière fondé sur les propriétés de la mémoire partagée récupérable (RSM). Cette technique s'accomode de caches et de protocoles de cohérence de caches standard et n'est pas sujette à l'effet domino en cas de retour arrière.

Les performances de l'architecture sont évaluées en fonctionnement normal et comparées à celles d'autres multiprocesseurs à mémoire partagée tolérant aux fautes. L'étude de performance a été réalisée par simulation à partir de traces d'adresses d'applications parallèles réelles.

Mots-clé : multiprocesseur à mémoire partagée, tolérance aux fautes, mémoire stable, récupération arrière.

1 Introduction

Multiprocessor systems based upon standard microprocessors are becoming ever-more commonplace, providing significant computational power at a fraction of the cost traditionally associated with systems of such power. While multiprocessors with distributed memory have gained much attention due to their theoretical peak performance claims, shared-memory multiprocessors continue to be the focus of development of several manufacturers, primarily due to the ease with which such systems can support traditional computing environments and programming paradigms. Nowadays, shared memory systems span the complete range of computing requirements from the personal workstation up to the supercomputer.

The dominant organisation of a typical shared-memory multiprocessor is as shown in figure 1, with a single shared bus used to connect processing elements to the shared memory and peripherals. Caches private to the processing elements together with various flavours of snoopy caching protocols minimise the bottleneck effect of the single bus. A discussion of the advantages and disadvantages of such architectures is not the concern of this paper, and we merely observe that shared bus systems are likely to continue to be constructed. What is of concern to this paper is how such systems can be constructed such that hardware faults affecting the CPUs in the system can be tolerated so that a reliable processing service can be provided in spite of those CPU failures.

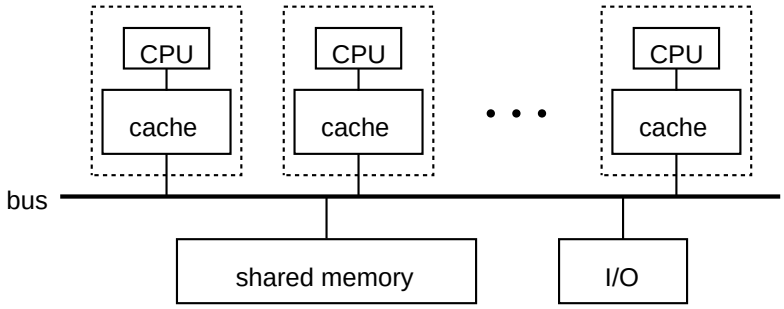

Figure 1: A typical shared memory architecture

The need for enhanced reliability is becoming an ever-more critical requirement as computing systems are used for applications where even short breaks in service are unacceptable. Moreover, it is simply infeasible with the complexity and range of present-day software systems to expect that such systems can be enhanced to implement hardware-fault tolerance. What is required is a hardware architecture that can transparently tolerate CPU faults, that is, without affecting the executing software and requiring no changes to

be made to that software. The presentation of such an architecture for shared memory multiprocessor systems is the primary purpose of this paper.

The remainder of this paper is organised as follows. The next section examines the fundamental problems of providing fault tolerance in a shared-memory multiprocessor, identifying the basic facilities that must be implemented and exemplifying these facilities with examples from some of the fault tolerant multiprocessor systems which are already available commercially. Section 3 discusses further the problems of error recovery in multiprocessor systems.

Section 4 introduces a fault tolerant architecture which directly supports shared memory semantics. The Recoverable Shared Memory (RSM) module which implements some of the features necessary for transparent fault tolerance, and which is the key novel feature in the architecture, is described in this section. Results from simulations of the architecture are given in section 5. Finally, concluding remarks are provided in section 6.

For simplicity and brevity, the paper concentrates on the problems of tolerating CPU failures in a shared memory environment and the novel solutions to these problems provided by the RSM. Other hardware fault scenarios, such as bus failures, are not covered here, although are clearly important for a complete system.

2 Fault Tolerance Issues

The basic principles behind fault tolerance are well understood [18]: a fault in a system will give rise to errors; the starting point for fault tolerance is the detection of an error, and an exception can be raised to signal that the fault tolerance provisions in the system need to be invoked. These provisions have to: (a) Deal with those errors, in particular to remove errors such that the state is no longer erroneous (error recovery); and (b) Deal with the fault that caused the errors, by identifying its location (fault location), reconfiguring the system to avoid the fault components (fault treatment), and switching the system back to providing its normal operation.

If the above actions are successful, such that the behaviour of the system has not breached the specification of the system, then the system will have successfully tolerated the fault and its effects, and no system failure will be apparent. Preventing system failure is of course the aim of the fault tolerance provisions.

As mentioned previously, this paper is concerned with tolerating the faults caused by failures of the microprocessor CPUs providing the processing power in a multiprocessor with the structure as shown in figure 1, and will therefore concentrate on the application of the above basic principles in this situation. Thus, regarding each processing element as a component of the

multiprocessor, we are concerned with providing reliable behaviour in the face of failures affecting these components.

To provide fault tolerance, the first requirement is that the effects of a CPU failure are detected. One approach, adopted in the Tandem-16 system [16], is to use a single CPU per processing element and to assume that a failure will result in fail-stop behaviour in that the processing element simply stops if something goes wrong in its logic. The other processors will detect this cessation in service through the absence of the "I am alive" messages which an active processor regularly broadcasts to all other processors. (Note, however, that the Tandem system is not a shared memory multiprocessor, and the single-CPU configuration for a processing element will not be considered further in this paper.)

More active forms of error detection are provided by replication checks where the activity of a CPU is replicated and the outputs from the replicas compared to detect an error. When duplicated CPUs are used, a comparator can detect differences caused if one of the CPUs fails and can raise an exception (or interrupt) to inform the rest of the system of the failure of this processing element such that the fault tolerance actions can be undertaken. This organisation is used in the processing elements of both the Stratus [25] and Sequoia [6] fault tolerant systems.

Higher levels of replication, such as using triplicated CPUs in a TMR organisation, can also be used, for instance as in the Tandem S2 System [13]. Here a different approach to fault tolerance is being taken as will be seen. In the case of the duplicated CPU discussed previously, a failure of a CPU results in the failure of the processing element of which it is a part: this processing element failure is a fault in the multiprocessor system, and actions elsewhere in the multiprocessor (as will be discussed shortly) have to provide the fault tolerance such that overall system behaviour is not impacted. In contrast, the application of TMR (and higher levels of replication) is simply the application of fault tolerance internal to the processing element such that failures of components within the processing element are never seen by the rest of the system. (For this reason, such applications of redundancy are sometimes referred to as masking redundancy.) Thus when a CPU fails in a TMR configuration, the divergence of its results from the other two CPUs can be detected by a voter which rejects the "odd man out" (error recovery), ignores the suspect CPU (fault treatment) and passes on the result of the majority to the rest of the system without interruption.

Returning to the situation of the dual-CPU processing element, errors in the system state (i.e. in the global memory) will have to be dealt with if a processing element fails. Errors could have spread in the system by there being a delay between the CPU failure occurring and the processing element actually stopping, during which time erroneous results could have been generated in the shared memory and hence propagated through the system.

In the dual-CPU case, there is unlikely to be such a delay and consequent propagation. However, there may still be errors since some of the state of the system will be contained within the failed processing element and this information may be inaccessible. For instance, the contents of the CPU registers and indeed the program counter are all part of the overall system state, and the caches on the processing element may also contain the up-to-date values of some memory locations. Thus, the global memory state may not be consistent with the processing that has been undertaken in the failed processing element, and some form of error recovery will be needed to cope with these errors. Without such error recovery, successful fault tolerance may not be achievable.

Two overall forms of error recovery could be applied: forward error recovery and backward error recovery [18]. Forward error recovery would require the "patching" up of the system state to fix the problems - for instance, if the failed processing element could be interrogated by another processing element to extract the necessary values, then the system state could be updated appropriately. However, it is unlikely that such an interrogation could take place reliably if a CPU failure has occurred- some of the information may be within the failed chip (e.g. in on-chip caches). Even with duplicated CPUs, it may be difficult to determine which of the pair has failed with the aim of extracting the information from the remaining "good" CPU. The alternative recovery strategy of backward error recovery requires the state of the whole system to be recovered to a prior known state (simulating the reversal of time, and hence is called backward error recovery) such that all of the errors are eradicated. How this can be achieved is discussed in more detail below.

The Stratus system effectively uses a forward error recovery scheme, but avoids the need to interrogate the failed processing element by running a computation simultaneously on two processing elements, each of which contains two CPUs (i.e. on 4 CPUs in total). If one processing element fails, then the other processing element can be used to provide all of the "internal" values, such that a new processing element can be brought into lock-step and the processing continued. (Alternatively, the computation can be continued on the single processing element pair with the hope that another failure does not affect that processing element. In this case, no error recovery is required.) In contrast, the Sequoia system effectively employs backward error recovery, and their scheme is described in the next section.

After error recovery has been carried out, the errors caused by the processing element fault have been dealt with, and so the next stage of fault tolerance is to deal with the fault itself. The location of the fault will be identified by the exception raised in the dual CPU configuration. If the fault was deemed to be transient (determined, for example, by running diagnostic checks on the faulty processing element), it may be appropriate to permit that processing element to continue to play a part in the system's activity.

If, however, the fault is permanent, then that processing element will not be used further, and the computation it was involved in can be restarted on another of the processing elements in the system. If forward error recovery has been used, for instance as in the Stratus system, no processing will have been lost, whereas if backward recovery has been invoked, as in the Sequioa system, some processing will have to be repeated. Note, however, that in a shared memory environment it is trivial to ensure that a computation can be picked up by another processing element - all of the information concerning that computation can be in shared memory and is accessible to all of the processing elements. In a distributed memory situation, as in the Tandem-16 system, this task can be much more complex.

