Communicating processes and fault tolerance : a shared memory multiprocessor experience

Michel Banâtre, Maurice Jégado, Philippe Joubert, Christine Morin

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1649

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

COMMUNICATING PROCESSES AND FAULT TOLERANCE : A SHARED MEMORY MULTIPROCESSOR EXPERIENCE

Michel BANÂTRE
Maurice JÉGADO
Philippe JOUBERT
Christine MORIN

Mars 1992

RR_1649

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Communicating Processes and Fault Tolerance: A Shared Memory Multiprocessor Experience *

Processus communiquants et tolérance aux fautes : une expérience multiprocesseurs à mémoire partagée

Michel Banâtre
Maurice Jégado
Philippe Joubert
Christine Morin

IRISA-INRIA
Campus universitaire de Beaulieu,
35042 Rennes Cedex (France)

Publication Interne n° 647 - Mars 1992 - 40 pages - Programme 1

Abstract

The concept of *backward recovery* is now well established as a means of restoring a consistent state of a fault tolerant system should some faults occur. In this paper, we consider a system of communicating processes mapped onto a multilevel execution support. A *shared memory* multiprocessor machine is assumed. Our interest is in tolerating the hardware faults that may occur during the execution of a concurrent computation. The machine provides a *hardware backward recovery protocol* based on a specialized memory device which tracks dependencies between the processors accessing shared data residing in memory. The transparency provided by the protocol is discussed considering successively the models of computation at the various levels of abstraction of the execution support.

Key words. Multiprocessor, fault tolerance, stable memory, backward recovery.

*This work has been partly supported by the FASST Esprit project.

Résumé

Le concept de *récupération arrière* est maintenant bien établi pour restaurer un état cohérent d'un système tolérant aux fautes lorsque des fautes se manifestent. Dans ce rapport, nous considérons un système de processus communiquants mis en oeuvre par un support d'exécution à plusieurs niveaux. Nous supposons une machine multiprocesseurs *à mémoire partagée*. Notre but est de tolérer les fautes matérielles qui peuvent se manifester lors de l'exécution d'une application parallèle. La machine fournit un *protocole matériel de récupération arrière* fondé sur une mémoire spécialisée qui note les dépendances entre les processeurs accédant les informations partagées de la mémoire. La transparence du protocole est discutée en considérant successivement les modèles de calcul offerts par les différents niveaux d'abstraction du support d'exécution.

Mots clés. Multiprocesseurs, tolérance aux fautes, mémoire stable, récupération arrière.

1 Introduction

The concept of *backward recovery* is now well established as a means of restoring a consistent state of a fault tolerant system should some faults occur [Randell 75]. Several algorithms have been proposed in the literature for providing backward recovery depending upon (i) the type of faults to be tolerated, (ii) the system characteristics, and (iii) the fault tolerance strategy.

In this paper, we consider a system of communicating processes mapped onto a multilevel execution support, the bottom layer being a shared memory multiprocessor machine. Our main interest is in tolerating hardware faults. The central idea of our work is to embed, as much as possible, the fault tolerance mechanisms within the hardware itself. A specialized *stable memory device* has been designed for that purpose. This work is the continuation of previous studies that we have been conducting in the field of hardware fault tolerance using stable storage technology [Banâtre 88, Banâtre 91b].

In a system of communicating processes, should a fault occur, the *recovery control protocol* must determine a set of process states which together constitute a consistent state of the system. Many recovery protocols assuming message passing communication have been proposed in the literature (see for instance [Wood 85, Strom 85]). In contrast, the recovery protocol implemented by the memory device of the architecture proposed in this paper relies on the fact that communication takes place through shared data. The memory device tracks directly the dependencies between the processors' references to the shared data.

Ideally, the fault tolerance mechanisms provided by the hardware itself would be sufficient to cope fully with the hardware faults thereby releiving the upper layers of the execution machine of handling fault tolerance issues. However, a realistic architecture is likely to embed non-recoverable objects such as i/o devices. This is also discussed in the paper when dealing with the operating system level of abstraction of the multilevel execution support.

The remainder of the paper is organized as follows. In section 2, a terminology and the basic principles of the recovery protocol are introduced. In section 3, the detailed design of a fault tolerant shared memory multiprocessor system implementing the protocol by hardware is presented. In section 4, we examine operating system issues. Considering a layered structure of the operating system, recoverability of each layer is discussed in turn. In section 5, related works are discussed. Some of the ideas proposed in the body of the paper are currently being explored within an Esprit European project (FASST) with other partners. The FASST project is presented in the last section.

2 A basic recovery protocol for processes communicating through shared data

In this section, we first introduce some definitions and background notions concerning backward recovery in a system of communicating processes. Second, we present a basic recovery protocol for processes communicating through shared data that will be used throughout the paper.

2.1 Definitions and background

Definitions A *recovery point* is *established* by a process at a point in time at which the state of the process is saved for possible regeneration in the event of recovery action. A process *commits* a recovery point when it no longer requires the capability to initiate recovery action to that point. The period of process activity between the *establishment* of a recovery point and the commitment to it is called the *process transaction* associated to that point. (Notice that the meaning of the word transaction here should be distinguished from the one which is usually given in transactional systems [Gray 78] where a transaction may refer to a consistency unit preserving some invariant of the system.) The most recently established recovery point of a process is said to be *active* or equivalently *current*. A recovery point which cannot possibly have recovery generated to it as a result of recovery action initiated anywhere in the system is said to be *discardable*. Part of the above definitions are borrowed from [Lee 90].

Model of computation We assume a model of computation of communicating processes where processes implement a succession of *non-nested* transactions, establishing a recovery point immediately on commitment to the preceding one. This is depicted in figure 1 where vertical bars denote the bounds of process transactions. The recovery control management offers the primitive $NewProcessTransaction(p)$ for committing the active recovery point and establishing a new recovery point for process p (for simplification purposes, initialization is not considered). Information flows between processes are assumed to be directed (unidirectional), and are represented by arrows in figure 1 when occurring between distinct processes. It is further assumed that all information sent out by a process is dependent on all information previously received by that process.

Definition For any two recovery points rp and rp' belonging to processes p and p' respectively, rp is a *direct propagator* to rp' if and only if information flows from p to p' while rp and rp' are the respective active recovery points of the two processes. As a particular case, a recovery point of a process is a direct propagator to the next recovery point of the same process. (For example, in figure 1, recovery point $B.2$ is a direct propagator to $A.2$ and $C.3$ while $A.1$ is a direct propagator to $A.2$.) For the commodity of the presentation, we will sometimes refer

to the propagator relation between process transactions instead of recovery points. A process transaction t_1 is a direct propagator to t_2 if the recovery point established at the beginning of t_1 is a direct propagator to the initial recovery point of t_2.

Definition For any two recovery points rp and rp' belonging to processes p and p' respectively, rp is a *propagator* to rp' if and only if the following holds : Either rp is a direct propagator to rp' or else, recursively, there exists a recovery point rp'' belonging to process p'' such that rp is a direct propagator to rp'' and rp'' is a propagator to rp'. (For example, in figure 1, recovery point $B.2$ is a propagator to $A.2$, $C.2$, $C.3$, and $D.2$.)

Definition We will often refer to the recovery *ancestors* and recovery *descendants* of a recovery point rp. An ancestor recovery point of rp is a propagator to rp. (For example, in figure 1, recovery points $A.2, B.2, C.2, C.3$, and $D.2$ are ancestors of $D.2$.) Conversely, if a recovery point rp' is descendant of rp, rp is a propagator to rp'. (For example, in figure 1, recovery points $A.2, C.2, C.3, D.2$, and $B.2$ are descendants of $B.2$.) As a particular case, notice that a recovery point is both ancestor and descendant of itself.

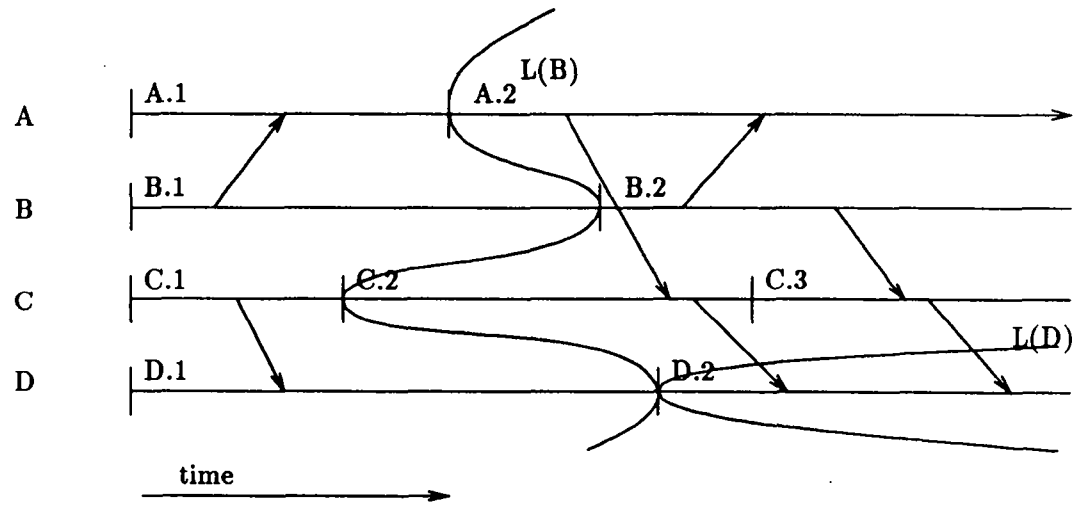

Figure 1: Communicating processes and recovery

Recovery in distributed systems

A recovery protocol must ensure that the system reverts to a consistent state in the event of one (or many) processe(s) initiating recovery action. As stated in [Wood 85], *a process initiating recovery must cause recovery of the descendants of its active recovery point* (including the active recovery point itself) in order to reach a consistent state. Another way to say this is that the

recovery control protocol must seek for a *recovery line* delimiting the boundary of an *atomic activity* [Randell 75].

