An active stable storage and its integration in a multiprocessor architecture

Michel Banâtre, Gilles Muller, Jean-Pierre Banâtre

▶ **To cite this version:**

Michel Banâtre, Gilles Muller, Jean-Pierre Banâtre. An active stable storage and its integration in a multiprocessor architecture. [Research Report] RR-0693, Inria. 1987. inria-00075860

HAL Id: inria-00075860
https://inria.hal.science/inria-00075860

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

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

N° 693

AN ACTIVE STABLE STORAGE AND ITS INTEGRATION IN A MULTIPROCESSOR ARCHITECTURE

Michel BANATRE
Gilles MULLER
Jean-Pierre BANATRE

JUIN 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

PUBLICATION INTERNE N° 362 - MAI 1987
22 PAGES

An Active Stable Storage and its Integration in a Multiprocessor Architecture.

Une Mémoire Stable Active et son Intégration dans une Architecture Multiprocesseur.

Michel BANATRE, Gilles MULLER
IRISA-INRIA
Jean-Pierre BANATRE
IRISA-INSA and INRIA

Campus de Beaulieu, 35042, Rennes-cedex. (France).

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Abstract:

A stable storage is a reliable device used to store persistent information (files) or temporary data structures such as those needed to implement commit protocols. Solutions provided to build such a device are based on data redundancy. This paper describes an active stable storage able to manage structured objects and to protect itself against faulty behaviour of the processor. A performance analysis shows the possibility of designing a fault tolerant machine using a stable storage as a main component.

Key words:

Stable storage, protection, redundancy, fault tolerant machine, failure recovery.

Résumé:

La mémoire stable est un support d'information fiable qui est utilisé pour le stokage de structures de données permanentes (fichiers) ou comme celles qui sont nécessaires à la mise en œuvre des protocoles de terminaison des actions atomiques. Cet article décrit une mémoire stable active adaptée à la gestion d'objets structurés. Elle possède des mécanismes de protection internes pour survivre aux comportements anormaux du processeur qui l'accède.

Une analyse comparative des temps d'accès de cette mémoire démontre la possibilité de l'utiliser comme le composant de base d'une machine multiprocesseur tolérante aux fautes.

Mots clés:

Mémoire stable, protection, redondance, machine tolérante aux fautes, traitement d'erreur.

PAPIER RECUPERE ET RECYCLE

Contents:

1. Introduction.

A stable storage is a reliable device used to store persistent informations (files) or temporary data structures such as those needed to implement commit protocols. Solutions provided to build such a device are based on data redundancy. The two main properties of a stable storage can be summarized as follows:

1-Resistance against external hardware or software failures (processor failure, power failure, bad memory access) and also resistance against internal failures such as memory decays.

2-Atomicity of read and write accesses.

The first implementation of such a memory was made in order to build a distributed file system, DFS [LAMP-81]. It was used to store information related to file management. In this implementation, two physical blocks on disk are associated with one logical block. Physical blocks are updated sequentially. In order to provide resistance against disk failures and decays, the two physical blocks are located on two independent drives. This solution is acceptable for file servers only because the grain of recovery is rather coarse [SVOB-84].

Recent studies in distributed systems provide atomic operations on abstract objects [LISK-84], [SPEC-85]. In order to get an efficient implementation of such operations, it is necessary to take into account the size of the objects, which may be small and the grain of recovery, which is fine as the execution time of primitives is very short. Of course, for efficiency reasons, it is not possible to use a stable storage built from disks due to their access time. A proposal was made within the framework of the ENCHERE system [BANA-86]. To manage small objects, this system provides a stable storage built from non volatile RAM memory. The memory is composed of eight banks **which belong to the address space of the processor**. However a major problem appears: if the stable storage is part of the machine address space, it would be vulnerable to erroneous programs. Since the stable storage should enable a system to recover from failures, **it should be guaranteed that the stable storage is unlikely to be overwritten by accident**. In that view, special care should be taken in order to provide appropriate mechanisms which will make it highly improbable that the stable storage will be damaged. In ENCHERE, hardware and software mechanisms, based on encryption were used to protect this memory against uncontrolled access due to hardware or software failures.

The stable storage provided in ENCHERE was designed only to manage objects consisting of contiguous memory blocks. It is important to avoid this limitation when one considers objects which have a concrete representation spread over the memory. To provide such facilities a possible solution could rely on a mechanism such as a commit protocol in order to provide atomic reading and writing of objects, but this has been abandoned for efficiency reasons.

Another reason to reconsider the stable storage design is related to fault-tolerant architectures. Such architectures should survive despite the possible unavailability of processor

storage facilities. Recovery techniques have been studied to provide high reliability in a transaction processing environment using multi-processor machines [BART-81], [BORG-83]. The hardware redundancy provided on these machines may be coupled with software techniques, as in TANDEM™ machines which implement the process-pair mechanism. However, all these machines, based on computer redundancy and message-passing facilities, are very expensive. The present study analyses of using a fast stable storage device as basis for cheap fault tolerant machines based on data redundancy.