Thus, in designing a fault tolerant multiprocessor, the designer is faced with typical engineering trade-offs. By adopting triplicated (or higher) levels of redundancy in the processing elements, the need for error recovery can be avoided. However, the cost and difficulties associated with this approach might suggest that a design based on duplicated CPUs and with provisions for backward error recovery would be more effective. Detailed discussions of such engineering trade-offs are not the purpose of this paper; indeed the different designs taken by Sequoia, Stratus and the Tandem S2 systems suggest that each approach has its place. In this paper we concentrate on the dual-CPU, backward error recovery approach and on the design of the Recoverable Shared Memory (RSM) module, a special memory module which supports backward error recovery in a shared memory environment. First, though, the basic problems of, and terminology for, backward error recovery in this environment must be discussed so that the facilities that the RSM must provide can be identified, and this is the purpose of the next section.

3 Backward Error Recovery in a Shared Memory Environment

The basic functions required for backward error recovery are that a processor can: (a) Establish a recovery point; (b) Recover the state back to that recovery point;(roll back) and (c) Commit a recovery point.

The time between the establishment of a recovery point and its eventual commitment is termed a recovery region. A recovery point is thus a point in a computation to which the state can be reset and hence the computation can be restarted from that point. If the establishment of the recovery point preceded the occurrence of a CPU failure, then recovery to that recovery point must eradicate all of the potentially erroneous effects of that fault (as discussed in the previous section).

To provide recovery, recovery data must be recorded, for which one of several techniques can be adopted. For example, a checkpoint can be taken when the recovery point is established, that is, a complete copy of the state taken and kept somewhere safe. Since the complete state is likely to be large, and a processor is unlikely to update a significant percentage of its state, more dynamic and optimal facilities can be provided. Shadow paging [20] provides a form of incremental checkpointing, by keeping a copy of only those memory pages that have been altered. The recovery cache [19] also provided incremental recording of recovery data. The Sequoia system makes use of a blocking cache [6] to provide recovery: having established a recovery point, a processor is not permitted to update main memory. Instead all writes are kept local to the processor in a blocking cache (i.e. non-write-through). If the processor fails, then the state in the main memory represents the state at the recovery point. The commitment of a recovery point by a processor consists of flushing its cache and its internal registers to main memory. Modified data are flushed into two distinct memory modules under processor control in order to handle memory and processor failures.

The CARER architecture [26] makes also use of a blocking cache with the assumption of fault free memory and cache. Assuming that memory is fault free avoids the need for a second memory module for recovery data and hence avoids the loss of time that would be necessary for copying modified data between the two modules. Assuming that caches are fault free limits the work to be done at commit time because blocks residing in cache can be included as recovery data. At commit time, all processor registers are first flushed to the cache and then all modified blocks in the cache are marked UNWRITABLE. This terminates the commit operation. UNWRITABLE blocks belong to the recovery point and have to be written back to memory if there are subsequently modified or replaced (i.e. copy on write).

While the changes a processor makes to memory can be undone by state restoration techniques such as those described above, not all of the manipulatable entities in a system can be recovered. For instance, as discussed in the previous section, the processing element itself is unrecoverable in that its contents (registers etc.) may not be accessible if that element has failed, so these have to be explicitly recorded when a recovery point is established (e.g. the program counter must be recorded to allow the computation to be restarted from that point). This topic is returned to in the next section. Also, a processor may manipulate other unrecoverable objects, such as peripherals, and the fault tolerant system must cope with the problem of backward recovery in this situation also. This point is not addressed in this paper and the discussion will concentrate on recovering from memory updates.

Since recovery data occupies some system resources, it is normal to commit recovery points at some interval, to allow this recovery data to be discarded. For example, in CARER a recovery point is committed each time

modified data in the cache has to be replaced, while in the Sequoia system a recovery point has to be committed when the blocking cache of a processor becomes full. For the tolerance of CPU faults it is, in general, sufficient to allow a processor to have a single extant recovery point such that the commitment of a recovery point can be synonymous with the establishment of the next recovery point, and thus two distinct operations (establish and commit) are not needed. In a more general situation, for instance if providing software-fault tolerance by recovery blocks [11], nested recovery regions and hence multiple extant recovery points and separate establish and discard operations make sense. As will be seen, even in the simple case it is useful to be able to separate the completion of a recovery region from the commencement of the next, and hence the two separate operations will be used in the following discussion.

In a shared memory multiprocessor, there is another complication to recovery that must be dealt with, concerning the parallel processors that will be executing simultaneously and the possible flow of information between these processors (via shared memory). Consider the following simple situation: two processors P1 and P2 have separately established recovery points, and P1 has written to a memory location that P2 has subsequently read from and acted upon. Now if P1 fails and backward recovery has to be applied, then it is also necessary to recover P2 to its recovery point which preceded the interaction with P1. Only by recovering P1 and P2 is a consistent system state restored. The recovery points of P1 and P2 which correspond to a consistent state are termed a recovery line [18].

One strategy for identifying a recovery line is to ensure that all processors establish recovery points simultaneously - that is, a system-wide recovery point. If recovery is then required because a processor fails, all processors have to be rolled back. This strategy has the disadvantage of unnecessarily recovering processors when no interactions between a processor and the failing processor have occurred. To avoid this disadvantage effectively requires processors to be recovered independently (rather than globally), and hence requires some other means for solving the problem of interdependencies (i.e. the problem of identifying a recovery line). One method is to avoid the need to identify a recovery line by ensuring that there are no inter-processor dependencies. This can be achieved by not actually providing shared memory (a strategy adopted in the Sequoia architecture) or by committing after each interprocessor interaction in order to remove the dependency (this is essentially what happens in CARER where a processor has to commit its recovery point each time one of its modified blocks in cache is accessed by another processor).

The Sequoia architecture prohibits direct data sharing between processors, leading to significant complications being imposed on the operating system software. While all memory modules can be accessed by all proces-

sors, shared data structures must be accessed within explicit critical sections protected by test-and-set locks, and the operating system has to carefully establish and commit recovery points and flush the blocking cache appropriately, to ensure the correct semantics [6].

An alternative solution to identifying recovery lines, which removes the need for the software complexities of the Sequoia approach and the frequent commitments of the CARER approach, is to actually compute a recovery line if recovery is required [18]. In order to do this it is necessary to record inter-processor dependencies which can be used to determine the set of processors which are dependent upon the processor which has failed [3]. Such a mechanism is provided by the Recoverable Shared Memory module described in the next section.

It should also be noted that in the Sequoia system it has been necessary to provide custom caches to provide the blocking behaviour. This precludes the use of standard snoopy caches and protocols in such a system. However, the speed of the latest generation of RISC chips is such that their manufacturers provide cache control logic (and chips) as part of their offerings, and it is increasingly difficult (and cost-ineffective) to design custom caching systems (and CPUs). Hence, it is desirable that standard processors, caches and caching protocols can be used in a shared memory multiprocessor, and yet fault tolerance using backward error recovery can be provided. This is another novel feature of the architecture to be presented in the next section.

4 Architecture of a fault tolerant shared memory multiprocessor

This section presents the salient features of a novel fault tolerant shared memory multiprocessor architecture [1] designed to transparently tolerate processor failures. The general architecture, which is depicted in figure 2, mainly consists of processing elements, and a Recoverable Shared Memory (RSM) module which provides normal memory functionality as well as a backward error recovery mechanism which is discussed in more detail below. The processing elements access the shared memory through private caches holding the most recently referenced memory locations.

The architecture has been designed to require specialised hardware only for the RSM. Standard processors, caches and cache coherence protocols can be used, and thus memory can be freely shared between processors. In particular, the recovery protocol avoids the use of dedicated blocking caches which require custom hardware and penalise the overall performance of the architecture. In principal, it should be possible to plug an RSM board into

[1] A description of this architecture can also be found in a previous report [5].

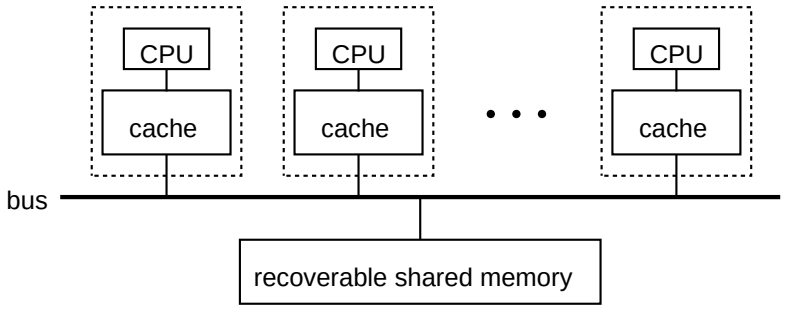

Figure 2: The recoverable shared memory architecture

an off-the-shelf shared memory multiprocessor to provide an error recovery mechanism for that system.

In the following, we present the backward error recovery protocol that is used in the architecture to obtain tolerance of processor failures, and discuss how this protocol is implemented by the RSM. Section 4.1 introduces the basic features and operation of the RSM with the simplifying assumption that there are no private caches interposed between the processors and the RSM. Section 4.2 then considers the additional complexities added when private (snoopy) caches are part of the multiprocessor.

4.1 Basic Features of the Recoverable Shared Memory

As discussed earlier, the backward error recovery protocol has to permit recovery points to be established, recovered to and committed, and must permit a recovery line to be identified when recovery is required. The basic mechanism in the RSM for providing recovery is to record recovery data for each memory location, by maintaining two copies of each location. When a recovery point is established, each copy contains the same data. Subsequent updates to a location are made to only one of the copies and thus the second copy retains the state of that location at the time the recovery point is established. As only a single extant recovery point is needed for tolerating CPU faults, only two copies of a location are ever needed.