Definition Traditionally, an atomic action conveys both the meaning of (i) an action which does not *interfere* with its environment, and (ii) a unitary action which has an *all or nothing* effect despite failures. For more precisions when needed, the first of these properties is referred to as *s_atomicity* (synchronisation) while the second is referred to as *u_atomicity* (unitary). The word atomic alone conveys both meanings.

Recovery lines are depicted as curved lines in figure 1. Not all the processes need be represented on a recovery line reflecting a situation where one process is not affected by recovery initiated by another. For instance, in figure 1, the recovery line ($L(D)$) associated to process D entails this single process only. Notice also that recovery may in general cause recovery of a process beyond its active recovery point. For instance, in figure 1, recovery initiated by process B will lead to recovery line $L(B)$ thus causing the restoration of process C to recovery point $C.2$ while $C.3$ is the active recovery point of C.

A process may belong to several recovery lines. For instance, in figure 1, process D belongs to both recovery lines $L(D)$ and $L(B)$. As far as recovery is concerned, we are interested in the recovery line which will lead to the minimal undo of computation ($L(D)$ in this example). This recovery line will be refered to as *the* recovery line of the process.

No information flow crosses from the inside of the recovery line to the outside but the converse is not the case. This requires that the recovery mechanism be capable, in case of recovery, to reproduce those informations entering into the domain delimited by the recovery line. For instance, should process D recover in figure 1, the information which has been produced by process C must be available after recovery takes place.

Recovery protocols fall into two broad categories namely *planned*, and *unplanned* [Lee 90, Randell 75]. Planned (or pessimistic) protocols bound the amount of system activity to be undone in case of recovery at the price of speeding down failure-free computation. In contrast, unplanned (or optimistic) protocols do not speed down failure-free computation but are prone to the so-called *domino effect* (cascading rollbacks) which in the extreme case could invalidate the whole computation in case of recovery.

A recovery protocol must provide the garbage collection of the discardable recovery points that are no longer required to provide backward recovery capability. While we do not expand on this issue in this report, it should be noted that this might be surprisingly difficult to implement when an unplanned approach to recovery is taken as illustrated in [Wood 85].

Communication in concurrent systems may take place by message passing or through shared data. While many recovery protocols based on message passing have been published in the literature such as [Wood 85, Strom 85], we did not find out a recovery protocol based on shared

data communication satisfying our requirements. This led us to the development of a protocol which is detailed in the next section. This concludes this brief introduction to recovery protocols in distributed systems, further informations can be found in the referenced bibliography.

2.2 The protocol principles

In this section we introduce a basic model of computation together with the characteristics of its recovery protocol which will be used throughout the paper. As before, we assume that processes implement a succession of non-nested transactions. A process may access a local state (the process registers) and a shared state represented in shared memory. Processes communicate through *shared data* of the shared memory. In other words, the significant events produced by a process consist in a trace of accesses to private registers and shared memory.

As a requirement of the protocol, the amount of recovery data must be bounded and limited to a single recovery point per process. Consequently, the protocol must adopt a planned approach to recovery and satisfy the following condition:

R : Recovery of a process must not go beyond its active recovery point

From the *R* condition, we infer: (i) the domino-effect is prevented, and (ii) it is easy to determine when a recovery point becomes discardable; a recovery point is discardable when it is committed.

In order to implement the *R* condition, the recovery protocol may implicitly establish a recovery point for a process. Notice that this contradicts somewhat the characteristics of the model of computation above since then the decision of establishment of a new recovery point may be not only explicitly performed by a process but also implicitly by the recovery protocol itself. However, this distinction will not be relevant in the text until section 4. In the following, we assume that a process implements a succession of non-nested transactions without further precisions.

In order to give insights in the protocol development, we will use an execution model based on traces in the following and take similar notations to those of [Best 82]. An execution U is modelled as a *sequence* $s_0 a_0 s_1 ... s_j a_j s_{j+1} ... a_{u-1} s_u$ where a_j ($0 \leq j < u$) denotes an action and s_j as well as s_{j+1} denote states. The state space S is defined as the set of mappings from the shared variables space to values. Each action a_j is an *s_atomic* action belonging to a component process $p_i (1 \leq i \leq n)$ denoted *component*(a_j). An action of p_i is either a write of a value e into a shared variable v denoted $w_i(v, e)$ or a read of a shared variable v denoted $r_i(v)$ or a commitment action denoted $c_i()$. The accesses of a process to its local state are not modelled since these are not relevant as far as communication is concerned. It should be noted that processes proceed asynchronously, and therefore flows of informations between them are *non-deterministic*. (This model may result from the implementation of an abstract model of computation not detailed

here; we may understand non-deterministic flows of information in the model proposed as an implementation property of the abstract model [Gries 81].)

Let s' belonging to S be an arbitrary initial state, the semantics m of the actions performed by a process is a relation $S \times S$ which can be characterized as follows:

- $s'\ m(w_i(v,e))\ s$ where $s(v) = e$ and $s(w) = s'(w)$ for all elements w of the state space different of v.
- $s'\ m(r_i(v)))\ s$ where $s = s'$. (The read action is supposed to deliver the value $s'(v)$ not modelled here.)
- $s'\ m(c_i())\ s$ where $s = s'$.

A *projection* of an execution U onto a component process p_i denoted $projection(p_i, U)$ is obtained by deleting from U all states $s_0 ... s_u$ and all a_j such that $component(a_j) \neq p_i$. We call U a *standard* (or correct) execution if it satisfies the two following properties [Best 82]:

- (P1) for all i, $projection(p_i, U)$ is a sequential control sequence of process p_i.
- (P2) (s_j, s_{j+1}) belongs to $m(a_j)$ for all j.

A standard execution is said to be *complete* if $projection(p_i, U)$ for all i is a complete control sequence of process p_i. While property (P1) captures the control aspect of an execution, property (P2) captures the semantic aspects with respect to data. (Further informations can be found in [Best 82].)

Let us turn our attention to recovery now. Should a process p_i roll back when an execution has reached the sequence U, the recovery protocol builds a new sequence W, by undoing the effects of all actions performed within the current transactions of the processes belonging to the recovery line of the process invoking recovery, from which post recovery computation will start. The post recovery computation catenated with W must be identical to a complete standard execution that would have taken place if recovery did not occur.

In order to give insight in the computation of the recovery line of the process invoking recovery and more generally in the necessary recovery actions to be taken, undoing the effects of a single rolled back process is modelled as the computation of an output *string* U' from an input string U. The string U' is such that $U' = [q_0]b_0[q_1]...[q_j]b_j[q_{j+1}]...[q_{u'-1}]b_{u'-1}[q_{u'}]$ where $[q_j]$ denotes chains of states, and b_j actions. As a particular case, notice that a sequence is a string. The string U' is obtained by:

1. Erasing from U the actions performed within the current process transaction of the rolled back process (but not the states), and
2. Appending if necessary a final state to the string denoting the effect of the state restoration applied to the variables of the state space whose values must be recovered by the protocol.

The final output U' of the recovery protocol (after having possibly executed the procedure above for several processes including the process invoking recovery) must be *equivalent* to a standard sequence W satisfying properties (P1) and (P2). This is defined in the following. Let $W = t_0 b_0 ... t_j b_j t_{j+1} ... t_{u'-1} b_{u'-1} t_{u'}$ such that (i) the string of actions b_j of W is identical to the string of actions of U', and (ii) the initial state t_0 of W is equal to the initial state $first(q_0)$ of U'. The string U' is said to be equivalent to the standard sequence W if:

(Q1) $last([q_{u'}])$ (final state of U') equals $t_{u'}$ (final state of W), and

(Q2) for each read action b_j, $last([q_j])(v)$ equals $t_j(v)$ assuming that v is the variable read by b_j.

If U' is equivalent to the standard sequence W as defined above, it is clear that the output of the recovery protocol is correct; the post recovery computation catenated with W will be identical to a complete standard execution that would have taken place if recovery did not occur since the rolled back processes will redo their computation from their current recovery points.

The previous characterization of a correct output of the recovery protocol gives a direct insight in the protocol development as discussed in the following.

Example. Consider the sequence $U = s_0 c_i() s_1 c_j() s_2 w_i(v,e) s_3 r_j(v') s_4 r_i(v') s_5$ and assume that process p_i initiates recovery. Let U' be the string obtained by erasing the actions performed within the current process transaction of p_i and appending a final state s_6 (result of a state restoration) such that $s_6(v) = s_0(v)$ and $s_6(w) = s_5(w)$ for all $w \neq v$. Precisely, $U' = [q_0] c_i() [q_1] c_j() [q_2] r_j(v') [q_3]$ where $[q_0] = [s_0], [q_1] = [s_1], [q_2] = [s_2; s_3], [q_3] = [s_4; s_5; s_6]$. The recovery line of p_i entails this single process since U' is obviously equivalent (as defined above) to the standard sequence $W = t_0 c_i() t_1 c_j() t_2 r_j(v') t_3$ where $t_0 = s_0$.

In practice, things might be more difficult that illustrated by the introductory example above since processes can be dependent. How the protocol deals with this situation is discussed below in a non-formal way considering in turn the so-called write read and write write dependencies.

Write Read dependencies

Let v be a variable written to within the current process transaction of p_i, this variable being firstly accessed later by a read action of p_j within its current transaction. Erasing only the write access from U will not be sufficient to produce a correct string U' since the read action of v by p_j would then not deliver the previous value written to v in U', and hence U' would not be equivalent to the standard sequence W as defined previously (the property (Q2) above would not be satisfied). Process p_i is in this case a direct propagator to p_j (or equivalently p_j is *dependent* of p_i) denoted $p_i \xrightarrow{wr} p_j$ meaning that rolling back p_i should cause a roll back of p_j. More generally, any process which reads a non-committed value written by p_i is wr dependent of p_i.

Recall that we do not want recovery of a process go beyond its active recovery point (the R condition). In order to ensure this, *the commitment of a process transaction will force the commitment of all its ancestors* (ancestors recovery points are referred to as potential recovery initiators in [Wood 85]). (If this were not the case, an ancestor initiating recovery might require some of its descendants to rollback beyond the current recovery point.) This is the overhead to pay for the planned approach which facilitates the computation of a recovery line in case of recovery.