This paper describes a new architecture for a stable storage which allows fine grain of recovery and high fault-tolerance. This device overcomes the deficiencies observed in the implementation of the ENCHERE system, in particular the following objectives have been fixed:

(i)-the access time of the stable storage should be of the same order as the access time of a usual RAM memory,

(ii)-it should be possible to handle atomically complex data structures, which are normally managed by atomic activities. It is clear that the provision of hardware mechanisms for atomic update of these data structures should improve the performance of the stable storage,

(iii)-the stable storage should not be sensitive to processor malfunctioning. This should imply:

- auto-protection of the stable storage against processor failure,
- autonomy of the stable storage. It should be able to take some decisions, for example to decide that the processor which normally accesses it is faulty and then initiate a reconfiguration.

These properties of **auto-protection** and **autonomy** are achieved through very strict access control performed by the stable storage which, unlike usual memories, is an **active** device.

Section 2 describes the structure and properties of a new stable storage board (SSB). Section 3 presents failure detection mechanisms associated with the SSB. The architecture and performance of the SSB are detailed in section 4. Section 5 describes the integration of the stable storage in a multi-processor architecture. Finally, section 6 provides a brief review and discussion.

2. Operational principles.

This section describes the operational principles of the SSB. After a brief presentation of the kind of objects handled by the SSB and an overview of its hardware structure, the stable read and update (or write) operations are presented.

2.1. Information structures handled by the SSB.

The SSB can manage objects of type record, array or list. Constituents of such objects may be physically contiguous or not on the SSB device. The elementary memory access is the word

access, so accessing an object results in accessing its word components.

The following property characterizes the semantics of read and update access to objects:

(P-access)-*Any access to an object implies accessing all its word components, once and only once.*

The property (P-access) is ensured if the following condition is assumed:

(C-access)- *All the word components of an object are accessed by logically increasing addresses.*

2.2. A first glance to the hardware structure of the SSB.

In order to implement stability, there should be at any time, at least one valid copy of any object. Actually, in stable storage, every object possesses two copies located on two physically independent memory banks. Accesses to these banks are mutually exclusive. This redundancy allows the detection of memory decays and their eventual correction assuming that a valid copy of the object exists on one of the two banks and that this valid copy may be identified.

In order to guarantee high performance, the SSB is built from non-volatile RAM memory banks belonging to the processor address space. Access control is ensured by internal hardware mechanisms (the object manager). The object manager does not rely on the processor and is sufficiently simple to be implemented in hardware. Its main components are described in section 3.

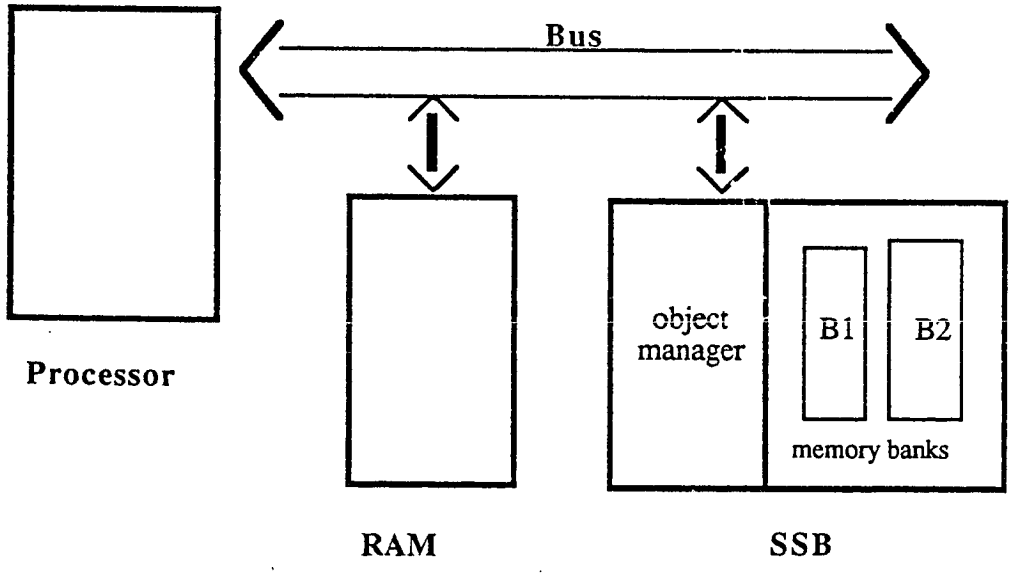

Figure 1: A processor and its associate SSB.

2.3. Atomic update operation.

In order to update an object O from volatile memory to SSB, a copy has to be made

atomically on both banks (figure 2). To do so, two processing steps are necessary:

(i) O is copied onto bank 1 (B1) giving Ob1.