To permit the identification of recovery lines it is necessary for the recovery protocol to (a) detect and record the existence of inter-processor dependencies which arise through their sharing of data in the memory; and (b) synchronise the recovery protocol operations of those dependent processors. This synchronisation is a vital part of the recovery protocol. Suppose, for instance, that processor P_j reads a cell previously modified by processor P_i within its current recovery region. If P_i is recovered because the processor on which it was executing fails, then P_j must also be recovered. Suppose now

that processor P_i writes into a cell previously modified by processor P_j within its current recovery region. If P_i is recovered then P_j must also be recovered since the value written by P_j has been overwritten by P_i and so cannot be recovered. In both previous cases, an extant recovery point is required for P_j so that the recovery line linking P_i and P_j actually exists. Thus dependent processors have to synchronise their actions on establishing, recovering to, and committing recovery points. One simplification can be obtained by ensuring that a processor always has an extant recovery point, by ensuring that a new recovery point is automatically established when a previous one is committed or restored. Thus the RSM protocol does not provide a separate *establish* operation. (For the interested reader, a more formal description of the recovery protocol may be found in [5, 14].)

Clearly, a key part of the recovery protocol is in detecting and tracking inter-processor dependencies. This is achieved in the RSM by means of recording *dependency relationships*. A dependency has to be recorded by the RSM in two cases :

- Whenever a processor P_i reads a cell previously modified by processor P_j within its current recovery region.
- Whenever a processor P_i modifies a cell previously modified by processor P_j within its current recovery region.

A processor p is said to be the *active writer* of a cell c if p has written to c within its current recovery region and c has not been written to subsequently by another processor.

This dependency information is stored within the RSM and used to compute recovery lines, that is the *dependency group* of processors involved in the commitment or restoration of a recovery point. Recovery of a processor P_i must induce recovery of its dependency group; this requires the values of all of the memory locations which have been updated by any processor in that group to be restored by the RSM to their prior values, using the recovery data recorded by the RSM for this purpose. Similarly, commitment of a process will induce the commitment of all processors in the dependency group; in this case the values of all of the memory locations which have been updated by any processor in that group must be committed. This entails the RSM in making the two copies of the memory location identical.

One important assumption for dependency tracking is that the RSM is only connected to the bus of the architecture and dependencies are tracked by *snooping* bus information. In order to record dependencies, when a memory location is accessed by a processor, the RSM needs the following information:

- The identity of the processor performing the access. This is achieved by fitting each processing element board with a unique identifier which is

transmitted whenever this processing element generate a read or write request.

- The type of access (read or write)
- The identifier of the processor that is active writer of the cell if any.

The commitment of a recovery point by a processor obeys a simple distributed *two-phase commit* [10] protocol. Processors are participants while the RSM is the coordinator of the protocol. In contrast to the standard two-phase commit protocol where the coordinator is responsible for triggering the protocol, it is a participant which initiates commitment; yet it is the coordinator itself which is responsible for actually committing data. When a participant wishes to issue a commit request, it must first flush its internal registers out to the RSM (since the values in these registers, which form part of the global state, cannot otherwise be accessed). It can then send a *do_commit* command to the RSM and wait for an interrupt meaning that commitment has terminated and that the processor can resume processing.

Upon receiving a *do_commit* command, the RSM scans its dependency information (to determine the recovery line) and informs all of the processors in the dependency group. A dependent processor can then flush its registers into the RSM if necessary and must acknowledge its completion of the first phase of the protocol. When all acknowledgements from the participant processors have been received, the RSM enters the second phase of the commit protocol. During this second phase, the recovery data of the cells whose active writers belong to the dependency group are discarded. Once this has been achieved, commitment is complete and a new recovery point is established for each processor belonging to the group. Thus the processors in the group are no longer dependent upon each other, and hence the dependency information in the RSM can be discarded and the participants allowed to proceed with their computations.

Let us consider now the implementation of the RSM in greater detail.

Servicing read and write requests

The RSM actions are best described by a finite state automaton. The automaton includes an *initialisation* state together with a *service* and *commit_phase2* states. In the service state of the RSM, most of the work is concerned with dependency management. Assume that dependency data is stored in a $n * n$ boolean matrix M; n being the maximum number of processors in the architecture. A matrix item $M(i, j)$ is set to true when processor P_i is dependent on P_j.

While read and write requests may refer to RSM cells, the RSM itself may record dependencies on a larger granularity. In the following, it is assumed

that the RSM physical space is divided into a set of contiguous *blocks* of identical size which is a power of two cells. Each block consists of (i) current values of the cells, (ii) a tag field containing either the identity of the active writer to the block (if any) or the *nil* value, and (iii) recovery data.

Basically a read request involving cell c requires the RSM to compute the identity of the containing target block b, record a dependency between the processor making the request and the active writer processor of the block (if any) in the matrix M, and deliver the current value of the cell. A write to a cell c will compute the target block b, record a dependency with the active writer if any, change the active writer of the block, and update the current value of the cell within the block b.

First phase of the commit protocol

Upon receiving a *do_commit* command from processor P_i, the RSM has to scan the dependency information it has recorded in matrix M during P_i's current recovery region to determine the *group* of processors which are required to commit atomically with P_i according to the recovery protocol. Once the dependency group has been computed, each processor in that group has to be informed that it is required to participate in this commitment, by means of a *prepare_to_commit* interrupt. This can be implemented in a variety of ways, using interrupt or message passing facilities provided by the bus. The RSM could generate such interrupts directly. Alternatively, the RSM could broadcast on the bus a bit vector conveying the group of dependent processors, with dedicated logic on each processor board checking whether the processor it is attached to has to participate to the group and generating the *prepare_to_commit* interrupt appropriately. (This checking could also be implemented in software.) Each processor in the dependency group must also issue a *do_commit* command in acknowledgement, meaning that as far as it is concerned, the first phase of the commit protocol is OK.

It should be noted that within the interval between the initial *do_commit* command and the receipt of acknowledgements from the dependent processors, some new dependencies may have been created as the RSM can continue servicing read and write commands from processors that are not blocked waiting for the end of the commit protocol. These processors have to be added to the dependency group if this concurrency is permitted. It may also occur that a processor not already part of the group decides to commit its current recovery point and sends a *do_commit* command to the RSM. This processor is added to the (current) group as well as all the processors dependent upon it. This mechanism provides a simple means for implementing multiple concurrent processor groups.

One crucial point is that computation of the dependency group has to be atomic with respect to read and write accesses to the RSM. If it is not

atomic, the group could be incorrectly calculated by the RSM. A simple way to implement this atomicity is to serialise the group computation and read write accesses.

The dependency group computation algorithm is given in the C programming language in figure 3, with the assumption that the number of processors n can be encoded within an integer variable, and the matrix M is implemented by an integer array where each array element is considered as a bit vector indexed by a processor identifier. Upon reception of a *do_commit* command, the RSM executes the *do_commit* procedure. The bit vector *group* denotes the processors which are members of the dependency group, while *do_commit_received* is the bit vector denoting the processors that have completed the first phase of the commit protocol. The *do_commit* procedure of figure 3 will cause a state transition of the RSM automaton into the *commit_phase*2 state implementing the second phase of the commit protocol if the following condition is verified :

$$Q: \ ((group = do\_commit\_received) \wedge (\forall i: \ i \in group : immediate\_ancestors(i) \in group)$$

This condition expresses the requirement that all processors belonging to the group of dependent processors have completed the first phase.

Q may not be satisfied in two situations. Firstly, some processors that have already been informed that they are group members have not yet completed the first phase, in which case the RSM must wait for the reception of their *do_commit* acknowledgement commands. Secondly, some new processors have become group members since the last computation of *group* and thus must be informed. Notice also that the dependency computation algorithm must avoid interrupting a given processor more than once.

There are many ways to devise an algorithm satisfying the previous requirements. In figure 3, a simple solution is given. The algorithm checks the Q condition and as a side-effect computes a new value of *group*. If the new value of *group* is different from the last value, the new members are informed. The complexity of the algorithm is $O(n)$, with the *do_commit* procedure being executed at most n times. The first phase of the commit protocol is thus $O(n^2)$. Note that termination is obvious from figure 3 assuming that a processor acknowledges a *prepare_to_commit* request within a finite time and since the number of processors is bounded.

Second phase of the commit protocol

The basic actions which have to be performed in the second phase of the commit protocol are the following:

1. Discard the recovery data of all the blocks whose active writers belong to the dependency group and establish a new recovery point. This can be achieved by setting the active writer field of those blocks to the *nil* value and by setting the recovery data values to the current values.

```
int state;             /* current state of the automaton */
int group;             /* dependency group computed so far (bit vector) */
int do_commit_received;/* bit vector of do_commit commands */
                       /* received from the processors */
int M[n];              /* dependency matrix */

INITIALISATION:
do_commit_received = 0; group = 0;
for(j=0;j<n;j++) M[j] = (1<<j); /* a processor is an ancestor of itself */
state = SERVICE;

SERVICE:
read(address) {
  /* create a dependency if necessary and
     return the value stored at address */
}

write(address, value) {
  /* create a dependency if necessary and perform the write */
}

do_commit(int i) /* i is a processor id */
/* the  processor i is willing to commit or acknowledges a request of
   the RSM following a commit request from a dependent processor */
{
  int dependent_members;
  int j;                /* processor id */

      /* add processor i to the group */
          group |= (1<<i);
      do_commit_received |= (1<<i);

      /* compute new dependent members */
          dependent_members = group;
          for(j=0; j<n; j++)
            { /* if processor j is a member of the group */
              if ((group & (1<<j)) != 0)
                 dependent_members |= M[j]; /* add immediate ancestors */
            }
      /* {dependent_members = group ==> group is exact} */

      /* check for termination condition Q and
         inform new members if necessary */
          if ((do_commit_received == group) &&
              (dependent_members == group))
              state = COMMIT_PHASE2;
          else if (dependent_members != group)
                  { /* broadcast (dependent_members&~group) on the bus */
                     group = dependent_members;
                  }
 } /* do_commit */
```

Figure 3: Computing a dependency group

2. Break the dependencies by updating the dependency matrix M appropriately.

3. Broadcast a *commit_done* interrupt to the processors belonging to the dependency group to permit them to restart computations.