Write Write dependencies

Let U' denote the string obtained by erasing from U all actions performed within the process transactions that are descendants of the current recovery point (as explained above) of process p_i invoking backward recovery. Let v be a variable which has been written to in U. If the last writing to v in U is not erased in U', the value of v in the final state of U' is correct but otherwise is not since U' would not be equivalent to the standard sequence W as defined previously (the property (Q1) would not be satisfied). Let $U' = [q_0]...w_i(v,e)...[q_{u'}]$ where $w_i(v,e)$ denotes the last writing to v in U' and assume that the last writing to v in U has been erased in U'. A correct string could be obtained by appending to U' the result of a state restoration reestablishing the value e of v and repeating this for all variables whose last writing to in U has been erased in U'. The difficulty here resides in finding out the value e within the whole history of the values taken by the variable v. How this is achieved is discussed in the following.

Definition A process p is said the *active writer* of a variable v if p has been writing to v within its current (active) transaction and v has not been subsequently written by other processes.

The protocol does not maintain the whole history of a variable but only both a *current* value and a *recovery* value. At commitment time of a process, the recovery value of a variable is replaced by the current value if the process is the active writer of the variable. Symmetrically, at rollback of a process, the current value of a variable is replaced by the recovery value if the process is the active writer of the variable. *In order to reestablish a valid final state of the string U' above, the protocol ensures that the last write action to v $w_i(v,e)$ of the string U' is committed and will thus restore the recovery value e of v.* This implies that a process committing its current transaction must force commitment of the active writers of the variables written within the committing transaction while rollback of a process transaction must cause the rollback of all transactions which have been writing to a variable whose active writer is the rolling back transaction. A simple way to achieve this goal is to record a dependency $p_i \xleftarrow{ww} p_j$ when a variable is successively written by two processes p_i and p_j within their current process transaction; this dependency will have the same effect as the $\xrightarrow{wr}$ dependency as far as commitment and roll back are concerned. (Notice that the $\xrightarrow{ww}$ relation is a predecessor

relation as opposed to the wr relation which captures a successor relation.) This concludes the presentation of the protocol (a formal proof would be desirable but is not discussed in this paper).

Summary

In summary, a $p_i \xrightarrow{wr} p_j$ dependency is recorded when p_j reads a variable whose active writer is p_i while a $p_i \xleftarrow{ww} p_j$ dependency is recorded when p_j writes to a variable whose active writer is p_i, p_j becoming then the new active writer. Rollback of a process p_i will induce a rollback of its descendants according to both relations $\xrightarrow{wr}$ and $\xrightarrow{ww}$; the recovery values of the variables whose active writer is a process member of this group, will be reestablished. Commitment of a process will induce the commitment of all its ancestors according to both relations $\xrightarrow{wr}$ and $\xrightarrow{ww}$; the recovery values of the variables to which processes member of this group are the active writer will be logically replaced by their current values.

3 Design of a fault tolerant shared memory multiprocessor machine

In this section, we present the salient features of an original fault tolerant shared memory multiprocessor machine. This machine implements the recovery protocol described previously, providing recoverability to *communicating processors.*

The general architecture which is depicted in figure 2 mainly consists of processing elements and of a shared Stable Transactional Memory (STM). As it is usually the case in shared memory multiprocessors the processing elements access the shared memory through (optional) *private caches* holding the most recently referenced memory locations [Smith 82]. We assume that a processor of the multiprocessor machine executes instructions which can access a *local* state (the processor's registers) and a *shared* state. The shared state is represented as a set of memory *cells* of the STM. The STM is a non-volatile fail free random access memory and offers the notion of *processor transaction* to each processor accessing it.

Definition A processor transaction is an *u_atomic* sequence of memory reads and writes (as defined in section 2) performed by a processor which may be undone before reaching commitment so as to tolerate processing elements failures.

The architecture has been designed so as to impose minimum requirements on non-STM hardware. Ideally, it should be possible to plug an STM board into an off the shelf shared memory multiprocessor to make it fault tolerant.

Most ideas of the architecture proposed here can be traced back to the reference [Banâtre 90]. However, in contrast to this reference which proposes a limited form of process transaction

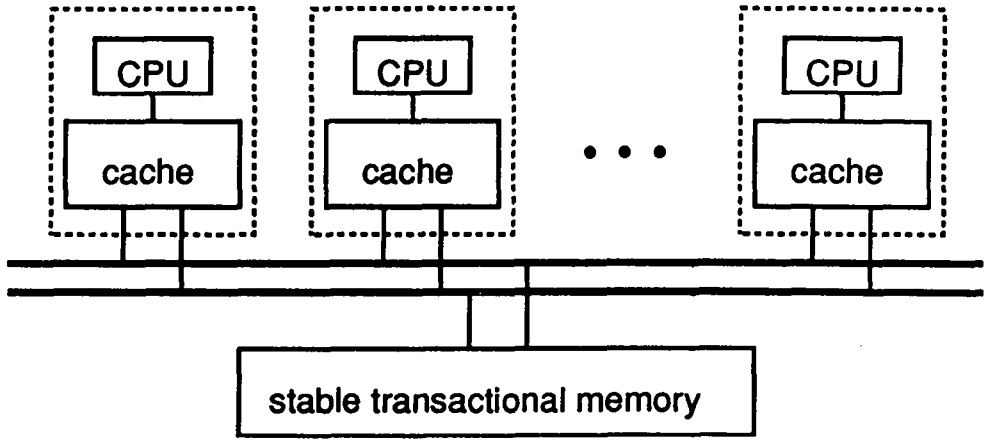

Figure 2: The shared memory architecture

implemented by specialized caches, the architecture discussed here implements the notion of processor transaction and can accomodate standard cache behaviour as discussed further in the text.

From a hardware point of view, a processor transaction splits into three steps: (i) a prelude for starting the transaction, (ii) a body of read/write commands issued by the processor, and (iii) a postlude for terminating the transaction. In the following, we examine these issues. First, we will assume that processors access the STM without caches. Second, we will concentrate on the complexity added by the caches themselves. We conclude the section by a brief account of performance evaluation and some comments.

3.1 The basic case: no caches

The STM provides both a recovery and a current value for each cell accessed. Recovery values of the data are not accessed during the transaction body so as to be capable of undoing the effects of the transaction. Apart from fulfilling the processors' memory requests, the STM has to *track dependencies between processors* when they share data according to the recovery protocol as discussed in section 2. That is, a dependency has to be recorded by the STM in two cases :

- Whenever a processor P_i reads a cell previously modified by processor P_j within its current transaction a $P_j \xrightarrow{wr} P_i$ dependency is created.
- Whenever a processor P_i modifies a cell previously modified by processor P_j within its current transaction a $P_j \xleftarrow{ww} P_i$ dependency is created.

Since the *wr* and *ww* dependencies are treated in the same way by the protocol, it is sufficient to record a single type of dependency. One important assumption for dependency tracking is that the STM is only connected to the bus of the architecture and does not directly observes

reads and writes performed by processors but rather their reflection on the bus; dependencies are tracked by *snooping* bus informations. In order to record dependencies, when an access to a cell is made by a processor, the STM needs to know:

- The identifier of the processor doing the access. This is achieved by fitting each processing element board with a unique identifier which is passed on the bus whenever this processing element puts an address on the bus. Most modern busses, like the SPARC MBus [Kitagawa 91], directly provide this information on their address lines.
- The type of access (read or write). This is directly provided to the STM by its memory interface.
- The identifier of the processor that is active writer of the cell if any.

The termination of a processor transaction obeys a simple distributed *two-phase commit* [Gray 78] protocol. Processors are participants while the STM is the coordinator of the protocol. In contrast to 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 in charge of committing data. When a participant issues commitment, after having flushed its registers into the STM if necessary, it sends a *do_commit* command to the STM and waits for an interrupt meaning that commitment is terminated and that the processor can resume processing.

Upon receiving a *do_commit* command, the STM scans the dependency information and informs all dependent processors. A dependent processor can then flush its registers into the STM if necessary and must acknowledge the first phase of the protocol. When all acknowledgements from the participant processors have been received, the STM enters the second phase of the commit protocol. During the second phase, the recovery values of the cells which active writers belong to the *group* of dependent processors are changed to their current values. The dependencies of the dependency group are broken and a new processor transaction is then started for each processor belonging to the group.

Let us consider now the implementation in greater detail.

Servicing read and write commands

The STM actions are better 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 STM, most of the work is concerned by dependency management. We assume that dependency data is stored in a $n*n$ boolean matrix M; n being the maximum number of processor transactions allowed to access the STM concurrently. Typically, n equals the maximum number

of processors in the architecture. A matrix item $M(i,j)$ set to true means that processor P_i is dependent of P_j.

While read and write commands refer to STM cells, the STM itself may record dependencies on a bigger granularity. In the following, we assume that the STM physical space is divided into a set of contiguous *blocks* of identical size which contain a power of two cells. Each block consists of (i) a current value, (ii) a tag field containing either the idendity of the active writer to the block (if any) or the *nil* value, and (iii) a recovery value. A simple implementation of the STM data would consist of two banks of memory, one bank containing the current values together with the tag fields while the other bank contains the recovery values.

Basically a read to a cell c will compute the target block b, record a dependency with the active writer processor of the block if any in the matrix M, and will deliver the current value of the cell. A write to a cell c will compute the target block b, record an opposite dependency with the active writer if any, change the active writer of the block, and will update the current value of the cell within the block b.

First phase of the commit protocol

The first phase of the commit protocol is initiated by a processor issuing a *do_commit* command to the STM. This communication can be implemented by using a dedicated address in the address space as this is the case with memory mapped I/O devices. An access by the processor at this address will trigger the requested action. (Other addresses may be used for different purposes such as initialization or self-testing.)