(ii) Ob1 is copied, onto bank 2 (B2) giving Ob2. This operation is internal to the SSB.

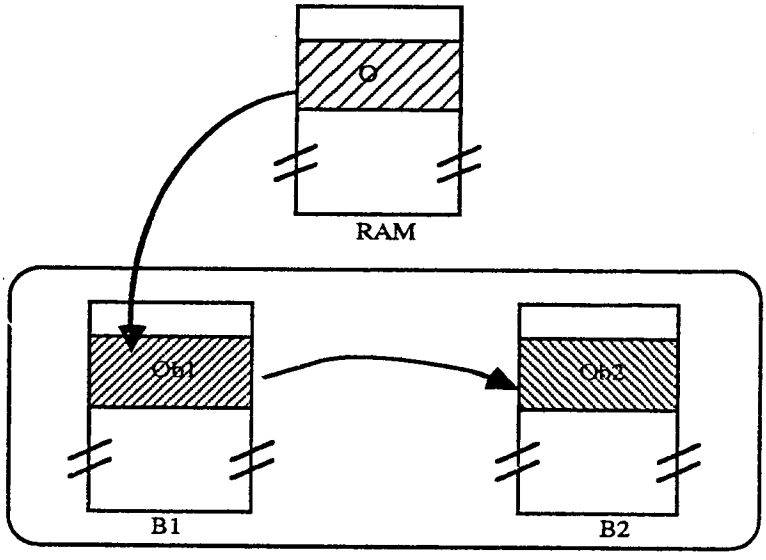

Figure 2: Updating an object in stable storage.

Should a crash occur in while copying O onto Ob1, Ob2 is used in order to restore Ob1, the update operation is considered as "not done".

Should a crash occur between step (i) and step (ii) or while copying Ob1 onto Ob2, Ob1 is used to restore Ob2, the update operation is considered as "done".

2.4. Atomic read operation.

The stable read operation consists of only one processing step which performs the actual reading of the object. This is done by reading and comparing, internally to the SSB, versions of the object situated on bank 1 and bank 2. The only failures which can be detected by this comparison are those subsequent to memory decays. Actually, the stable reading of an object involves the stable reading of all words of this object (property P1), the stable reading of a word w implies reading the two versions w_1 on bank 1 and w_2 on bank 2, and comparing their value. Several situations may occur:

- $w_1=w_2$ and no decay is detected (by using a parity bit when reading a memory

word) , then the read value is returned to the processor.

•$w_1 \neq w_2$ and no decay has been detected, then a fatal error is signaled.

•A decay is detected on only one word, then the other word is validated and returned.

•A decay is detected on both banks, a fatal error is signaled.

This simple technique allows the recovery from decays if a valid copy exists on one of both banks. Of course, it is useless if both words w_1 and w_2 are decayed.

2.5. Create and delete operations.

Creating and deleting an object can be viewed as a particular case of updating an object in the SSB. However, as it will be shown hereafter, the update operation uses such information as the topology of the object for protection purposes. Additional hardware mechanisms have been provided to implement atomically the create and the delete operations since no topological information is available in this case. These mechanisms being quite similar to those used to implement atomic update are not detailed in the paper.

3. Failure detection mechanisms.

In this section,we think of a stable storage board as a stable store with an integrated object manager to control access to objects. The purpose of access controls provided by the object manager is twofold:

•first, checking that it is the desired object which is actually accessed,

•second, ensuring that read and update operations respect their semantics (P-access property).

Access control implementation is based on two mechanisms: (i) an identification key which is used to control that the right object is actually accessed and (ii) topological information about the structure of the accessed object which is used to check that the semantics of read and update operations are followed. These mechanisms are detailed in section 3.1 and 3.2. It is also necessary to control that the sequence of operations used for copying an object from bank 1 to bank 2 is correct. This is performed by an automaton as described in section 3.3.

3.1. Using a key to control access to an object.

Access control to an object O consisting of n words ($w_1,\ldots,w_n$) is made with a key. This key, denoted k(w1), is the content of the word w_1 and must already be stored in the RAM memory. The update operation is as follows (it involves the processor and the SSB):

Part of the operation performed by the processor.

(1) write ($k(w_1)$, reg-key)

where k(w1) is kept in the processor memory and reg-key is a register belonging to the SSB.

(2) update(w_1,v_1), v_1 is the new value which is going to be stored into w_1.

Part of the operation performed by the SSB device, after step (2).

```
if reg-key=k(w1)
        then write(w1,v1)
        else error
fi
```

If the content of reg-key is equal to the actual content of $w_1$currently stored in the SSB, then access to the object is granted. One can remark that the value of $k(w_1)$ changes when the value of w_1 changes. Here, when the update operation has proceeded, the value of $k(w_1)$ is v_1. The create operation initializes correctly k(w1) and stores it in RAM memory.