As in the first phase of the commit protocol, these operations need to be atomic with respect to processor accesses.

The implementation of the second phase of the commit protocol has a great impact on the overall performance of the architecture since a processor must not modify a block for which the recovery data has yet to be recorded. Several implementations of the second phase can be devised. A trivial solution would consist of sequentially checking every block of the RSM and copying the current value of the block to its recovery counterpart if necessary. This leads to the second phase of the commit protocol taking a time proportional to the size of the RSM, which may not be desirable. Moreover, the processors must be prevented from restarting before this copying has been completed.

Several refinements to this straightforward algorithm can be made. The RSM could maintain a per processor linked list of modified blocks. The time needed to perform the second phase is then proportional to the number of memory blocks which had been updated by the processor group, but at the cost of an extra storage within the RSM.

Alternatively, it is possible to permit the RSM to start normal memory operations, and to delay the effective copying of a block (to provide recovery data for the new recovery point) until it is really needed, that is, if a processor attempts to modify that block. This copy on write mechanism allows the second phase time to be interleaved with normal processor accesses, and processors do not need to be stalled until copying has finally completed. However it is still necessary to mark blocks within the RSM to determine whether a given block has to be copied or not on a subsequent write access. One approach is to use *checkpoint identifiers* [26]. In this approach, a checkpoint identifier is associated within the RSM with each block and with each processor. When a block is modified, the current value of the checkpoint identifier of the active writer is stored along with the block. When a processor commits, its checkpoint identifier is incremented. Upon each write access, if the checkpoint identifier of the block is less than the checkpoint identifier of its last active writer, the current value of the block is needed for recovery data and so needs to be copied. Before allowing the write to perform, the block is copied to its recovery counterpart and the active writer and checkpoint identifier fields of the block are set accordingly. Similar optimisations were provided in various implementations of the recovery cache [18].

Tolerating processor failures

In this paper we have assumed that the processors used are *fail-stop* [21], and that a failed processor can be easily identified. This is not a severe constraint on the architecture for fail-stop processors are common practice in the field of hardware fault tolerance (e.g. through the use of duplicated CPUs). In case of failure, the processor ideally will signal a failure interrupt on the bus which will be caught by one of the live processors. This processor can trigger the recovery process by issuing a *do_rollback(i)* command to the RSM, where i denotes the processor that failed.

Upon reception of the *do_rollback(i)* command, the dependency group of i is computed by the RSM, thus identifying the group of processors which must be recovered in order to reset the system state to a consistent state, that is the state at a recovery line, in a manner similar to the second phase of the commit protocol discussed earlier. However, the values of the blocks modified by the members of the dependency group have to be reset to their prior state, by copying the values held in the recovery data associated with those blocks. Each dependent processor must be interrupted by a *roll_back* interrupt, to cause them to abandon their current processing. The dependencies are broken by resetting matrix M appropriately, and the RSM reenters the service state. Recovering back to a recovery point is a simple protocol requiring a single phase compared to the commit protocol which requires two phases.

A particular situation may occur if group commitment is in progress when recovery is demanded. Since the same processor may belong both to a recovery group and a commit group, it is necessary to check for this at the end of the recovery procedure. Members of the recovery group are removed from the commit group. If the remaining commit group is not empty, the *do_commit* procedure of figure 3 is executed taking one member of the commit group as an argument.

Finally, after recovery has taken place, the global system state is consistent, and the processors which have been recovered can recommence execution of normal computations. The computation that was running on the failed processor can be re-executed on one of the remaining processors and hence a system failure will have been averted. Moreover, the processor failure will have been tolerated transparently, since no alterations were required to the software of the application to provide these tolerance actions.

4.2 Cache coherence protocols

The previous section described the basic operation of the RSM and how the recovery mechanisms work. This section addresses the complexities added when the multiprocessor architecture contains caches private to each processing element, as is the case with any realistic multiprocessor. In fact, the

primary changes needed to the RSM concern the dependency tracking mechanisms. Thus, this section examines the influence of the *cache coherence protocol* on these mechanisms.

Most hardware cache coherence protocols proposed so far rely on the fact that bus traffic can be monitored (*snooped*) by all caches attached to that bus. Snoopy caches maintain a *tag field* stored along with each loaded line to indicate the state of that line. The tag field generally encodes whether the line has been modified with respect to the contents of the corresponding shared memory location, and whether the line has been loaded into another cache. Two main classes of snooping cache coherence protocols can be distinguished depending upon the actions performed by caches when a shared line is modified. *Write Invalidate* protocols cause an invalidation message to be broadcast on the bus whenever any data potentially loaded into other caches is updated, to cause cache lines to be invalidated. *Write Update* protocols broadcast the new value whenever data potentially resident in other caches is updated. For the sake of brevity, only the Berkeley protocol [15] is examined below, as a representative of the write invalidate family of protocols. The changes necessary for write update protocols such as the Firefly protocol [24] are not detailed here for space reasons.

In the Berkeley protocol, a cache line can be in one of the four following states (line states are described according to the terminology found in [23]) :

1. Invalid (I). The cache copy is not up-to-date.

2. Non-modified Shared (S). The line has not been modified since it was loaded into this cache. Other caches may also have a copy; one of these copies might be in state O while others must be in state S.

3. Modified Exclusive (M). The line is modified with respect to shared memory. No other copy exists. This cache is the owner of the line.

4. Modified Shared (O). The line is modified with respect to shared memory. Other caches may have a copy (in state S). This cache is the owner of the line. (Hence the abbreviation O.)

Figure 4 depicts the state transition diagram for the Berkeley protocol.

Recall that the RSM maintains dependencies on memory blocks. In contrast to the previous section, where the block granularity could be as small as a RSM cell, when caches are present the RSM must record dependencies on at least a cache-line size granularity, since a cache-line is the minimal unit of transfer on the bus. In the following, we examine the operations performed by the cache protocol and the various actions taken by the RSM so as to track the dependencies when a processor performs respectively a read miss, a write hit, and a write miss on its cache. (A read hit does not generate any action on the bus and thus does not need to be considered further.)

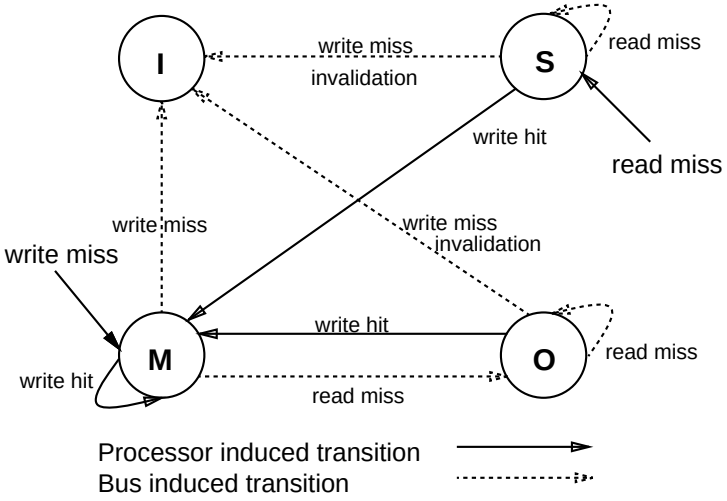

Figure 4: Berkeley state transition diagram

Processor P_i performs a Read Miss If there exists a cache with a copy of the line in state M or O, this cache must supply a copy of the line to the requesting cache and set its state to O. Otherwise the line must come from shared memory. In both cases, the line is loaded in state S in the requesting cache.

If the target block containing the line has an active writer P_j, a dependency must be created in the RSM between P_j and P_i. As far as dependency management is concerned, no distinction is made whether the requested line comes from another cache or from the RSM, although the inter-cache transfer must be detected by the RSM snooping on the bus.

Processor P_i performs a Write Hit If the line is already in state M, the write proceeds without delay. Otherwise, (in state S or O) an invalidation signal must be sent on the bus (see figure 4). All other caches matching the line address must invalidate their copy. The line state is changed to M in the originating cache.

The invalidation signal is snooped by the RSM. If the corresponding block has no active writer, P_i becomes its active writer. Otherwise, if P_j was the active writer, a P_j dependency is created and P_i becomes the active writer of the block.

Processor P_i performs a Write Miss Like a read miss, the line comes from its owner or from shared memory. All other caches invalidate their copy if any. The line is loaded in state M.

The RSM snoops the data transfer if the line comes from another cache. As above, if the corresponding block has no active writer, P_i becomes the active writer; otherwise, a dependency is created between P_i and P_j and P_i becomes the active writer of the block. Since cache lines can contain several processor addressable cells and the line is now cached by P_i in state M, the RSM cannot detect a further read on a different cell of the line because it would not generate any bus traffic (see figure 4). So, a dependency between P_j and P_i is also created to prevent the case in which a cell previously modified by P_j would be locally read by P_i. In other words, the RSM adopts a conservative approach by creating some dependencies which are not strictly required by the protocol to preserve the coherence of processor checkpoints.

It should be noted that the RSM must keep pace with the information exchange rate on the bus due to the cache coherence protocol. If this were not the case, the RSM might miss some dependencies that need to be recorded. Satisfying such a requirement typically depends on the detailed specifications and timings of the bus and caches and will not be examined here in greater detail.

The commitment of a recovery point when caches are present is similar to the situation where no caches are present. What is required is that when a participant processor initiates commitment or acknowledges a *prepare_to_commit* request from the RSM, the processor must flush its cache as well as its internal registers. Similarly, recovery must cause a cache invalidation.

In summary, no special purpose caches or coherence protocols are needed in the architecture being presented here, which can accommodate standard cache behaviour with the RSM performing dependency tracking by snooping the bus traffic. This is a notable difference with other proposals for fault tolerant shared memory multiprocessors [6, 26, 1].