Upon receiving a *do_commit* command from processor P_i, the STM has to scan the dependency information it has recorded during the body of P_i's current transaction to determine the *group* of processors which are required to commit atomically with P_i according to the recovery protocol. Once dependency information has been computed internally, the STM broadcasts on the bus a bit vector conveying the group of dependent processors within the memory cycle of the *do_commit* command like in an usual memory read cycle. Interrupt or message passing facilities provided by the bus used in the architecture, may also be convenient for this purpose. Dedicated logic on each processor board snoops this bit vector and checks whether the processor it is attached to has to participate to the group simply by masking the broadcast vector with its own processor identifier. If so, a high priority *prepare_to_commit* interrupt is sent to the processor which in response to it, must also issue a *do_commit* command 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 of the initial commit command and the acknowledgements of the dependent processors, some new dependencies may have been created for the STM carries on servicing read and write commands from processors that are not blocked waiting for the end of the commit protocol. It may also occur that a processor not already part of the

group decides on its own to commit its current processor transaction and sends a *do_commit* command to the STM. This processor is added to the (current) group as well as all the processors dependent from it. This mechanism provides a cheap way for implementing multiple concurrent processor groups.

One crucial point is that group computation has to be atomic with respect to read and write accesses. If not, dependencies might be created during group computation resulting in an incomplete group being validated by the STM. A simple way to enforce this property is to serialize the group computation and read write accesses as it is the case here since the group computation is performed within the memory cycle corresponding to the *do_commit* command.

The dependency group computation algorithm is given, in C dialect, in figure 3. Under the assumption that the number n of processors can be encoded within an integer variable, the matrix M is implemented by an integer array where each array element is considered as a bit vector indexed by a processor identifier. The algoritm uses bit vector operators.

Upon reception of a *do_commit* command, the STM executes the *do_commit* procedure. Let *group* be the bit vector denoting the processors member of the dependency group and *do_commit_received* be 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 STM automaton into the commit state implementing the second phase of the commit protocol if the following condition is verified :

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

The condition Q expresses that all processors belonging to the exact group of dependent processors have completed the first phase.

If Q is not verified, two cases arise. First, some processors that have already been informed that they are group members have not completed yet the first phase in which case the STM must wait for the reception of their *do_commit* command. Second, some new processors became group members since the last computation of *group* and thus must be informed. Notice that the dependency computation algorithm must avoid interrupting a given processor several times.

There are many ways to devise an algorithm satisfying the previous requirements. In figure 3, a simple solution is given. The algorithm checks for 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 $O(n)$. The *do_commit* procedure being executed at most n times, the first phase of the commit protocol is thus $O(n^2)$.

Second phase of the commit protocol

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

```
    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:
    do_commit(i)
    int i;                  /* processor id */
    /* the  processor i is willing to commit or acknowledges a request of
       the STM 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 ((group & (1<<j)) != 0)     /* if processor j is a member of the group */
                    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) onto the bus */
                       group = dependent_members;
                    }
     } /* do_commit */
```

Figure 3: Computing a dependency group

1. Replace the recovery values of all the blocks whose active writers belong to the dependency group by their current values and set the active writer field of those blocks to the *nil* value.

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

3. Broadcast a *commit_done* vector on the bus conveying the identities of the processors belonging to the dependency group so as to restart the processors waiting for the end of the commit protocol.

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

A straightforward algorithm would (i) perform those actions sequentially within the same memory cycle of the last *do_commit* command, and (ii) would implement step (1) above by repeatedly checking every block of the STM and copying the current value to its recovery counterpart if necessary. Some comments are in order. First, all processors participating to the commit protocol are blocked during the three steps above and other processors may also be blocked during this period waiting for a bus grant. Second, the commit time is directly proportional to the total size of the STM.

Many refinements of this algorithm can be made. We examine some of them below:

- Firstly, the time needed to perform step (1) above may be considered prohibitive and can be improved in at least two ways:

  1. We may maintain a linked list per processor chaining the memory blocks the processor is the active writer to. The time needed to perform step (1) will then be proportional to the number of blocks to be committed by traversing the list but at an extra storage overhead in the STM, however.

  2. Since the operations performed on each block are independent we may also envisage to have step (1) performed by several parallel memory units hence bounding the time needed for step (1) to that needed for a single unit.

- Secondly, we may envisage to free the bus and restart the processors waiting for the end of the commit protocol at the beginning of the second phase before actual step (1) takes place. This will reduce the blocking period of the processors but ensuring the atomicity of the second phase would then be more difficult to implement.

Failure detection and recovery

We assume that each processor is *fail-stop* [Schneider 87]. Fail-stop processors are mandatory given the behaviour of the STM which does not detect incorrect processor accesses. This not a severe constraint on the architecture for fail-stop processors are common practice in the field of

hardware fault tolerance. In case of failure, the processor ideally will signal a failure interrupt on the bus which will be caught by a live processor triggering then the *do_rollback(i)* command of the STM where i denotes the processor that failed. If we assume that a processor may fail without informing the outside, it would then be necessary to introduce a time-out mechanism in the commit protocol so as to ensure that commitment is achieved within a finite time (the time-out could be managed by the STM or the processors). In the whole text we take the former assumption. We also assume that the other components of the architecture are fail-free.

Upon reception of the *do_rollback(i)* command (in the service state), the rollback group of descendants of i is computed recursively by the STM. In a manner similar to the second phase of the commit protocol, the current values of the blocks modified by the processors members of the group are substituted by the recovery values. A bit vector conveying the dependent processors is broadcast onto the bus so that the dedicated logic attached to each processor may interrupt its processor by a *roll_back_done* interrupt if needed. The dependencies are broken (precisely, $(\forall i : i \in rollback\_group : M[i] = 1 << i)$), and the STM reenters the service state. Rolling back a transaction 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 while rolling back is demanded. Since a same processor may belong both to a rollback group and a commit group, it is necessary to check for this at the end of the rollback procedure. This is briefly sketched in the following. Members of the rollback group are removed from the commit group and the *do_commit_received* set if any. If the remaining commit group is not empty, the *do_commit* procedure of figure 3 is executed taking as argument one member of the commit group.

Summary of a processor activity and termination of the commit protocol

Similarly to the STM, the activity of a processor can be expressed by a finite state automaton which is depicted in figure 4. The states names should be self explanatory.

The labelled transitions of the automaton have the following meanings:

(1) The processor willing to commit triggers the commit protocol.

(1 bis) The processor receives a *prepare_to_commit* interrupt request from the STM.

(2) The processor issues a *do_commit* command to the STM.

(3) The processor receives a *commit_done* interrupt from the STM.

(4) The processor receives a *rollback_done* interrupt from the STM.

(fail) The processor fails.

If a processor is in *waiting* state (within an *uncertainty* period [Bernstein 87]) after having sent a *do_commit* command to the STM, the processor will eventually be awakened by a *commit_done* interrupt or a *rollback_done* interrupt. The former case refers to the termination of

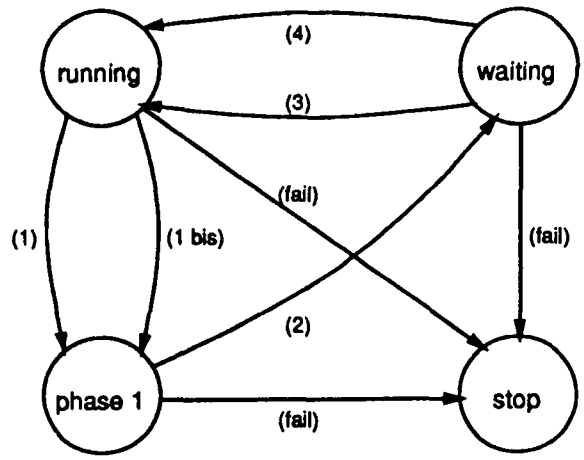

Figure 4: Processor activity automaton

the commit protocol in case of no failures. Termination is obvious from figure 3 assuming that a processor acknowledges a *prepare_to_commit* request within a finite time since the number of processors is bounded. The latter case refers to failures which according to our assumptions result in the STM being informed of the situation so as to take the appropriate actions to rollback and inform the processors.

3.2 Cache coherence protocols

In this section we assume that processors perform their memory accesses through private *coherent* caches. The complexity of adding caches to the architecture compared to the previous section resides in the dependency tracking work. We examine below the influence of the *cache coherence protocol* on the dependency tracking.

A cache system is said to be *coherent* if every read of a memory location returns the value most recently written to that location [Censier 78]. In a shared memory multiprocessor where processors access shared memory through private caches, there can be potentially as many copies of the same memory location as there are processors in the architecture. Inconsistencies may occur when several processors access writable shared data. When a data is modified, its modification has to be reflected into all the other caches which hold a copy of the data. The unit of information managed by the caches is referred to as a *line* while a processor access refers to a cell. Typically a cache line size ranges from 4 to 32 cells.

The protocols for avoiding cache inconsistencies are often referred to as *cache coherence protocols* (the term *cache consistency protocols* can also be found in the litterature). Most hardware cache coherence protocols proposed so far rely on the fact that bus traffic can be monitored (*snooped*) by all caches. Snoopy caches maintain a *tag field* stored along with each loaded line to indicate the line state in each cache. The tag field generally encodes whether

the line is modified with respect to shared memory and whether the line is 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 a data potentially loaded into other caches is updated. All caches snoop these invalidation messages and invalidate their corresponding entry. A further read miss will cause the up to date data to be loaded into the cache.
- *Write Update* protocols broadcast the new value whenever a data potentially loaded into other caches is updated. All caches snoop the write and update their copy of the data accordingly.

These protocols mainly differ by their relative hardware cost and performance in terms of bus traffic generated to maintain coherence (see [Archibald 86] for a survey and performance evaluation of those protocols). For the sake of simplicity, we only examine in the following, the Berkeley protocol representative of the write invalidate family.

The Berkeley coherence protocol [Katz 85] was originally designed for the SPUR workstation at the University of California at Berkeley. This protocol introduces the notion of ownership of a line, the owner being responsible for writing the line back to main memory as well as for supplying directly the line to any other cache requesting it. In this protocol, the tag field of a memory line of a given cache can be in one of the four following states (line states are described according to the terminology found in [Sweazey 86]) :

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 5 depicts the state transition diagram for the Berkeley protocol.