3.2. Using object representation to check the semantics of access operations.

Even after access control, it is possible that the sequence of basic read or write accesses is incorrect. Imagine that O consists of n memory components ($w_1,\ldots,w_n$), an access is erroneous if:

(i)-after accessing the w_i word of O, the next word to be accessed is w_j with $j\neq i+1$,

(ii)-the first access to O is made through an access to the w_i component with $i\neq 1$.

So even if access to the first word is valid, it is still necessary to check that further accesses are made in a right order to all the words of the object (C-access condition).

Detection of failures of type (i) and (ii) is made by linking together all components w_i of the object O. This link information is kept in a "link vector" (LV) such that LV[i] contains a coding of w_{i-1} address, denoted cd(w_{i-1}).

When accessing w_i, the following control procedure is executed on the SSB:

```
procedure control;
    begin
        if LV[i]=cd(w_{i-1}) address
            then
            #access to w_i authorized, save cd(w_i)#
            else
            #error#
        fi
    end
```

This procedure checks that the components of an object are accessed in increasing order. It cannot determine that all components of the object have been accessed. In order to do so, it is necessary to keep supplementary information (the bounds of the object) in a vector (BV) which allows a check that the last access to O concerns the last component of O. The vectors LV and BV are hardware implemented in the object manager. They are updated when creating or deleting an object.

3.3. Control of correct sequencing of operations.

The implementation of the stable update operation should assume that the sequencing of elementary processing steps be respected. This property is checked by an automaton A belonging to the SSB.

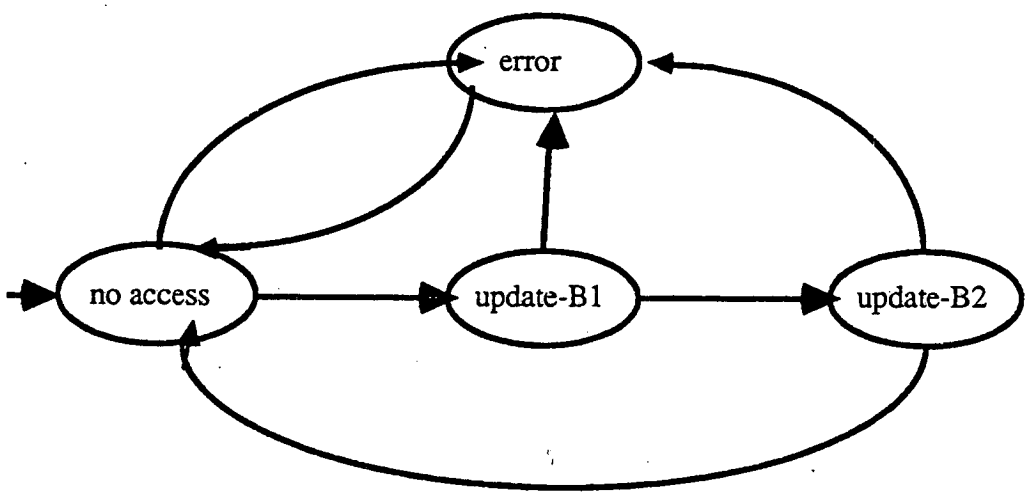

Figure 3: Automaton describing the update operation.

The current state of A is stored in a special SSB register "reg-state". In order to check the correct sequencing between processing steps, the processor loads into another SSB register "reg-com" the identity of the processing step to be executed, and uses the automaton in order to check the validity of the transition (current state to new state).

3.4. A complete description of the update operation.

The instructions executed by the processor for an object update, the actions and control operations executed by the object manager are summarized on the following figure.

w1,...,wn are the addresses of words of an object in the SSB.
m1,...,mn are the addresses of words of the copy O' of O in RAM memory.
k(w1) is the access key, stored in RAM memory.

Code executed by the processor	Control operations executed by SSB
#copy of O' onto Ob1#	
command:="update-B1"	Check that the transition (no access → update-B1) is valid
	reg-state:= command
#access control#	*#access control#*
write(k(w1),reg-key)	check if k(w1) is equal to reg-key
write(m1,w1)	
for all *#words w2,...wn#* do write(mi,wi) od	Controls are executed according to the control procedure. write operation is done on bank1
#copy of Ob1 into Ob2#	
command:="update-B2"	Check that the transition (update-B1→ update-B2) is valid
	reg-state:=command
#provision of the sequence address wi of the object O in the SSB#	Controls are executed according to the control procedure. Words of bank 1 are copied onto bank 2 according to the addresses provided
command:= "no-access"	Check that the transition (update-B2 → no-access) is valid
	reg-state:=command

Figure 4: Execution of an update operation.

Notice that copying an object from bank 1 to bank 2 is made **internally** to the SSB, the processor provides only the successive word addresses of the concerned object.