5 Performance Evaluation

This section presents the performance evaluation of a shared memory multiprocessor machine incorporating a RSM (which will be termed the RSMM - Recoverable Shared Memory Multiprocessor - in the following discussion). Through simulation, the performance of the RSMM is compared against the performance of a standard multiprocessor architecture without any fault tolerance capabilities and against that of two other approaches for fault tolerant shared memory multiprocessors, namely CARER and Sequoia.

5.1 Methodology and workload

The simulations were conducted using an instruction level simulator driven by a set of memory references generated by instrumenting application code with the Abstract Execution technique [17]. The simulator implements an efficient execution driven simulation method similar to that described in [7]. (Further information on the simulation tool may be found in [9].) Execution driven simulation controls the address trace generation to ensure that the trace corresponds to that which would be obtained if that application was actually executed on the architecture being simulated. This technique thus supports the derivation of simulations which accurately model the architecture. The simulation models were parameterized with the characteristics of Sun multiprocessor SparcServers, with a 320Mbytes/sec. synchronous bus, 64kbytes unified direct-mapped caches with 32 bytes lines and IEEE write invalidate cache coherence protocol. To simplify the performance comparison, all of the fault tolerant architectures are modelled with these parameters in common.

For RSMM, the error recovery protocol is that described in section 4. For the second phase of the commit protocol the stable memory implements the copy on write mechanism described in section 4.1. Since the extent of recovery regions in the RSMM is not controlled by any hardware or application parameter, it is necessary to fix a rate for the frequency of recovery point establishment (and hence commitment) for the simulations. The only situation where RSMM may be forced to commit a recovery point and to establish a new one is to prevent the loss or duplication of an operation on an unrecoverable object, for instance, I/O devices [18]. This classical technique for dealing with unrecoverable operations is used by CARER and Sequoia, and ensures that an I/O operation cannot be repeated. Thus, for the RSMM simulations, each I/O operation leads to the establishment of a recovery point. To obtain an average I/O rate, the interrupt rate on a NFS file server was measured, and from this measurement a rate of 1000 interrupts per second was used in the simulations.

For the CARER simulation, a recovery point is committed and a new one established whenever a modified line in a cache needs to be replaced and whenever a modified line is read by a different CPU. For Sequoia, recovery points are established and committed as described in section 3, that is, whenever a blocking cache is full or a modified cache line needs to be flushed, or whenever a critical section which requires coherence of the shared memory is exited.

The workload comprises 4 parallel applications drawn from the SPLASH benchmark suite [22]. Application cholesky performs sparse matrix factorisation; mp3d simulates rarefied hypersonic flows; pthor simulates digital circuits at the logic level; and water simulates the evolution of a system of

water molecules. Only the parallel phase of the computation was simulated, resulting in 65 to 80 million memory references for each application. The four applications were simulated on the four architectures for 1 to 8 CPUs.

5.2 Experimental results

5.2.1 Performance of the architectures

Figure 5 shows the MIPS (million instructions per second) performance of the four architectures for the four simulated applications. The performance degradation for the RSMM compared against the standard (non fault-tolerant) architecture is relatively small, despite a high commit rate for the RSMM (1000 per second). Performance degradation with eight CPUs remains below 15% except for mp3d where it is about 30% (for reasons discussed below).

For the other fault tolerant approaches, the performance of CARER is relatively close to these of RSMM for cholesky and water (10% performance degradation) but the degradation grows to 65% for pthor and mp3d. CARER achieves these results despite the restrictive failure hypothesis (i.e. the caches are fault-free) that permit a very efficient implementation of its commit protocol. The Sequoia approach appears to offer the lowest performance of all three fault tolerant architectures. Performance remains below 100 MIPS independent of the application or number of CPUs used. The performance degradation for this architecture always exceeds 20% for one CPU and can be as high as 85 % (pthor with 8 CPUs).

These results are very encouraging for the RSMM approach to fault tolerance. Some degradation in performance over a non fault tolerant architecture is inevitable, due to the error recovery provisions in the RSMM. Nevertheless, these simulations suggest a relatively modest degradation in general. When compared to the other fault tolerance approaches, the simulations suggest that the RSMM approach provides the best overall performance.

5.2.2 Behaviour of the applications

It may be observed from figure 5 that for all the architectures considered, the performance degradation varies significantly and is application dependent. This section studies the characteristics of these applications that have the most influence on performance degradation.

Figures 6 and 7 show, for each application the distribution of bus transactions for 10000 memory references: (a) misses serviced by shared memory; (b) misses serviced by caches; (c) write invalidations; (d) write backs arising from the replacement of a modified cache line (only for RSMM since Sequoia and CARER use blocking caches); and (e) write backs arising from cache flushes for RSMM and Sequoia or from the replacement of UNWRITABLE

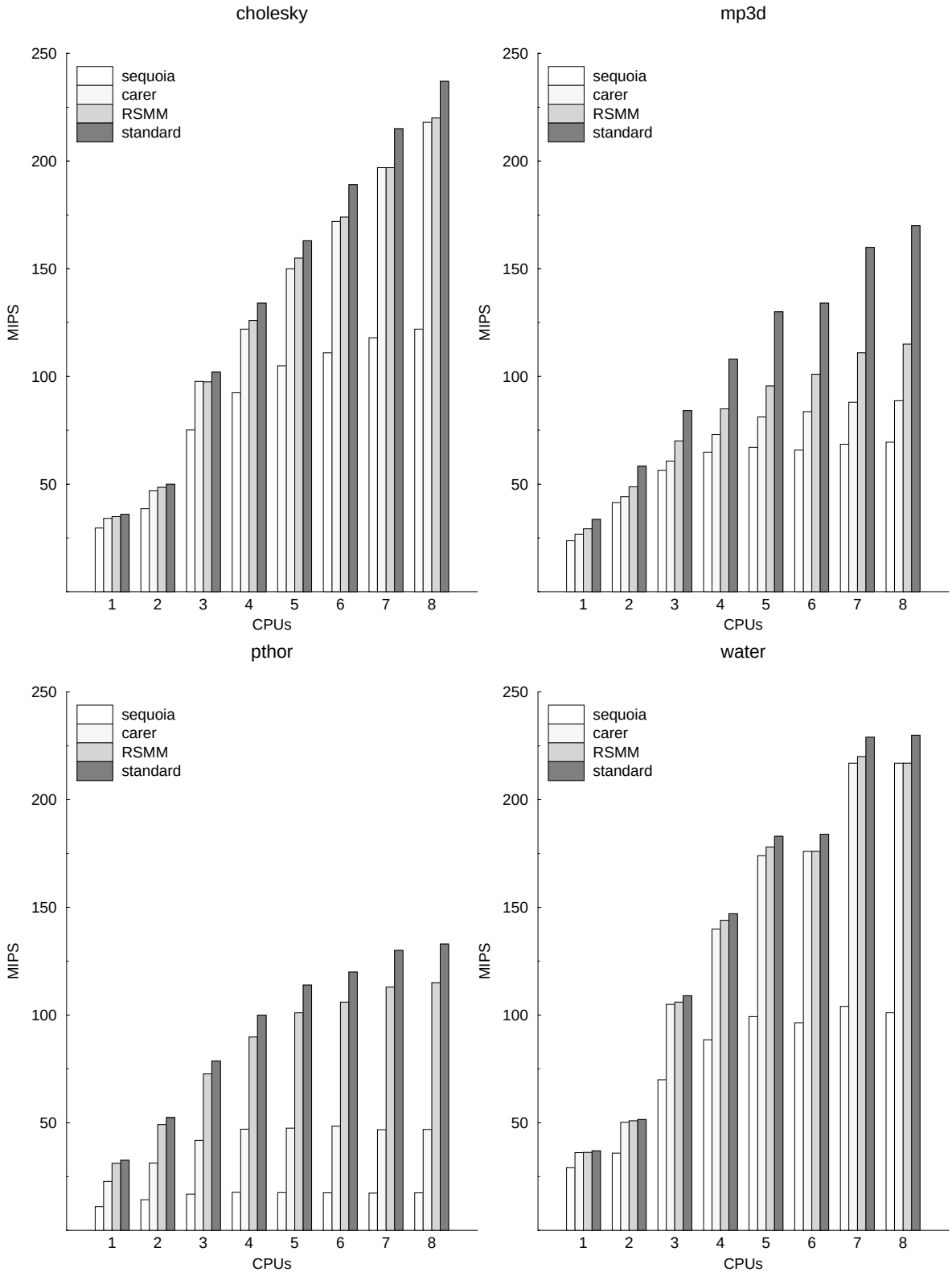

Figure 5: Performance

cache lines for CARER. The figures also show the number of recovery points established for 10000 memory references. Three reasons for establishment are distinguished: the recovery points established because of interrupts (only for RSMM), the recovery points established before replacing a modified cache line (for CARER and Sequoia) and the recovery points established because of data sharing. For RSMM the latter are the recovery points established because of dependencies; for CARER they are established because of a miss on a cache line that has been modified in another cache; and for Sequoia they are the recovery points established upon exit from a critical section.

cholesky

The standard architecture attains good performance for cholesky, with a speedup of 6.8 with 8 CPUs. This good behaviour is caused by a high cache hit rate of 99.2%. Data sharing is at a coarse granularity in this application. Although the 6% write ratio is comparable to other applications, the caches contain a low proportion of modified data due to the good locality of write references. Only 30% of replacements require a write back.

These characteristics allow performance degradation for RSMM to remain always below 10% for this application. The caches do not contain a lot of modified data; on average with 4 CPUs, 360 cache lines are flushed at each commit. Also, the data sharing pattern of the application only creates a small number of dependencies (the average size of the group of dependent processors is 2.8 with 8 CPUs). These two factors explain the good performance of RSMM for this application.

The performance of CARER is close to that of RSMM for this application despite disproportionate recovery point establishment rates (CARER establishes 15 times more recovery points than RSMM), due to the low cost of establishing a recovery point in CARER. Most recovery points are established when modified cache lines are replaced; only a few result from data sharing.