Recall that the STM maintains dependencies on blocks. In contrast to the previous section, where the block granularity could be as small as a STM cell, one must notice that the STM must record dependencies on at least a line size granularity since a line is the minimal unit of

transfer on the bus. In the following, we examine the operations performed by the protocol and the various actions taken by the STM so as to track the dependencies when a processor performs respectively a read miss, a write hit, and a write miss. (Notice that, a read hit does not generate any action on the bus and thus is not handled by the protocol.)

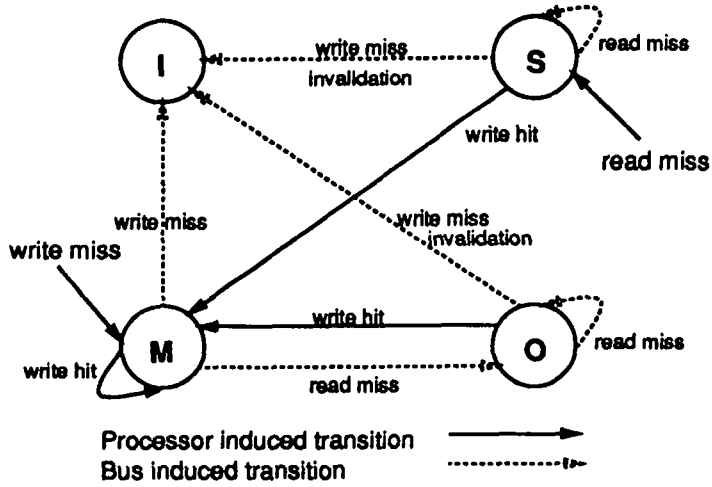

Figure 5: Berkeley state transition diagram

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 comes 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 $P_j \xrightarrow{wr} P_i$ dependency is created. As far as dependency management is concerned, no distinction is made whether the requested line comes from another cache or from the STM.

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 5). All other caches invalidate their copy upon matching the line address. The line state is changed to M in the originating cache.

The invalidation signal is snooped by the STM. If the corresponding block has no active writer, P_i becomes its active writer. Otherwise, let P_j be the active writer, a $P_j \xleftarrow{ww} P_i$ 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 STM 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 $P_j \xleftarrow{ww} P_i$ dependency is created 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 STM cannot detect a further read on a different cell of the line because it would not generate any bus traffic (see figure 5). So, a $P_j \xrightarrow{wr} P_i$ dependency 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 STM adopts a conservative approach by creating dependencies which are not strictly required by the protocol to preserve the coherence of processor checkpoints.

It should be noted that the STM must keep pace with the information exchange rate on the bus due to the cache coherence protocol. If this were not the case, the STM might miss some dependencies 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. Similar principles to those discussed above in the framework of Write Invalidate protocols also apply to Write Update protocols such as the Firefly protocol [Thacker 88] which we do not detail here for space reasons.

In summary, no special purpose caches or coherence protocols are needed in the architecture, which can accomodate standard cache behaviour; the STM performing dependency tracking by snooping the bus traffic. This is a notable difference with other proposals for fault tolerant shared memory multiprocessors [Bernstein 88, Wu 90, Ahmed 90] as discussed in section 5.

The termination of a transaction 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 STM, the processor flushes its cache. Similarly, a transaction roll back must cause a cache invalidation. So far, we have implicitly assumed a single level cache hierarchy. Extension to a primary on-chip cache and secondary cache hierarchy is straightforward since a primary write back cache can also be flushed.

As a final comment, it should be noted that cache flushing might be a resource (bus) consuming operation. This is one of the reasons why we strived for designing a fine-grained recovery protocol in section 2 avoiding to keep track of unnecessary dependencies so as to diminish the cost of commitment. Notice for instance that the simple strategy of committing all processors (global checkpointing) would not be acceptable from a performance point of view in general although this strategy has the net advantage of avoiding the need of keeping track of dependencies all together.

3.3 Performance evaluation

In order to evaluate the performance of the architecture discussed above, a simulation model has been built. The model allows to vary many parameters such as the cache associativity degree and size, the bus and memory timing or the cache coherence protocol. The simulator uses memory address traces produced by instrumented shared memory applications executing concurrently to the simulation. Instrumentation of object code is done at compile time. In order to evaluate the overhead induced by the backward recovery protocol in the non-failure case, performance figures are compared with those obtained by the same architecture when standard shared memory with no recovery capability is used.

Our results reveal that the performance degradation is reasonable and depends on (i) the committing frequency (ii) the time needed by the STM to perform the second phase of the commit protocol, and (iii) the amount of bus traffic generated by each application. For instance, initial results with SPLASH scientific application package [Singh 91] show that the overhead induced by the recovery protocol would be less than 10 percent under the Berkeley cache coherence protocol with a commit frequency of 1000 commit per second. The detailed results will be published elsewhere.

3.4 Comments

While the physical implementation of the STM board is not the subject of this paper, past experience in the actual building of stable storage boards [Banâtre 88, Banâtre 91b] makes us confident that this can be achieved at a reasonable cost.

In the previous sections, we have been implicitly assuming that the STM was composed of a single centralized board. In some environments, it might be necessary to scale the memory capacity of the architecture by adding multiple STM boards to the architecture so as to overcome the memory capacity limit of a single STM board. Another advantage of having multiple memory modules is to increase memory bandwitdh by using a *split transaction bus* which can handle concurrent memory requests on multiple modules. The implementation of the two-phase distributed commit protocol in this case is a matter for further research.

4 Operating system issues

The role of an operating system is to control the resources of the machine and to provide the base upon which the application programs can be written. Two basic structuring principles are applied in modern operating systems. First, the system may be organized as a hierarchy of layers, each one constructed upon the one below it. Second, system services may be provided as a set of communicating processes. To request a service, a process (referred to as a *client*) sends

a request to a *server* process which then does the work and sends back the answer.

Both of these principles are applied in the following figure which depicts the structure proposed for the operating system controlling the machine described in the last section. Level L_0

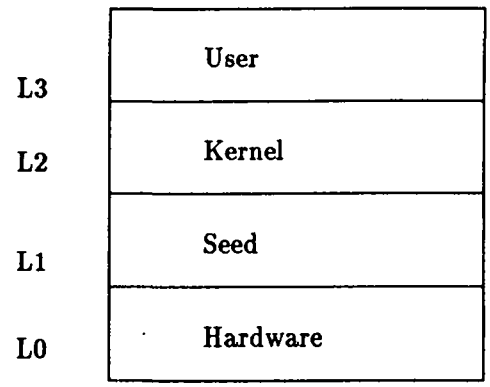

Figure 6: Structure of the operating system

denotes the hardware (section 3). Level L_1, referred to as the *seed* deals with the basic process management. Above the seed, the kernel services (L_2) can then be structured as a set of communicating processes. Finally, level L_3 deals with users which may request the services of the kernel from their application programs. In the following, we discuss the design principles of these layers with particular emphasis on recoverability but first consider the recovery provision for the basic model of computation as introduced in section 2.

4.1 Providing recoverability for the basic model of computation

Recall that in the basic model of computation, a process may access a local state (process registers) and a shared state which we assume to be represented in the STM. We examine how recoverability can be provided for this model considering the simplifying case where a separate processor would be dedicated to each process. (The realistic case where the multiprocessor may support the execution of an arbitrary number of processes competing for a limited number of available processors is examined further.)

Recoverability of a system of communicating processes responding to the basic model is simply provided by *mapping a process transaction to a processor transaction.* When a new process transaction is started (following an explicit request of the process or an implicit action of the recovery protocol), the local state of the process (the registers of the processor executing the process) is written into memory, the cache if any is flushed into the STM, and the current (active) processor transaction associated to the processor executing the process is committed. The processor begins a new processor transaction.

The facilities offered by the STM are almost sufficient themselves for implementing correctly

the model of computation defined previously and is very adequate for tolerance to hardware faults (processor failures). When a processor failure occurs, the current transaction is rolled back and the process can safely restart its computation on another processor after having loaded the process local state from the STM into the registers of the new processor allocated to the process. The set of processors forced to roll back their current transaction due to the dependency protocol must also load their registers with the values safely stored in the STM on the last transaction commitment before pursuing forward activity. The other processors are not affected.

In summary, the fault tolerance mechanisms of the STM are (almost) sufficient to cope fully with processor failures so as to make these failures *transparent* to the processes obeying the basic model of computation.

Our conjecture is that this is also the case for any abstract model of computation that can be *mapped* onto this basic model. A rough argument of this (conjectured) property, based on the abstract data type theory might be as follows. Consider a program P responding to an abstract model of computation, starting in an initial abstract state s and terminating in a final abstract state f denoted $P(s) = f$. Assume that P is mapped to a program P_0 responding to the basic model of computation such that $P_0(s_0) = f_0$; s_0 and f_0 denoting respectively the initial and final concrete states of P_0. Let *abs* be the abstraction function from the concrete domain onto the abstract domain. Assume that a correct mapping satisfies the following predicate: $((P_0(s_0) = f_0 \wedge P(s) = f \wedge abs(s_0) = s) \Rightarrow abs(f_0) = f)$. Should a component process of P_0 be rolled back due to a processor failure, the recovery protocol constructs a program P_0' equivalent to P_0 (section 2) leading to the same final concrete state f_0. It is then easy to see that the abstract program P is not influenced by the recovery actions since the abstract final state of P will be the same. (Clearly, a more formal proof of this property would be desirable but is not tackled in this paper.) This property will be particularly relevant in this section where we consider the implementation of operating system layers as exemplified by the implementation of the basic synchronisation primitives discussed in section 4.2.2.

4.2 The seed

The role of the seed is to deal with the basic process management and to provide higher layers with a useful model of communicating sequential processes. In addition to the basic model of computation introduced in section 2, we require that (i) the seed offers abstract synchronisation and communication primitives, and (ii) allows for the dynamic creation and deletion of processes. The seed hides machine-dependent features (e.g. interrupt handling) and thus provides a machine independent interface which facilitates the portability of kernel services. The bulk of memory management is not considered as being part of the seed but as a kernel service running at a higher layer. This is discussed in section 4.4.