Another solution would have implied a read operation from bank 1 to RAM then a write operation from RAM to bank 2, as done in [LAMP-81]. However, as RAM may be accessed by the processor it could be possible that between read and write operations the value of the object be destroyed. Our solution avoids this kind of failure.

3.5. Discussion.

The three mechanisms which have just been proposed are complementary. The first

ensures that wrong access to the first word of an object is detected (with a high probability). However, it may be too weak, for example if the object is composed of words all initialized with the same value, then the key is not very significant, and access to any word would be granted. The second mechanism could detect this error as the link table should be accessed sequentially starting with its first element. Finally as the first two mechanisms deal with one bank only, it was necessary to add a new mechanism which could detect errors occuring during the copy from bank 1 to bank 2, this is the role played by the control automaton.

Classical protection mechanisms, rely on the use of keys and encryption[GIFF-81], capabilities or objects [LEVY-84], allow a strict control of the use of the object operations, but do not check that the operation performs correctly. So they solve protection problems, but do not take into account a possible faulty behaviour of the processor which could result in an incorrect execution of the operation. The mechanisms presented in this section allow the detection of this kind of failure which may be disastrous when we consider a stable storage implementation.

4. The architecture of the stable storage.

4.1. The hardware.

The stable storage board (SSB) is made of three parts:

(i)-a object manager which implements access mechanisms,

(ii)-two memory banks

(iii)-a multiplexer for communication between memory banks and the processor.

The board has battery backup power capable of saving the information that it contains for several hours in the event of a power failure.

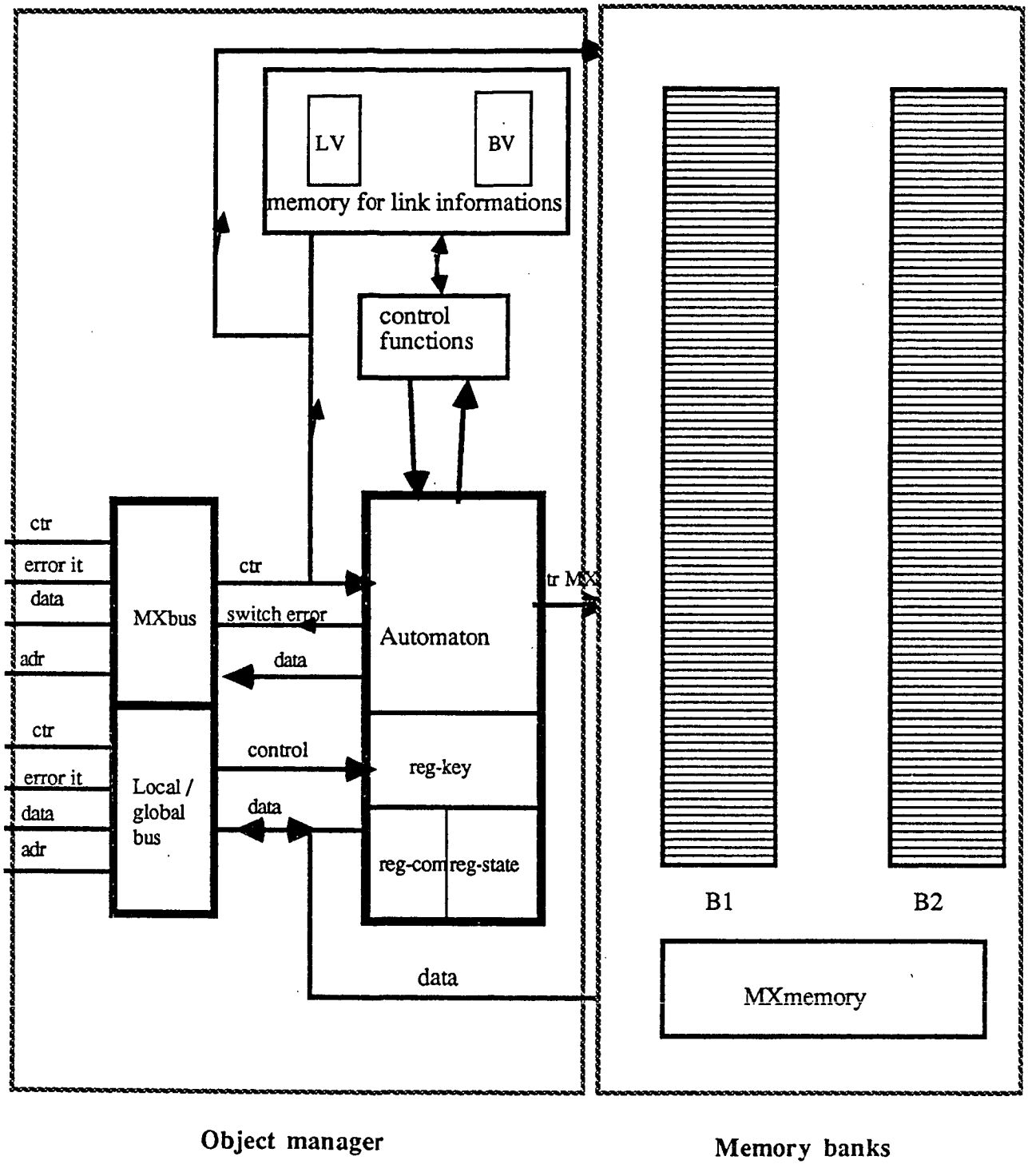

Figure 5: Stable storage board architecture.