Sequoia suffers from an even higher recovery point establishment rate that CARER, mostly caused by critical sections that require frequent cache flushes. Moreover, the invalidation of unmodified cache lines on entry of a critical section contributes to lower hits rate from 99.2% to 98%.

mp3d

The behaviour of mp3d is clearly worse than cholesky for the standard architecture with a speedup of 5 for 8 CPUs. Cache hit rate is lower (98.3%) due to a worse write locality and to a 10% write ratio with 70% of replacements leading to write backs. Data sharing is very prevalent in this application; 77% of reads and 87% of writes reference shared data. Due to this heavy

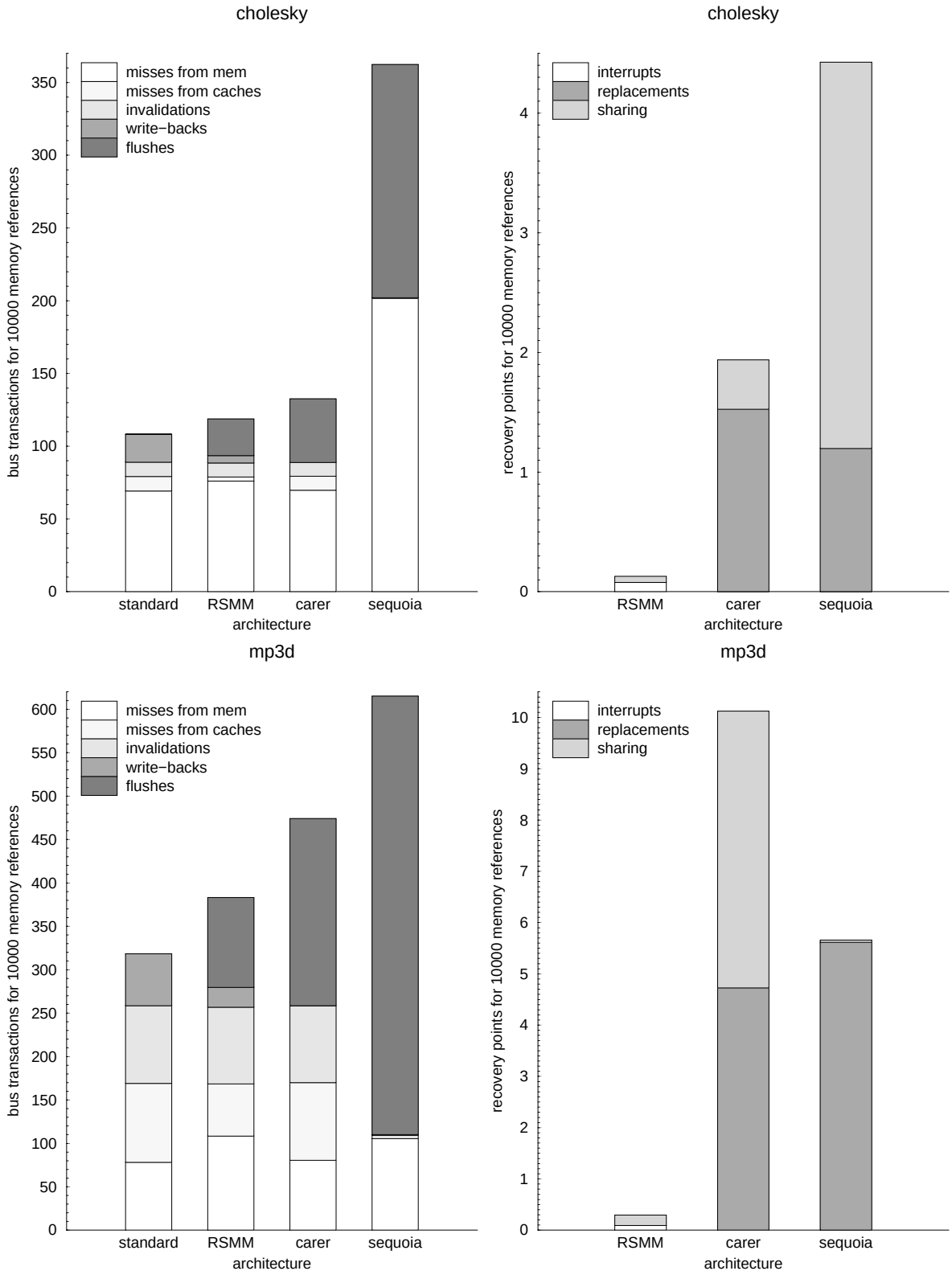

Figure 6: Application behaviour

data sharing, half of the misses are serviced by caches. The large number of cache to cache transfers lowers the performance since a cache servicing a miss cannot service requests coming from its CPU.

Of all four applications, RSMM has the worse performance degradation for mpd3. Performance degradation is 13% for one CPU and attains 33% for eight CPUs. The major factor contributing to this result is the large amount of modified data loaded within caches when a recovery point is committed - with four CPUs, an average of 1250 cache lines are flushed to memory. Moreover, due to the heavy data sharing, all processors are dependent. This considerably lengthens the duration of the first phase of the commit protocol.

CARER also does not behave well for this application because of the heavy data sharing that forces the establishment of a large number of recovery points. Moreover, a high number of modified cache lines are replaced.

Although synchronisation operations are infrequent, Sequoia suffers from the large number of modified cache lines replaced.

pthor

The standard architecture obtains low performance for the pthor application, with a speedup of 4 for 8 CPUs. Cache hit rate is low (97.5%) due to a large working set. Caches perform a significant number of write backs. Data sharing, although less intensive than for mp3d, contributes to limit performance.

Performance degradation for RSMM is much less for pthor than for mp3d (13% degradation for 8 CPUs). This behaviour is caused by the different amount of data flushed to memory prior when a recovery point is committed. With 4 CPUs, 570 cache lines are flushed, compared to 1250 for mp3d.

For CARER, although data sharing is less intense for pthor than for mp3d (38 cache to cache transfers vs 89 for 10000 memory references), the number of recovery points established because of data sharing is higher for pthor than for mp3d (12 vs 5 for 10000 memory references). This is caused by the data sharing pattern which is different for the two applications. In pthor, shared variables are accessed within short - but frequent - critical sections, thus leading to the establishing of a lot of recovery points.

The large number of locking operations and the large data set that causes a lot of replacements severely limit Sequoia performance for this application. No improvements of performance are achieved above 3 CPUs. Due to data cache invalidations upon exit of critical sections, the cache hit rate is lowered (84% instead of 97.5%).

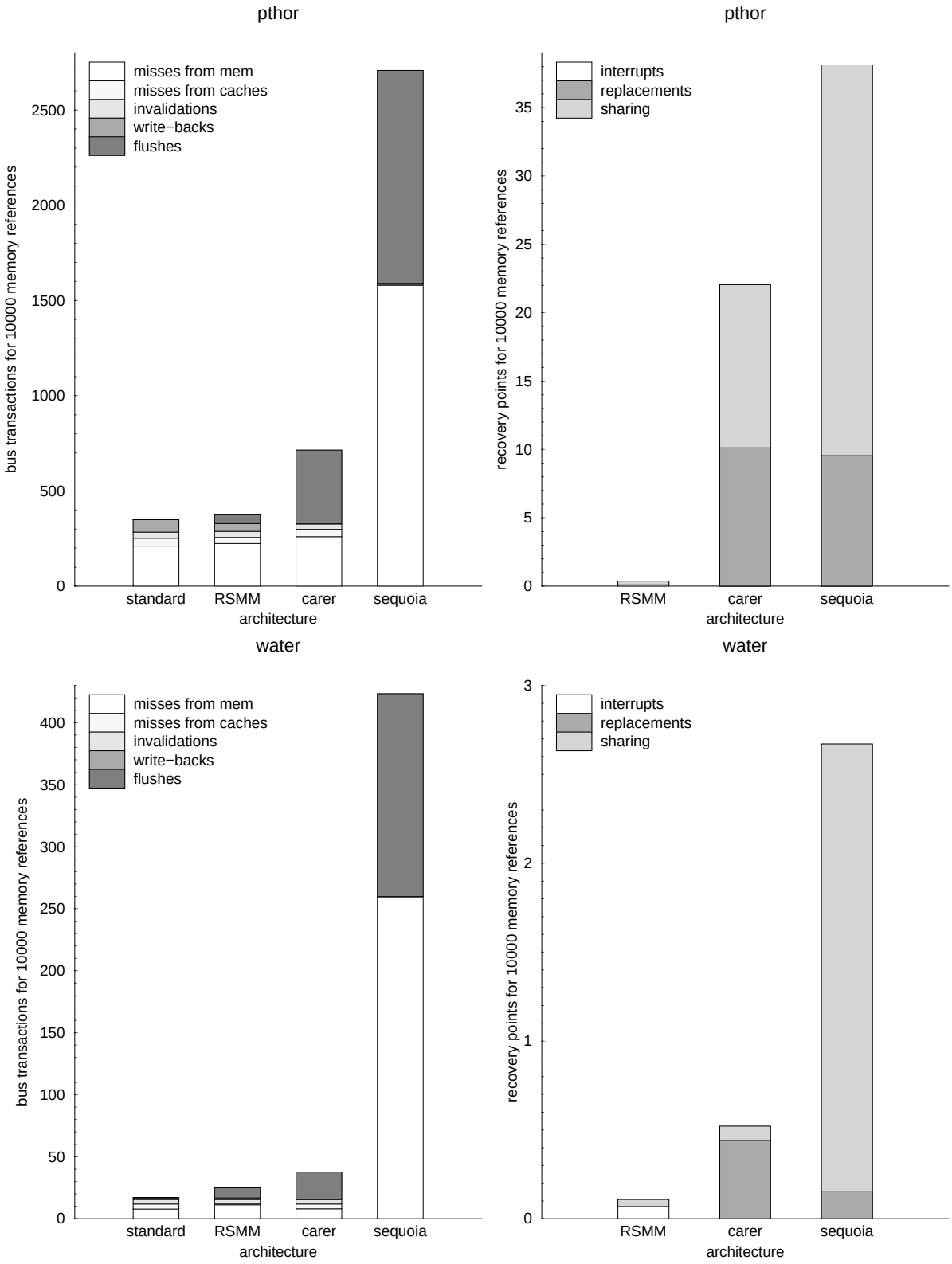

Figure 7: Application behaviour

water

The water application offers high performance for the standard architecture because of a high hit rate (99.88%) and a small data set. Moreover, data sharing is negligible.