Conceptually, the seed layer has many similarities with micro-kernels that have been proposed

in the operating system field such as Chorus [Rozier 88] or Mach [Baron 88] and could thus be termed micro-kernel. Since our interest is in design principles, we will not attempt, however, to compare in detail the seed, as discussed in the following, to existing micro-kernels.

4.2.1 Virtualization of the processor resource

Dedicating a physical processor to each process (as discussed previously) is not realistic in a classical multiprocessor architecture where the number of available processors is limited, and therefore our first step is to virtualize this resource. A *virtual processor* is allocated to each process.

Time sharing is a well known technique to virtualize the processor resource. On a monoprocessor, the physical processor is allocated to each virtual processor for a time quantum. Virtual processors may be managed by a short term *scheduler* according to a round robin discipline. This same technique easily extends to a multiprocessor. We assume, as it is generally the case, that a virtual processor may be mapped to different physical processors during its activity.

The most straightforward implementation of process recoverability in this context is to conceal the scheduling activity and provide the notion of *virtual processor transaction.* A single active virtual transaction is associated to each virtual processor; a process transaction is mapped to a virtual processor transaction. In the following, the word transaction (alone) stands for virtual processor transaction.

Consider now the scheduler design in greater detail. We assume that a zone of the STM is allocated to each process for *stacking* private data. We also assume that once a process has consumed its time quantum on a processor, some clock device sends an interrupt to the processor. This has the effect to copy the local state of the process on top of its private stack, and triggers the execution of the scheduler. Symmetrically, returning from the scheduler has the effect to pop the local state at the top of the current stack into the processor's registers. We assume that a *context* is allocated at a fixed address in the STM for containing the process local state together with a link field used for list management. An array *active* such that *active*[k] refers to the context of the active process on processor k is maintained by the scheduler. A single list *ready* protected by a global lock chains the contexts of the processes that are not active. When the scheduler is entered by processor k in response to the event that the active process on processor k has consumed its time slice, the active process is descheduled and chained at the tail of the *ready* list to the benefit of the head process of the ready list which is removed from the list and made active. In other words, the set of processes contained in the system are the processes pointed to by the *active* array and those chained in the *ready* list, this set is maintained as an invariant by the scheduler if we ignore the dynamic creation/deletion of processes.

Without further constraints on scheduling, a processor transaction will embed activities belonging to distinct transactions, and a transaction will be mapped to several processor trans-

actions. However, for the sake of simplicity of this presentation, we propose to commit a transaction each time a virtual processor is allocated a physical processor by the scheduler. The basic steps performed by the scheduler entered on processor k by process p are described in figure 7.

1. Save the registers' values residing on top of the stack of p into the context of p.
2. Commit the current processor transaction.
3. $\{active[k] = p \ and \ ready = q + X\}$ Schedule a new process q $\{active[k] = q \ and \ ready = X + p\}$.
4. Begin a processor transaction.
5. Install the context of q on top of the stack of q.
6. Return from the scheduler.

Figure 7: The short-term scheduler: specifications

Consider the treatment of a failure of processor k. For now, we assume that this event cannot occur while performing the scheduling sequence above (this hypothesis will be relaxed further).

Let p_i denote a processor member of the set of the rolled back processors due to the failure of processor k. This set is automatically determined by the hardware recovery protocol. Basically, recovering from a processor failure boils down to the following actions:

1. The context pointed to by $active[p_i]$ must be loaded into the registers of processor p_i. Alternatively, this context could be inserted into the ready list and the dispatching of a new process be done on p_i. This action can be performed by processor p_i itself.

2. The process which context is pointed to by $active[k]$ must be inserted into the ready list. This action must be performed by a processor *elected* out of the remaining live processors (we do not detail here such an algorithm). The process will then be dispatched on another live processor by the scheduler.

One can notice that the failure of a processor does not affect all the processors of the machine but only those that are dependent on the failing one. Yet, almost all the work is directly achieved by hardware.

Logically, the data structures maintained by the scheduler itself are not part of the shared state of a process given our basic model of computation. How should recoverability provided for those objects? A first possibility is to make those data structures not implicitly hardware recoverable, recoverability being then achieved by explicit *forward recovery* [Randell 78]. A second possibility is to make those structures also implicitly hardware (backward) recoverable. We illustrate the second possibility in the algorithm of figure 8.

Consider now the case where processor k may fail while performing the scheduling sequence (3.1-3.5). Fortunately, this is equivalent to the situation where the failure occurs outside of

```
(1)         pop(stack[active[k]], context[active[k]])
(2)         NewProcessorTransaction
(3.1)test:  test_and_set (val, GlobalLock)
(3.2)       if val <> 0 then goto test fi
(3.3)       PutTail(ready, active[k])
(3.4)       RemoveHead (ready, active[k])
(3.5)       GlobalLock := 0
(4)         NewProcessorTransaction
(5)         push(stack[active[k]], context[active[k]])
(6)         return
```

Figure 8: The short-term scheduler: implementation

the scheduler (discussed above) if we can guarantee that the section (3.1-3.5) is an atomic action. Bracketing this compound operation within the primitives *NewProcessorTransaction* is not sufficient to ensure the atomicity property since implicit commitments due to the recovery protocol may occur while performing the action. However, in this particular case, within the seed itself where the programmer has a control over the implicit commitments performed by the recovery protocol, a cheap way to ensure the atomicity of the above action is to differ the treatment of any commit request (received from the outside) until the *NewProcessorTransaction* primitive is encountered. Naturally, the differing period should be short, which is the case here, since some processors are blocked waiting for the end of the commit protocol.

Now, we can also observe that a commit request cannot happen while performing section (3.1-3.5) since the processor has committed before entering this section which matches a processor critical section. By definition, this section can only be entered by a single processor at a time, and the processor within the section cannot be a potential recovery initiator of another processor willing to commit.

As a final remark, notice that the *NewProcessorTransaction* primitive of figure 8 does not need triggering a register flush but only a cache flush (if any) followed by a call to the *do_commit* command of the STM.

4.2.2 Basic synchronisation primitives

We assume that processes can exchange *messages* through *ports*. To simplify the presentation, we assume that (i) a port may retain the memory of an arbitrary number of messages, and (ii) a message may contain a variable size collection of data, although a particular implementation may restrict these hypothesis. Sending a message m to a port p (*send (p, m)*) is assumed to be an asynchronous operation while receiving a message from a port (*receive (p, m)*) is blocking if the port is empty.

Recoverability for this model of computation can be provided in almost the same manner as explained for the basic model. What is required is that a dependency between a process sending a message and the process receiving it later on must be recorded. But, as we are assuming that ports are represented in the STM, the hardware will automatically does this since sending and receiving a message are operations which will access common STM data.

It should be noted that the default dependency tracking policy provided by the hardware may lead to more dependencies than strictly required by the model of computation due to the access to the concrete data representing the abstract model of computation. For performance reasons, one may be willing to have an explicit control on the dependencies in order to avoid some "overhead" dependencies that might occur due to the default policy. This suggests that the STM should provide the possibility of escaping the default policy to the benefit of an explicit programmed dependency tracking policy. Inhibiting the default policy could be achieved by a special STM primitive (this primitive should be reserved to the priviledged mode of operation of the processors for protection purposes).

We will not go into implementation details of the basic synchronisation primitives here but just mention that the scheduler algorithm discussed above is a sound basis upon which to build them.

4.3 Standard vs non-standard processes

So far, we have been considering that a process may access a local state (process registers), a shared state in the STM, and ports which are assumed to be represented in the STM. Those objects have the particularity to be made implicitly recoverable by hardware. Considering a realistic architecture, there are clearly other objects to be controlled by an operating system for which recoverability might not be implicitly hardware provided (ex. i/o devices). A process which accesses a local state and implicitly hardware recoverable objects will be referred to as a *standard* process while a process which accesses not implicitly hardware recoverable objects will be referred to as a *non-standard* process. Given the previous distinction, we may expect the kernel services to be programmed as a set of standard and non-standard communicating processes. User processes should be standard in the sense that recoverability should be transparent to them.

Programming a non-standard server process will depend on the type of unrecoverable objects the process is dealing with. However, it is interesting to propose some programming guidelines. These are discussed in the following.

General principles for non-standard servers

If each service of a server is programmed as a *restartable* operation O, servicing a request despite a processor failure can be obtained in the following way [Lampson 81]: (i) save the server's context in stable storage, (ii) perform O, and (iii) erase the server's context from stable storage. If a processor failure occurs while performing O, the process will resume after (i) and will perform O again, the resulting execution sequence is equivalent to a single execution of O by definition of a restartable action.

Given our model of computation, we may think to embed the server's operation O within two *NewProcessTransaction* primitives so as to ensure service despite failures. There are two aspects to take into account, however. First, recall from section 2 that such a region may be dynamically broken into multiple transactions due to the implicit commitments performed by the recovery protocol. Notice that consequently, the restartable property if any of a service does not lead immediately to a solution in contrast to above. Second, O will in general be a compound action made of both recoverable and non-recoverable actions. The first situation is particularly embarassing so that we may think of providing explicit *kernel transactions* (or seed transactions) which would be fully under the programmer's control. This approach is taken in [Banâtre 91a] for instance and facilitates the programmer's work who has to deal only with the recoverability provision of the non-recoverable objects used within a kernel transaction. Providing kernel transactions requires much work to be done and thus we wish to dispense ourselves from implementing such a mechanism. Rather, we envisage to use only the properties of the services to be programmed together with the tools discussed below to program (the limited number of) non-standard servers.

The provision of some limited amount of non-recoverable memory by the architecture might be useful for programming non-standard servers so as to record the state of objects for which explicit recovery is needed. (By definition, the contents of a (private) non-recoverable memory cell used within a process transaction is not restored should the current process transaction be rolled back.)

In many cases, a non-standard process, which current transaction is rolled-back, will wish to perform exceptional work before carrying service. The seed provides an exceptional mechanism *RollBackAt*(*address*) for that purpose meaning that the flow of control of the process will be resumed at *address* should the current process transaction be rolled-back. Such a mechanism may be triggered in different ways by the calling process. First, *RollBackAt* might be provided as an explicit seed primitive. Second, the roll-back *address* might be provided as an explicit exceptional *continuation* [Livercy 78] argument to each seed primitive. Third, if the programming language offers a mechanism for handling exceptions, it is appropriate to map the roll-back of a process transaction onto an exception which may then be handled according to the rules defined by the language.

A non-standard server may explicitly commit its current transaction. The seed offers the primitive *NewProcessTransaction(p)* for that purpose where *p* denotes a process. Basically, a call to this primitive will execute a code sequence similar to the scheduler sequence depicted in figure 8.

Communication issues

A client process requests the service of a server by sending its request on the server port. Conversely, the server replies to the request by sending its reply on the client port. As far as communications are concerned, it is clear that communications between standard processes (inside the standard domain) themselves do not raise difficulties as explained previously. But, communications with non-standard processes may obey a particular protocol which we present in the following.

In order to facilitate the provision of fault tolerance measures within a non-standard server, the server may require that the client *commits* its request before it can be got by the server. In other words, the client's request is an *intention* [Lampson 81] that has to be performed by the server. The underlying reason is that in general it will be easier for a non-standard server to restart the processing of a request that must be performed than to be possibly obliged to cancel the processing of a request (an *orphan* execution) retracted by the client should a client's roll back occur. Notice that in the general case, a client's call will give rise to nested calls which to be cancelled would require cancelling recursively all orphan executions raised by the call. Communications with non-standard processes is likely to occur very often and therefore a cheap implementation of the previous protocol must be done. This dictates that, the protocol must be implemented at the bottom layer (the seed) so as to use the hardware facilities in the most efficient way.

Summary

In summary, the following guidelines for programming non-standard servers can be proposed.

- A non-standard server may make use of non-recoverable memory in order to record the recovery data of some objects.
- A non-standard server may provide a handler for dealing with a roll back of its current process transaction triggered by the recovery protocol due to a processor failure.
- A non-standard server may explicitly commit its current process transaction. (Recall however that a process has not a full control over its transactions since the recovery protocol may itself implicitly commit its current transaction.)

- A non-standard server may obey to a particular communication protocol so as to facilitate the provision of its fault tolerance measures.

The application of these techniques will typically depend on the type of unrecoverable objects to be dealt with and is left to the creativity of the system programmer. Only the experience will reveal if these principles are sufficient in practice.

4.4 Memory management

Memory management is a main and complex component in any operating system. Our aim, in this section, is not to describe in detail the intricacies of sophisticated virtual memory management schemes (this has been discussed elsewhere [Krakowiak 85]) but to attempt isolate the new problems that virtual memory management may raise as far as fault tolerance is concerned. For the purpose of illustration, we introduce below the main features of a memory management scheme. (A particular implementation may not correspond exactly to this example, but this framework is adequate for us to illustrate the issues discussed.)

We assume that that a *shared segmented virtual space* is provided to processes. A process references a word within a segment by a couple $< SegmentId, SegmentOffset >$ where $SegmentOffset$ denotes the offset from the beginning of the segment. A segment is a linear address space. For the purpose of physical memory management, each segment is being paged. A $SegmentOffset$ gets decomposed into a couple $< SegmentPageNo, PageOffset >$. The main memory is divided into memory blocks, each block being capable to hold a page. We assume that a secondary storage memory is available for extending the capacity of the main memory. A segment page may then be resident in main memory or not. In the former case, we assume that the page has a *single* copy in main memory. In the latter case, the page is resident on secondary storage and can be brought into main memory if necessary. We assume a perfect model for secondary storage that is writing a block is assumed to end up correctly and reading a block is assumed to return the correct value of the last writing to the block.

The model of computation provided above the memory management (residing on top of the seed) is identical to the one provided by the seed itself except that now dependencies between processes must be (logically) tracked on the *virtual* adresses referenced by the communicating processes. Ideally, we would like the STM hardware still providing the necessary abstraction although the STM is only aware of the physical accesses. It is clear that as long as a segment page is not relocated, the STM provides the necessary abstraction since a segment page can have only a single copy in physical memory. If the relocation activity were only involving standard processes, we may also convince ourselves that the STM will achieve the necessary abstraction. A dependency will be recorded between a process accessing a page p before relocation and a process accessing p after relocation since the relocation activity itself must access both physical locations.

Now, as it will be the case in a realistic environment, paging may involve non-standard processes. In that case, (without further constraints and assumptions), the STM cannot by itself provide the necessary abstraction. Consider, for instance, the case where swapping out and swapping in pages are devoted to separate non-standard server processes. Logically, swapping in a given page is dependent of the last swapping out of the page, but without further assumptions, the STM will not record this dependency since these activities do not operate within the standard domain.

There are several possibilities to overcome this difficulty. First, swapping out and swapping in might be programmed so as to access explicitly common STM objects for which the hardware will implicitly track the dependencies. Second, some commitments might be forced when relocation is performed so as to ensure that within a group of dependent process transactions, a given virtual access cannot be mapped to distinct physical locations.

Now, let us turn our attention to process transaction commitment. A simplifying principle might be to consider that secondary storage contains only committed data since then rolling back a transaction will not affect the secondary storage. This principle does not appear very restrictive for if a page is not committed, it is likely to be part of the working set of a process anyway, and therefore should better reside in main memory. For performance reasons, we do not wish commitment involve any secondary storage operation either.

Typically, the virtual memory management will reside above the seed layer except for some small hardware dependent parts such as handling the page fault trap interrupt for instance. Now, virtual memory management may not be always required in real-time applications for instance. In that case, we may envisage the simplified memory management services to be provided by the seed itself.

The purpose of this section was to discuss some main problems that memory management may raise as far as fault tolerance is concerned. We are aware not having solved entirely these problems which would require further detailed studies. For instance, an idea which we intend to investigate is whether part of the work of dependency tracking could not be shifted at the level of the page tables instead of the memory cells themselves when virtual memory is implemented.

5 Related works

Several proposals for making a computing system tolerant to faults have been published [Lee 90]. In this section, we review some of them with particular emphasis on tolerance to processor failures.

The Stratus system [Wilson 85, Harrison 87] has adopted the approach of using hardware redundancy as the main strategy for achieving fault tolerance. A Cpu is located on an individual circuit board and contains two copies which operate synchronously together. This mechanism is

similar to our proposal which provides fail-stop processors. However, in Stratus, every board also runs in a duplex fashion so that in the event of a board failure, the redundant board can continue processing without interruption. Consequently, each program is actually executed simultaneously by *four* Cpu's. This replication technique is also applied to other critical components (memory, I/O controllers ...).

Another method for achieving fault tolerance in message based loosely coupled architectures is based on the *process – pair* scheme. Every process running in the system is backed-up on another processor. A primary process executes the main computation and is periodically synchronised with an inactive back-up process that holds checkpoints which are process states from which computation can be safely restarted if the processor running the primary fails. Tandem [Bartlett 87] and the recent Targon/32 system [Borg 89], for instance, use this strategy. This scheme has also been integrated into the MACH micro-kernel [Babaoglu 90]. It should be noted that with this strategy, much of the fault tolerance is implemented by software as opposed to the previous strategy which is based on hardware redundancy.

The initial proposal of stable storage is described in the seminal paper [Lampson 81]. In this reference, a disk stable memory is proposed. A stable block is represented by two physical images residing on separate disks. Writing a block is a unitary u_atomic operation despite failures and causes the successive writing of both block images. Reading a block is also a unitary u_atomic operation. (Further details can be found in [Lampson 81].) Our proposal is different in many respects from that proposal. First, the access speed to the stable memory is improved due to the fact that the STM contents are directly addressable by the processors. Second, the STM offers the u_atomicity property of transactions which may embed several object accesses. (Both of these properties are also provided by the stable storage described in [Banâtre 88].) Third, the STM provides recoverability to passive objects but also to process communications in a straightforward manner as explained previously.

Yet, another method, which we examine in greater detail since it is closer to our approach, is illustrated by the Sequoia multiprocessor machine [Bernstein 88]. In the Sequoia machine, each processor is associated with a *blocking* cache allowing multiple writes on a cache line before main memory is updated. No replacement of modified lines is made within the cache until it is flushed, hence the blocking cache term. Each processor performs memory updates locally within its cache and periodically checkpoints its state by flushing the cache and its internal registers to main memory. (A cache flush may be initiated by the cache controller if a dirty line needs to be replaced.) Modified data is flushed into two distinct memory modules under processor controls in order to handle memory and processor failures.

To avoid potential rollback propagation, the Sequoia architecture prohibits direct data sharing between processors. Shared data structures must be accessed within explicit processor critical sections protected by test-and-set locks. This is a notable difference with our proposal

which implicitly tracks dependency when sharing occurs. Since direct data sharing is prohibited in Sequoia, a processor willing to modify shared data structures first invalidates the contents of its cache after acquiring the lock in order to fetch upto date data, performs the update locally, checkpoints modifications by flushing its cache so that the updated data becomes accessible to other processors and last releases the lock.

Another scheme for fault tolerant shared memory multiprocessors is presented in [Wu 90]. In this approach, checkpoints are stored within the cache associated with each processor. Shared memory and the caches are assumed to be reliable. Only transient processor failures are tolerated. With this assumption, it is not necessary to flush the caches to shared memory when processors are checkpointed. A processor p is checkpointed whenever a cache line modified since the last checkpoint has to be written back to shared memory, that is on cache miss. To keep checkpoints coherent and avoid potential rollback propagation, processor p is also checkpointed whenever another processor reads a line modified within the cache of processor p.

An extension to this scheme was made in [Ahmed 90]. In this approach, a limited form of dependency tracking is proposed: each cache maintains a single flag bit indicating if an interaction has occured with some other processors during its current checkpointing interval. Whenever a processor checkpoints or rolls back, this event is snooped by other caches which take the appropriate action.