For each memory access, the processor provides the object manager with an address and appropriate control signals concerning this access. If the object manager validates this access, then it sends appropriate commands (read, write,...) to the multiplexer (MXmemory) which selects

the bank where the data is to be read or written.

The SSB can be accessed via two buses in an exclusive way. Switching from one bus to the other is controlled by the automaton (switch error interrupt).

4.2. Performance of the stable storage.

In this section, we compare the performances of the SSB with those of two existing stable stores: the stable storage implemented in DFS whose performance is given in [MITC-82], and the ENCHERE stable storage board [BANA-86].

First, we give some information about implementation of these stable stores which will help in understanding the performance figures.

.DFS.

DFS stable storage is implemented using disks with sectors of 1024 bytes.

.ENCHERE.

ENCHERE stable storage is a 64 Kbyte RAM, divided into 8 banks. The size of a memory word is 8 bits. The stable storage is accessed via the bus MULTIBUS-1 and the processor used is an INTEL Isbc-8630.

.the SSB device.

The stable storage is composed of 2 banks of 1024 Kbytes each. The size of a memory word is 16 bits. The machine used is a multi-processor SPS7 (cf §5), equiped with Motorola-68010 microprocessors.

.RAM memory.

This memory is an ordinary one (no stability is provided), with 16 memory words. The machine used is also a SPS7. Performances of RAM memory are displayed only to evaluate the loss of performance observed when using stable stores instead of usual memories.

Update operation	DFS	ENCHERE	SSB	RAM
Byte	140 ms	19.5µs	12.6 µs	3 µs
Word (16 bits)	140 ms	39 µs	12.6 µs	3 µs
Word (32 bits)	140 ms	78 µs	17.4 µs	5.4 µs
1024 bytes	140 ms	20 ms	1.9 ms	0.9 ms

Figure 6: Comparison of performances of different stable storage.

The difference between DFS and the SSB is due to nature of physical device used in the two solutions. Any object, in DFS, demands at least one sector, so storing two bytes or 1024 bytes takes the same time. This stable storage is well adapted to the management of objects of great size. In order to update a stable sector, four disk accesses are necessary (write, check, write, check), every access takes 35 ms, so all in all 140ms are necessary.

The difference in performance between ENCHERE and the SSB device is due to new hardware solutions, the management of atomic access to objects and also to them progress of technology. For example the bus access time of ENCHERE stable storage is much longer than with the SSB.

One can notice that the acces time for the SSB device is between two and three times more than the one needed for an ordinary RAM memory. Due to this, we use the SSB device in order to build fault tolerant machines based on multiprocessor architectures as explained in the following section.

5. Integration of the SSB in a multi-processor architecture.

We have developed a prototype SSB for a multiprocessor machine called SPS7 [BULL-85]. The structure of our machine is represented on the figure 7.