As might be expected, RSMM performance is very good for this application because of the small amount of modified data within caches when recovery points are committed and of the few dependencies (2.6 dependent CPUs in average for 8 CPUs). CARER also behaves well for this application because of the small working set that only causes a few replacements of modified cache lines.

For Sequoia, only a small number of recovery points are established because of the replacement of modified cache lines, since the working set is small. Most of the recovery points are established because of critical sections.

5.3 Recoverable shared memory implementation

As there are several ways in which an RSM could be implemented, as mentioned in an earlier section, it is important to consider the influence such implementations would have on the performance of a system incorporating an RSM. Of particular concern is the potentially expensive operation of copying the current values of RSM cells for use as recovery data, when a recovery point is established. This is addressed in this section. The following section investigates the cost of implementing dependency tracking in the RSM.

Figure 8 shows the influence of this aspect of the RSM implementation on performance degradation. Three implementations are considered: one using copy on write, one using a per processor list of modified memory blocks and the last which is a control case where the copying time is assumed to be nil.

As can be seen, and as might be expected, the different implementations greatly influence the performance degradation suffered by an application. However, the degradation ratio between the different implementations remains constant independent of the application. The copy on write implementation behaves better than the implementation using a per processor list of modified memory blocks, although the number of blocks to be copied (and so the time needed to copy those blocks) is the same for both. With the list of modified blocks implementation, the duration of the copy is concentrated at the end of the first phase of the commit protocol. Although the CPUs restart their computation at the end of the first phase, they are not allowed to access the bus until all blocks have been copied and so quickly become delayed waiting for the bus to be released. If copy on write is used, the copying can be interleaved with CPU memory accesses. Thus the CPUs are only kept waiting for short periods of time resulting in better overall performance. The per-

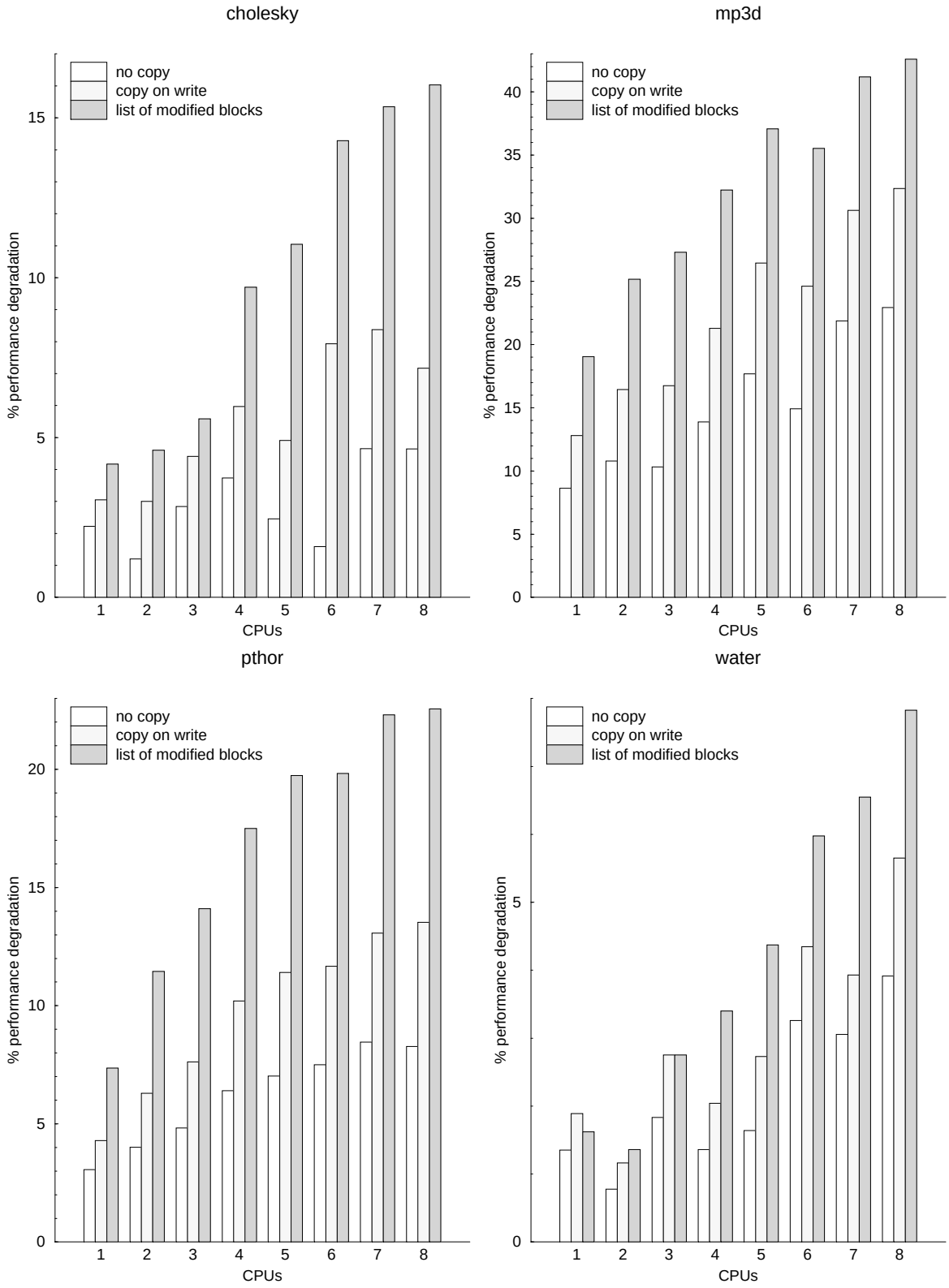

Figure 8: Recoverable shared memory implementation

formance is naturally the best for the control (no copy) case, which hence indicates the upper performance bound for the RSM.

5.4 Dependency management

Dependency management adds some complexity to the implementation of the RSM. In a simple implementation, all CPUs could establish a global recovery point thus avoiding the burden of dependency management within the RSM. In this case, a standard memory interface is sufficient for the RSM since it no longer needs to snoop bus transactions to log dependencies The commit protocol is also simpler. However, the potential gain of dependency tracking is in minimising the number of processors that have to be recovered in the event of a failure of one processor. Thus, it is necessary to study the impact of dependency management on performance to investigate whether it is worth the added implementation complexity.

Figure 9 presents for each application the performance degradation observed with eight CPUs, for RSM with and without dependency management. The figure also shows the average number of dependent processors at each commitment of a recovery point. For pthor and mp3d, the performance of the two versions of the RSM are nearly identical since for these applications all processors are dependent, and hence the presence of dependency tracking is irrelevant. In contrast, for cholesky and water, where the average group size never exceeds three processors, the dependency management shows its efficiency since it reduces the performance degradation by a factor of two. The main reason for this is that with dependency management the number of blocks flushed when a recovery point is committed is reduced since less CPUs are dependent and hence forced to flush their caches. For example, with the water application, 190 cache lines on average are flushed to memory each time a recovery point is committed. When dependency management is suppressed, the number of cache lines flushed grows to 330.

5.5 Summary of the simulation results

These simulations have demonstrated that the CARER and Sequoia approaches to implementing a fault tolerant shared memory multiprocessor both exhibit similar performance behaviour. Both require the commitment of a previous recovery point and the establishment of a new recovery point each time a modified cache line has to be replaced, and when data sharing occurs (for Sequoia, data sharing is enforced explicitly by means of the locking protocol). The difference in performance of these two architectures primarily results from the differing costs of recovery point operations. A realistic implementation of CARER should consider errors within caches and so would obtain roughly the same performance as Sequoia because of the necessary

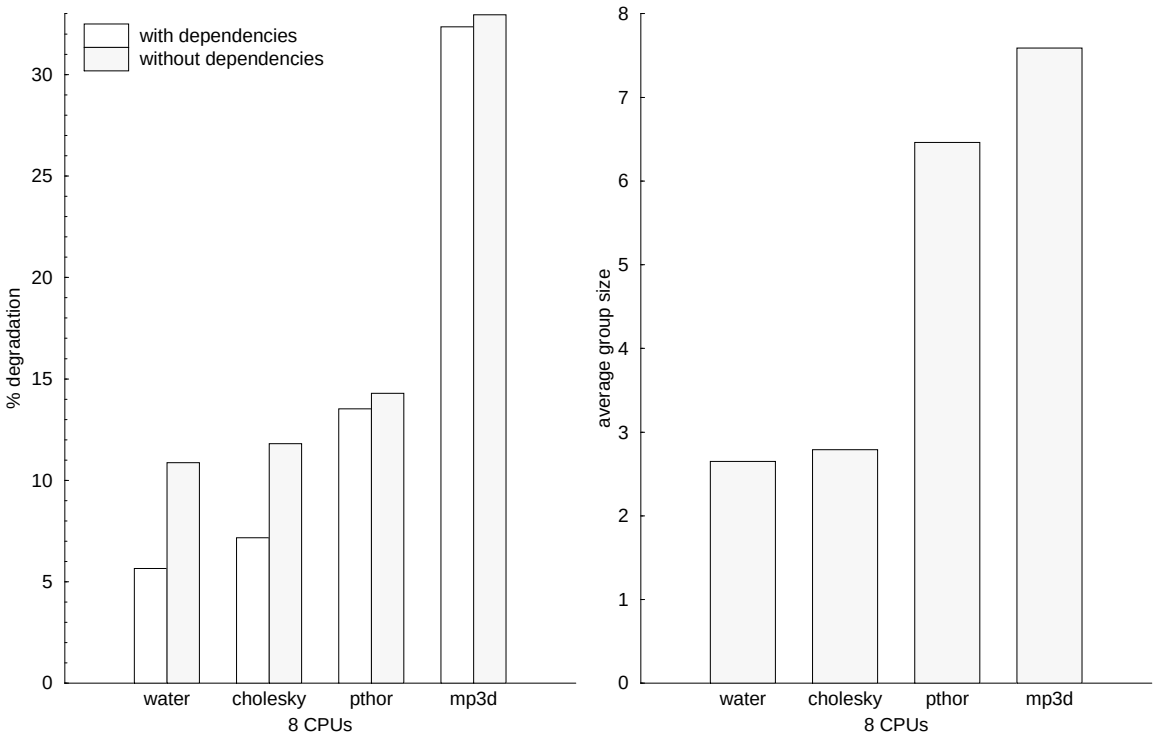

Figure 9: efficiency of dependency management

cache flushes. The rate at which recovery points are established and committed is controlled by cache parameters (size, associativity, replacement policy) and by the data sharing pattern of the application programs. This results in a high, uncontrollable and unpredictable frequency of recovery points establishing (between 25 and 100 times more than for the RSM approach). Some memory access patterns in the applications can even force the establishment of a recovery point at each data reference.

The RSM approach to implementing fault tolerance in a shared memory multiprocessor eliminates most of these disadvantages. The need for commitment/establishment of a recovery point is controlled primarily by the interactions of the architecture with its external environment (e.g. for I/O) independently of any architectural parameter. These interactions are much less frequent than cache line replacement. Recovery points are also independent of the communication patterns of the application programs, owing to the dependency tracking mechanism.

The fault tolerance overhead is concentrated in the commitment phase of the recovery protocol. Three factors can influence this overhead:

- the amount of data modified,
- the number of dependent processors,
- the bus load of the machine.

The amount of data modified is the major factor that influences performance degradation. The duration of the commit is directly proportional to the amount of data that has been modified. In turn, the amount of data is governed by the number of processors that are dependent upon the processor that issued the commit. Thus, the dependency management mechanisms in the RSM minimises the number of processors that are affected by the commit (except of course if they are all dependent). The dependent processors impose another overhead on commitment, since some modified data may be resident in the processor-private caches. Thus commitment requires the modified lines in the caches to be flushed back to the RSM. Thus, the importance of the dependency tracking mechanism in the RSM increases with the number of processors in a system, in that minimising the number of dependent processors will minimise the amount of cache flushing and data committing. Note, of course, that the cache flushing is required in an ordinary shared memory multiprocessor, and the overhead of cache flushing is not just RSM-specific.

The bus load also influences the performance degradation. Performance degradation grows with the number of processors as does the bus load. If the bus is lightly loaded, cache flushes can proceed without interfering with the activity of the processors that do not participate in the commit protocol, and in this case the performance degradation remains constant whatever the number of processors.

As stated in [12], the key issue to obtain good performance is to keep the frequency of recovery point operations independent from any architectural parameter. This is what the RSM approach attempts to do.

6 Conclusion

The architecture presented in this paper allows processor failures to be tolerated transparently, that is, without affecting the software being executed on the architecture. The only specific hardware component required is the RSM, and it is believed that the RSM can be implemented at a reasonable cost. The RSM copes with standard caches and cache coherency protocols, and this provides an advantage over the other approaches studied, such as CARER and Sequoia. The dependency tracking mechanism provided by the RSM allows shared memory to be provided to and used by the software. In contrast, the Sequoia system only permits memory sharing within the operating system, and requires complex software structures to ensure the correct semantics.

From a performance point of view, simulation results show that an architecture incorporating the RSM offers the best performance if compared with the Sequoia and CARER approaches. This is mainly due to the fact that recovery points are committed less frequently in our architecture than

in others. Moreover, the number of blocks to be copied in a commit operation is kept as low as possible by the fine-grained recovery protocol presented in section 4 together with the RSM-supported dependency tracking mechanism.

A prototype fault tolerant shared memory multiprocessor system is currently being constructed in an ESPRIT project called FASST (Fault-tolerant Architecture with Stable Storage Technology) [8] which involves the authors and several European partners both from industry and universities. The hardware configuration in FASST is similar to the architecture presented in section 4. For brevity, this paper has implicitly assumed that the RSM is composed of a single centralised board. In the FASST architecture, the shared memory will be composed of multiple RSM boards, to overcome the memory capacity limit and bandwidth of a single RSM board. While the physical design of a reliable and efficient RSM board is one of the challenging issues of this project, past experience in the actual building of stable storage boards [2, 4] provides confidence that this can be achieved at a reasonable cost.

This paper has just considered the case of parallel applications which consist purely of computation, with no input/output operations. Providing backward error recovery in the face of unrecoverable operations such as I/O is a further challenge which has not been addressed here. Continuing research is addressing this problem and the extensions to the recovery protocol necessary to provide the required abstraction of backward error recovery with both shared memory and other unrecoverable operations being executed by applications programs.

Acknowledgements

Some of the issues discussed in the paper have benefitted from discussions with FASST partners whose comments are gratefully acknowledged.

References

[1] Ahmed, R., Frazier, R., and Marinos, P. Cache-aided rollback error recovery (carer) algorithms for shared-memory multiprocessor systems. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems* (Newcastle, June 1990), pp. 82–88.

[2] Banâtre, J., Banâtre, M., Lapalme, G., and Ployette, F. The design and building of enchere, a distributed electronic marketing system. *Communications of the ACM 29*, 1 (January 1986), 19–29.

[3] Banâtre, M., and Joubert, P. Cache management in a tightly coupled fault tolerant multiprocessor. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems* (Newcastle, June 1990), pp. 89–96.

[4] Banâtre, M., Muller, G., Rochat, B., and Sanchez, P. Design decisions for the ftm : A general purpose fault tolerant machine. In *Proc. of 21th International Symposium on Fault-Tolerant Computing Systems* (Montréal, Canada, June 1991), pp. 71–78.

[5] Banâtre, M., Jégado, M., Joubert, P., and Morin C. Communicating Processes and Fault Tolerance : A Shared Memory Multiprocessor Experience. Research report 1649, INRIA, March 1992.

[6] Bernstein, P. A. Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing. *IEEE Computer* (February 1988), pp. 37–45.

[7] Davis, H., Goldschmidt, S., and Hennessy, J. Multiprocessor simulation using tango. In *Proc. of 1991 International Conference on Parallel Processing* (August 1991), pp. II 99–107.

[8] FASST Project Consortium. *The FASST Architecture: Overall Requirements and Specifications*, January 1992.

[9] Gefflaut, A., and Joubert, P. SPAM : a multiprocessor execution driven simulation kernel. Research report, INRIA, December 1992.

[10] Gray, J. *Notes on Database Operating Systems.*, vol. 60 of *Lecture Notes in Computer Science*. Springer Verlag, 1978 pp. 394–481.

[11] Horning, J. J., Lauer, H. C., Melliar-Smith, P. M., and Randell, B. A program structure for error detection and recovery. In *International Symposium on Operating Systems* (Rocquencourt, France, April 1974), pp. 171–187.

[12] Janssens, B., and Fuchs, W. Experimental evaluation of multiprocessor cache-based error recovery. In *Proc. of 1991 International Conference on Parallel Processing* (August 1991), pp. I 505–508.

[13] Jewett, D. Integrity s2: A fault-tolerant unix platform. In *Proc. of 21th International Symposium on Fault-Tolerant Computing Systems* (Montréal, Canada, June 1991), pp. 512–519.

[14] Joubert, P. *Conception et évaluation d'une architecture multiprocesseur à mémoire partagée tolérante aux fautes.* Phd. thesis, University of Rennes I, January 1993.

[15] Katz, R. H., Eggers, S. J., Wood, D. A., Perkins, C. L., and Sheldon, R. G. Implementing a cache consistency protocol. In *Proc. of 12th Annual International Symposium on Computer Architecture* (Boston, 1985), IEEE, pp. 276–283.

[16] Katzman, J. A fault-tolerant computing system. In *Proceedings of Eleventh Hawaii International Conference on System Sciences* (Honolulu (HA), January 1978), pp. 85–102.

[17] Larus, J. Abstract execution : A technique for efficiently tracing programs. *Software Practice and Experience 20*, 12 (December 1990).

[18] Lee, P.A., and Anderson, T. *Fault Tolerance : Principles and Practice*, second revised ed., vol. 3 of *Dependable Computing and Fault-Tolerant Systems.* Springer Verlag, New York, 1990.

[19] Lee, P.A., Ghani, N., and Heron, K. A recovery cache for the PDP-11. *IEEE Transactions on Computers C-29*, 6 (June 1980), pp. 546–549.

[20] Reuter, A. A fast transaction-oriented logging scheme for undo recovery. *IEEE Transactions on Software Engineering SE-6*, 7 (July 1980), pp. 348–356.

[21] Schneider, F. B. The fail-stop processor approach. In *Concurency control and reliability in distributed systems, Chapitre 13.* Barghava, 1987, pp. 370–394.

[22] Singh, J., Weber, W., and Gupta, A. Splash : Stanford parallel applications for shared-memory. Tech. Rep. CSL-TR-91-469, Computer Systems Laboratory Stanford University, April 1991.

[23] Sweazey, P., and Smith, A. J. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proc. of*

13th Annual International Symposium on Computer Architecture (Tokyo, June 1986), ACM/IEEE, pp. 414–423.

[24] THACKER, C. P., STEWART, L. C., AND SATTERTHWAITE, E. H. Firefly : A multiprocessor workstation. *IEEE Transactions on Computers 37*, 8 (August 1988), pp. 909–920.

[25] WILSON, D. The stratus computer system. In *Resilient Computer Systems* (1985), T. Anderson, Ed., pp. 208–231.

[26] WU, K., FUCHS, W., AND PATEL, J. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), pp. 231–240.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399