This section was an attempt to review some of the main works related to the field of hardware fault tolerance. Clearly, this survey is not complete; this area being a very active research field.

6 The FASST project

Some of the ideas discussed in this paper are currently being explored in an ESPRIT project called FASST (Fault-tolerant Architecture with Stable Storage Technology) [FASST 92]. FASST involves several European partners both from the industry and universities. The project started in 1991 and aims at (i) carrying research works in the field of fault tolerance, and (ii) building an industrial fault tolerant prototype system. The structure of the prototype system is depicted in figure 9. We now examine each of the prototype system layers in turn.

The hardware level is a shared memory multiprocessor machine involving multiple components communicating on the bus. Processors are fail-stop. The shared memory of the multiprocessor would basically offer the same functionality as the Stable Transactional Memory described in section 3. The hardware configuration will be completed by i/o devices.

The μ-kernel layer will control this hardware. The μ-kernel approach is in the case of FASST very appropriate as discussed in the following.

First, this will facilitate the provision of multiple run-time supports at the operating system layer such as a transactional support system (TSS) and a general-purpose kernel such as the

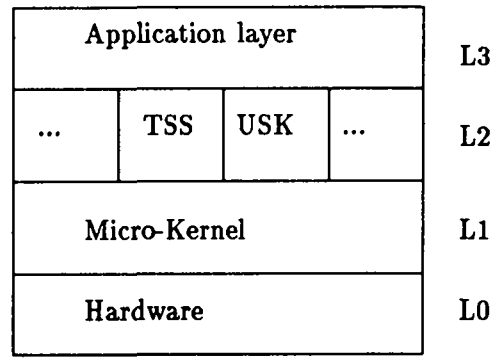

Figure 9: Structure of a FASST system

UNIX kernel (USK).

Second, ideally, the μ-kernel in collaboration with the hardware services would mask any single point of hardware failure that might occur at the hardware level thereby relieving the upper layers to cope with them.

At the application layer, several important application domains have been identified. It is planned to evaluate a specific application for each of them. Fault tolerance mechanisms should be transparent to the application layer so that applications can take full adavantage of the fault tolerant features provided by the architecture without any additional programming effort.

Acknowledgements

Some of the issues discussed in the report have benefitted from discussions with FASST partners whose comments are gratefully acknowledged. We also wish to thank A. Gefflaut for his valuable work on performance evaluation. Finally, we thank I. Puaut, whose comments enabled to improve earlier drafts of the document.

References

[Ahmed 90] R.E. Ahmed, R.C. Frazier, and P.N. Marinos. Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, pp 82–88, Newcastle, June 1990.

[Archibald 86] J. Archibald and J. L. Baer. Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[Babaoglu 90] O. Babaoglu. Fault-Tolerant Computing Based on Mach. *Operating System Review*, 24(1):27–39, January 1990.

[Banâtre 88] J. P. Banâtre, M. Banâtre, and G. Muller. Ensuring Data Security and Integrity with a Fast Stable Storage. In *Proc. of 4th International Conference on Data Engineering*, pp 285–293, Los Angeles, February 1988.

[Banâtre 90] M. Banâtre and P. Joubert. Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, pp 89–96, Newcastle, June 1990.

[Banâtre 91a] M. Banâtre, P. Heng, G. Muller, and B. Rochat. How to Design Reliable Servers using Fault Tolerant Micro-Kernel Mechanisms. In *USENIX Mach Symposium*, pp 223–231, Monterey, Californie, November 1991.

[Banâtre 91b] M. Banâtre, G. Muller, B. Rochat, and P. Sanchez. Design Decisions for the FTM : A General Purpose Fault Tolerant Machine. In *Proc. of 21th International Symposium on Fault-Tolerant Computing Systems*, pp 71–78, Montréal, Canada, June 1991.

[Baron 88] R. Baron, D. Black, W. Bolosky, J. Chew, D. Golub, R. Rashid, A. Tevanian, and M. Young. *MACH Kernel Interface Manual.* Department of Computer Science, Carnegie-Mellon University, September 1988.

[Bartlett 87] J. Bartlett, J. Gray, and B. Horst. Fault Tolerance in Tandem Computer Systems. In Avizienis A., Kopetz H., and Laprie J.C., editors, *The Evolution of Fault-Tolerant Computing*, volume 1, pp 55–76. Springer Verlag, 1987.

[Bernstein 87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[Bernstein 88] Ph. A. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, pp 37–45, February 1988.

[Best 82] E. Best. Relational Semantics of Concurrent Programs (with some Applications). Technical Report 180, University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7Ru, England, July 1982.

[Borg 89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[Censier 78] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[FASST 92] FASST Project Consortium. *The FASST Architecture: Overall Requirements and Specifications.* Stollmann GmbH, Hamburg, Germany, January 1992.

[Gray 78] J. Gray. *Notes on Database Operating Systems.*, volume 60 of *Lecture Notes in Computer Science.* Springer Verlag, 1978.

[Gries 81] D. Gries. *The Science of Programming.* Springer Verlag, New York, 1981.

[Harrison 87] E. S. Harrison and E. Schmitt. The Structure of SYSTEM/88, a Fault-Tolerant Computer. *IBM Systems Journal*, 26(3):293–318, 1987.

[Katz 85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proc. of 12th Annual International Symposium on Computer Architecture*, pp 276–283, Boston, 1985. IEEE.

[Kitagawa 91] M. Kitagawa. Understanding MBus. In B. Catanzaro, editor, *The Sparc Technical Papers*, pp 425–442. Springer Verlag, 1991.

[Krakowiak 85] S. Krakowiak. *Principes des systèmes d'exploitation des ordinateurs.* Dunod informatique, 1985.

[Lampson 81] B. Lampson. Atomic Transactions. In *Distributed Systems and Architecture and Implementation : an Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pp 246–265. Springer Verlag, 1981.

[Lee 90] P.A. Lee and T. Anderson. *Fault Tolerance : Principles and Practice*, volume 3. Springer Verlag, New York, second revised edition, 1990.

[Livercy 78] C. Livercy. *Théorie des Programmes.* Dunod, 1978.

[Randell 75] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[Randell 78] B. Randell, P. Lee, and Treleaven P. C. Reliability Issues in Computing System Design. *ACM Computing Surveys*, 10(2):123–166, June 1978.

[Rozier 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Léonard, S. Langlois, and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems-Usenix*, 1(4):305–370, 1988.

[Schneider 87] F. B. Schneider. The FAIL-Stop Processor Approach. In *Concurency control and reliability in distributed systems, Chapitre 13*, pp 370–394. Barghava, 1987.

[Singh 91] J.P. Singh, W. Weber, and A. Gupta. SPLASH : Stanford Parallel Applications for Shared-Memory. Technical report, Computer Systems Laboratory Stanford University, 1991.

[Smith 82] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Strom 85] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[Sweazey 86] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proc. of 13th Annual International Symposium on Computer Architecture*, pp 414–423, Tokyo, June 1986. ACM/IEEE.

[Thacker 88] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite. Firefly : A multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Wilson 85] D. Wilson. The STRATUS Computer System. In T. Anderson, editor, *Resilient Computer Systems*, pp 208–231, 1985.

[Wood 85] W.G. Wood. Recovery Control of Communicating Processes in Distributed Systems. In S.K. Shrivastava, editor, *Reliable Computer Systems*, pp 448–484. Springer Verlag, 1985.

[Wu 90] K.L. Wu, W.K. Fuchs, and J.H. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, April 1990.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 635 A NOTE ON CHERNIKOVA'S ALGORITHM
Hervé LE VERGE
Février 1992, 28 pages.

PI 636 ENSEIGNER LA TYPOGRAPHIE NUMERIQUE
Jacques ANDRE, Roger D. HERSCH
Février 1992, 26 pages.

PI 637 TRADE-OFFS BETWEEN SHARED VIRTUAL MEMORY AND MESSAGE-PASSING ON AN iPSC/2 HYPERCUBE
Thierry PRIOL, Zakaria LAHJOMRI
Février 1992, 26 pages.

PI 638 RUPTURES ET CONTINUITES DANS UN CHANGEMENT DE SYSTEME TECHNIQUE
Alan MARSHALL
Mars 1992, 510 pages.

PI 639 EFFICIENT LINEAR SYSTOLIC ARRAY FOR THE KNAPSACK PROBLEM
Rumen ANDONOV, Patrice QUINTON
Mars 1992, 20 pages.

PI 640 TOWARDS THE RECONSTRUCTION OF POSET
Dieter KRATSCH, Jean-Xavier RAMPON
Mars 1992, 22 pages.

PI 641 MADMACS : A TOOL FOR THE LAYOUT OF REGULAR ARRAYS
Eric GAUTRIN, Laurent PERRAUDEAU
Mars 1992, 12 pages.

PI 642 ARCHE : UN LANGAGE PARALLELE A OBJETS FORTEMENT TYPES
Marc BENVENISTE, Valérie ISSARNY
Mars 1992, 132 pages.

PI 643 CARTESIAN AND STATISTICAL APPROACHES OF THE SATISFIABILITY PROBLEM
Israël-César LERMAN
Mars 1992, 58 pages.

PI 644 PRIME MEMORY SYSTEMS DO NOT REQUIRE EUCLIDEAN DIVISION BY A PRIME NUMBER
André SEZNEC, Yvon JEGOU, Jacques LENFANT
Mars 1992, 10 pages.

PI 645 SKEWED-ASSOCIATIVE CACHES
André SEZNEC, François BODIN
Mars 1992, 20 pages.

PI 646 INTERLEAVED PARALLEL SCHEMES : IMPROVING MEMORY THROUGHTPUT ON SUPERCOMPUTERS
André SEZNEC, Jacques LENFANT
Mars 1992, 14 pages.

PI 647 COMMUNICATING PROCESSES AND FAULT TOLERANCE : A SHARED MEMORY MULTIPROCESSOR EXPERIENCE
Michel BANATRE, Maurice JEGADO, Philippe JOUBERT, Christine MORIN
Mars 1992, 40 pages.

ISSN 0249-6399