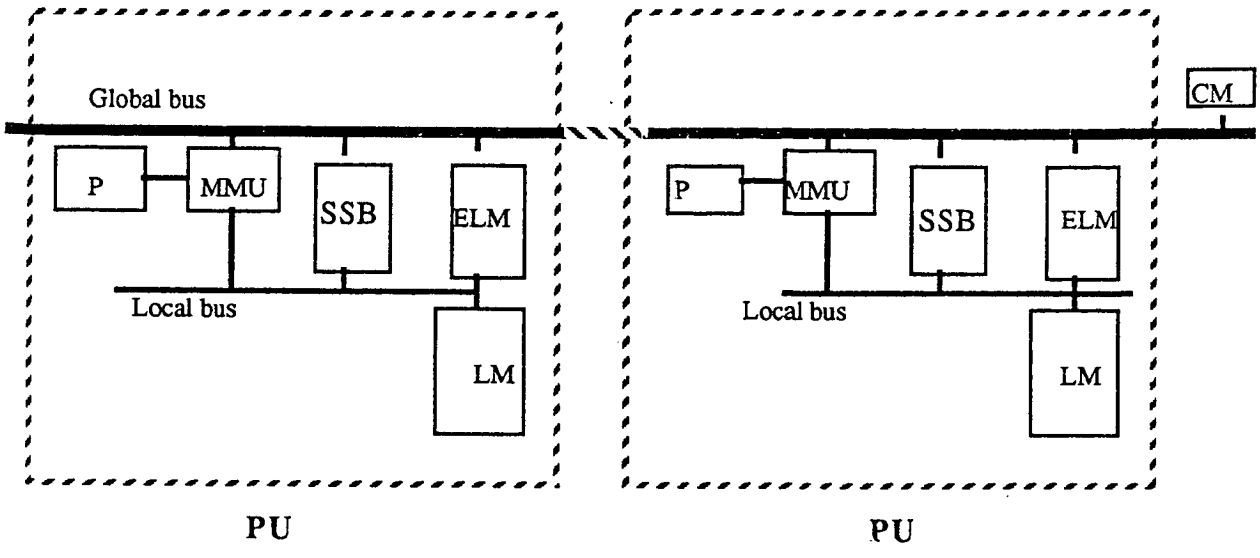

Figure 7: SPS7 architecture.

The SPS-7 architecture supports up to 8 processing units (PU), 16 exchange units (EU) and 16 megabytes of memory. The machine is organized around a global bus to which the different boards are attached. Moreover, every PU has its own local bus.

Every memory module can be plugged either into the global bus or into a local bus. In the first case this common memory (CM) is shared by the processors of different PU's. In the second case, the local memory (LM) can be accessed only by the processor of the PU. It is also possible to plug a memory board into both the global bus and a local one; in this case this external local memory (ELM) can be accessed through the two busses.

Every PU is equipped with a microprocessor (Motorola-68010), with a small amount of private memory and with a memory management unit (MMU). The MMU is a set of 1024 registers each containing a segment descriptor (physical address, length, access right, etc.). Segments described by such a descriptor can be located in any memory module connected either to the local bus of the processing unit or to the global bus.

The EUs are intelligent I/O modules, each one is made of a microprocessor, some memory and device controllers.

Every PU is enriched by a stable storage board normally accessed by the local bus,

(essentially for performance reasons). The integration of this component is rather straightfoward.

An interesting feature of this new architecture is its ability to provide a normal service (possibly degraded), even if some processing units have failed. Actually, it is sufficient that one processing unit and the global bus service survive in order to get a normal service as explained hereafter.

The detection of a crash of a processor P_i is performed by SSB_i **itself.** Every time period T, SSB_i sends a signal to P_i. If it happens that this signal is not acknowlodged by P_i, then SSB_i assumes that P_i is crashed. A failure of the processor P_i of PU_i makes inaccessible its stable storage SSB_i. As SSB devices contain critical information such as process states, checkpoints..., it is necessary to make this information accessible from another processor. This is done by granting access to SSB_i to another processor P_j via the global bus. So SSB_i is accessed by P_i through its local bus in normal use and by P_j ($j \neq i$) through the global bus in case of P_i's failure. Of course, these two access links are mutually exclusive.

Such a fault-tolerant architecture possesses several advantages:

- easy to build from existing multiprocessor machines,
- cheap, only SSBs have to be added to the processors,
- efficient as far as failure detection is concerned.
- efficient in the sense that it maximizes the productive use of hardware during normal execution (potential parallelism). The performance of the architecture is decreased only after a processor crash. In normal use all processors may execute different useful tasks.

6.Discussion

This paper has described a new high performance stable storage possessing the following qualities:

- it belongs to the machine address space,
- its access time is of the same order as the access time observed for usual RAM memories,

•it implements very strict access control, which should prevent it from any damage even in case of processor failure,

•it can be easily integrated to a multiprocessor architecture.

Presently, four prototype boards have been developed and integrated into a multiprocessor machine.

Apart from using the SSB to implement atomicity, our present research concentrates on the building of fault tolerant machines by using the active stable storage concept. The basic component of this new type of fault tolerant machines is a multiprocessor architecture, where every processor is equipped with a SSB, (as described in the previous section). Every SSB may be accessed via the local bus of the processor to which it is attached, via the global bus of the multiprocessor to which this processor belongs or via the global bus of the "neighbour" multiprocessor (figure 8).

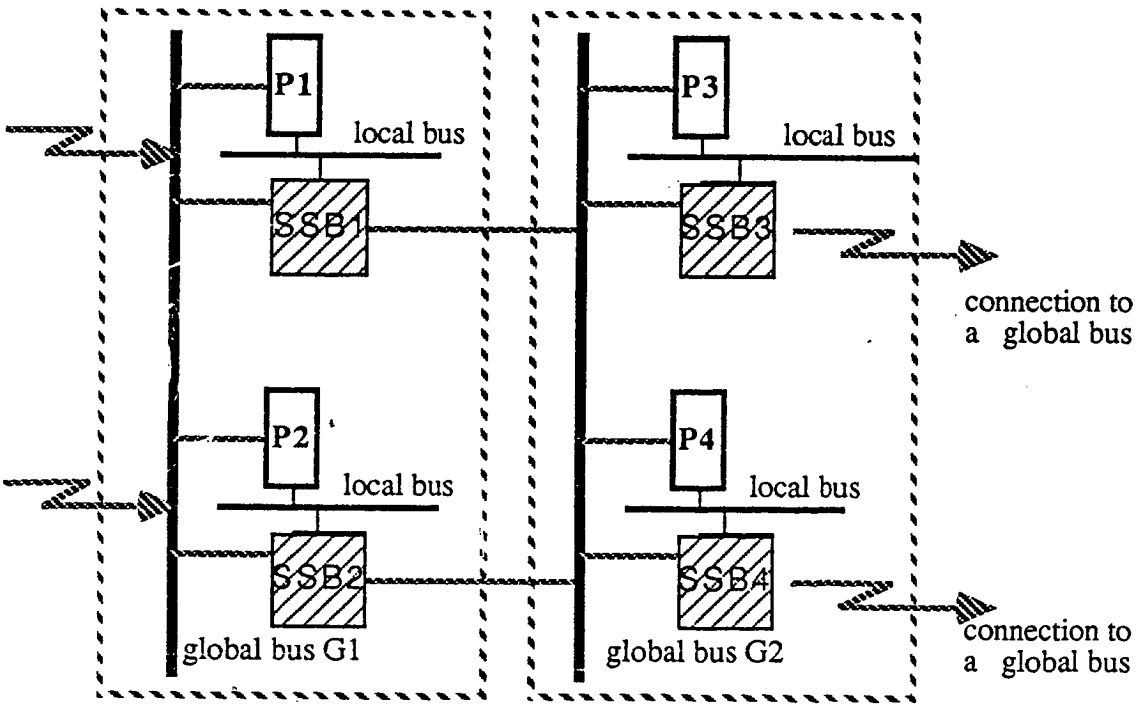

Figure 8: Architecture of a fault-tolerant machine.

Consider a process p_i running on processor P_i equipped with SSB_i. From time to time

pi may store information related to checkpoints in its SSB_i. These informations are either implicit, i.e., obtained through the implementation of appropriate software structuring mechanisms such as atomic actions, or explicitly provided by the programmer. In particular, we study mechanisms allowing a kind of micro-recovery where checkpoints are taken for every procedure call.

Should a failure occur during a procedure execution, an attempt to recover from it could be taken locally to the process executing the call, and only if this process cannot recover from this failure, the global mechanisms related to atomicity are invoked. The implementation of this kind of micro-recovery is realistic only because of the performance of the SSB.

As far as reconfiguration is concerned, the basic idea is the same as the one explained in §5. Actually SSB_i is able to detect a failure of P_i and then it becomes accessible from another processor P_j through the global bus of the multiprocessor to which P_i belongs, or through the global bus of the "neighbour" multiprocessor.

Though much work remains, our first experiences to date have convinced us that efficient fault tolerant systems based on stable storage are feasible and useful for a wide variety of transaction processing applications.

References.

[BANA-86] BANATRE J.P., BANATRE M., LAPALME G., PLOYETTE Fl.
The Design and Building of ENCHERE, a Distributed Electronic Marketing System.
Comm.of the ACM, vol 29, N° 1, jan. 1986, pp 19-29.

[BART-81] BARTLETT J.
A NonStop Kernel
Proc. of 8th on Operating System Principle, Pacific Grove, Calif. 1981

[BORG-83] BORG A., BAUMBACH J., GLAZER S.
A Message System Supporting Fault Tolerance.
Proc. of 9th on Operating System Principle, Bretton-Woods, N.H. (Oct. 1983),pp.90-99.

[BULL-85] BULL (C^ie)
General structure of SPS 7.
Technical Report, BULL, January 1985.

GIFF-81] GIFFORD D.
Information Storage in a Decentralized Computer System.
CSL-81-8, Xerox PARC, June 1981.

[LAMP-81] LAMPSON., ed.
Distributed Systems Architecture and Implementation: an Advanced Course.
LNCS 105, 1981.

[LEVY-84] LEVY H.
Capabilities-Based Computer System.
Digital Press, 1984.

[LISK-84] LISKOV B.
The Argus Language and System.
LNCS 190, 1984, pp. 343-430.

[MITC-82] MITCHELL J.G., DION J.
A Comparison of two Network-Based File Servers.
CACM 25,4, pp233-245. April 1982.

[SPEC-85] SPECTOR A.Z., DANIELS D., DUCHAMP D., EPPINGER J.L., PAUSCH R.
Distributed Transactions for Reliable Systems.
Proc. of 10th on Operating System Principle, Orcas Island, Washington, 1985.

[SVOB-84] SVOBODOVA L.
File Servers for Network-Based Distributed Systems.
Computing Surveys, Vol. 16, N° 4, December 1984. pp.353,398.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique