

Design decisions for the FTM : a general purpose fault tolerant machine

Michel Banâtre, Gilles Muller, Bruno Rochat, Patrick Sanchez

▶ **To cite this version:**

Michel Banâtre, Gilles Muller, Bruno Rochat, Patrick Sanchez. Design decisions for the FTM : a general purpose fault tolerant machine. [Research Report] RR-1400, INRIA. 1991. inria-00075160

HAL Id: inria-00075160
https://inria.hal.science/inria-00075160

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.

IRISA

Rapports de Recherche

N° 1400

Programme 1
Architectures parallèles, Bases de Données, Réseaux et Systèmes

DESIGN DECISIONS FOR THE FTM : A GENERAL PURPOSE FAULT TOLERANT MACHINE

Michel BANÂTRE
Gilles MULLER
Bruno ROCHAT
Patrick SANCHEZ

Mars 1991



IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX
FRANCE
Téléphone : 99.36.20.00
Télex : UNIRISA 950 473F
Télécopie : 99.38.38.32

Design Decisions for the FTM: A General Purpose Fault Tolerant Machine[1]

Les Choix de Conception du FTM : une Machine Tolérante aux Pannes d'Utilisation Générale.

Michel Banâtre, Gilles Muller,
IRISA-INRIA
Bruno Rochat,
IRISA-BULL
Patrick Sanchez
IRISA, Campus de beaulieu, 35042 Rennes Cedex (France)

Publication Interne n° 570 - Janvier 1991 - 30 pages
Nouveau programme 1

Abstract

Until now, fault tolerance has been reserved to specialized areas. However, due to the generalization of micro-computers and workstations in distributed environment, more users are concerned with reliability. For instance, diskless workstations are disabled by a failure of a file server. Consequently there is a need for a general purpose fault tolerant system which can support a wide range of applications. This is our goal in the design of the Fault Tolerant Multiprocessor machine.

The FTM hardware architecture is built from standard open machines connected by an interconnection sub-system. In such architecture processing elements are built using dynamic redundancy. When a processor fails the interconnection sub-system enables another processor to recover and restart computations from a non-erroneous state.

Our paper is devoted to the FTM architecture, in particular we detail an implementation of the interconnection sub-system with a stable transactional memory.

Key-words : fault-tolerance, stable transactional memory, open architecture, multiprocessor architecture, atomic actions.

---

1 This research was supported in part by the DRET under the grant n° 90 346 and the MRT under the grant n° 89 S 0625.

Résumé

Au départ réservé, à quelques domaines spécialisés, le besoin de fiabilité s'est répandu avec la généralisation des micro-ordinateurs et des stations de travail en environnement distribué. Par exemple, les stations de travail sont parralisées par la panne d'un serveur de fichier. Par conséquence, il y a un besoin pour des systèmes à tolérance de pannes autorisant l'exécution d'applications diverses et variées. C'est le but du système FTM (Fault Tolerant Multiprocessor).

L'architecture du FTM est construite en associant des machines ouvertes reliées par un sous-système d'interconnexion. La redondance dynamique est utilisée dans la construction des éléments processeurs. Lorsqu'un processeur a une défaillance, le sous-système d'interconnexion permet à un processeur de secours de recouvrer un état sain et de relancer les calculs interrompus.

Ce rapport décrit l'architecture du FTM et en particulier la mise en œuvre du sous-système d'interconnexion au moyen d'une mémoire stable transactionnelle.

Mots clés : tolérance aux fautes, mémoire stable transactionnelle, architecture ouverte, architecture multiprocesseurs, actions atomiques.

Table of contents

1 Introduction 3
2 The FTM design 3
2.1 Results of the GOTHIC experiment 3
2.2 Basic approach in FTM 5
2.3 Hardware design rules 6
3 The FTM prototype 9
3.1 The Pair of Stable Nodes 10
3.2 The mirrored disk subsystem 10
3.3 The structure of the Stable Transactional Memory (STM) 11
3.4 The FTL structure 13
3.5 Power supplies 15
4 STM functionality 16
4.1 STM object management 16
4.2 STM transactions 19
4.3 Group of transactions 22
4.4 STM commands synchronization 23
4.5 Coping with controller failures 23
5 Summary and conclusion 25

1 Introduction

The fault-tolerant systems which have been built so far can be placed into two groups: special purpose systems and general purpose systems. Special purpose systems, historically the first systems to employ fault tolerance, are designed to achieve a very high degree of reliability and availability. Examples of such systems are SIFT [18], developed for aircraft control, and ESS [8] for telephone switching.

Reliability requirement of general purpose systems is most of the times lower than special purpose one. They are able to tolerate only single hardware faults. Two kind of general purpose systems can be encountered: on line transaction processing systems and fault tolerant UNIX systems. The goal of on line transaction processing systems such as TANDEM NON-STOP [5] and STRATUS [11] is to support commercial applications which need data consistency and high integrity. Applications are oriented towards database management and have to be designed using transactions. UNIX fault tolerant systems provide hardware reliability to users without modification of existing application software. Solutions are hardware based (TANDEM S2, SEQUOIA [6]) or software based (TARGON [7]).

Due to generalization of micro-computers and workstations in distributed environments, more users are now concerned with reliability problems. For instance, diskless workstations are paralyzed by the failure of a file server or the crash of an accountancy system can seriously affect a company. Consequently, there is a need for general purpose fault tolerant systems which can support a wide range of applications. This is our goal in the design of the Fault Tolerant Multiprocessor (FTM).

The report is composed as follow. In section 2 we present the design principles of the FTM. Then we describe in section 3 the architecture of a FTM prototype. The functions of the Stable Transactional Memory board designed in the FTM are detailed in section 4. Section 5 is devoted to conclusion.

2 The FTM design

2.1 Results of the GOTHIC experiment

GOTHIC is a distributed system designed at INRIA [3] which investigates a new concept, the *multiprocedure* which integrates nicely procedural control and parallelism. Multiprocedure is in particular, well-adapted to fragmented-data handling. Opposite to traditional data which have a centralized representation,

fragmented data may be split up into several pieces, which may eventually be dealt with in parallel.

In order to implement reliable multiprocedure call [14] a *stable storage* was needed. A stable storage provides memory that has a high probability of surviving processor failures and has the important property that write operation are atomic, that is, either they happen or they do not; partially completed write operations cannot occur. A popular disk-based implementation of this abstraction was proposed by Lampson and Sturgis [12] some years ago.

However, the GOTHIC implementation of stable storage had to be efficient for managing small data structures such as the kernel objects involved in a distributed commit protocol. This requirement prevents the use of disk stable storage which have unacceptable performance for small objects. Therefore, we have introduced a fast Stable Storage Board (SSB), which offers atomic operations on small data structures with very good response time [15].

The SSB is built from two banks of non-volatile, random access memory [2], each bank consisting possibly of several megabytes of memory (the board has battery backup power in case of power failure). Although the algorithms adopted in the SSB to implement atomicity are essentially in the same spirit as Lampson and Sturgis's, the SSB differs in two ways: (i) the SSB is part of the processor address space, it is then necessary to implement a way of controlling secure access to the board to prevent stable storage alteration by a faulty processor; (ii) the SSB provides atomic operations on groups of objects.

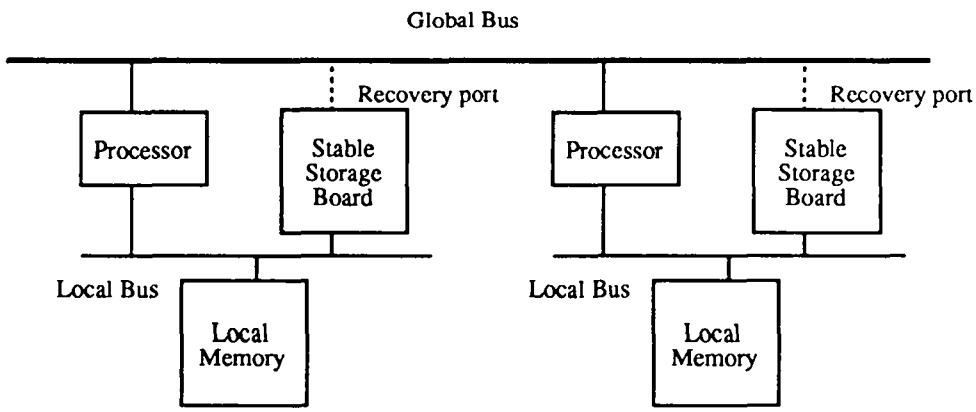


Figure 1: GOTHIC Multiprocessor Architecture

The SSB has been integrated into a multiprocessor architecture by associating a stable storage board with each processor (see Figure 1). Every process in the

system can allocate stable memory and manipulate stable objects. Moreover, a checkpoint service has been designed to provide explicit checkpointing to user's tasks. The system can tolerate the following hardware failures:

Power failure: at restart, SSB aborts the active atomic operation and internally restores a consistent state of stable objects. Then system services can resume from persistent stable objects. For instance, the checkpoint service restarts user tasks from checkpoint stable objects.

Processor failure: the SSB can detect failure of its associated processor either by violation of the access protocol to stable objects or by the time-out of an internal watch-dog which has to be regularly reset by the processor. If the processor fails, the SSB sends an interrupt on the global bus to "ask" a safe processor to take control of the SSB through the recovery port connected to the global bus. Then the backup processor can restart services and user processes. This "non-stop" ability works only if the global bus is not frozen by the processor failure.

Despite some limitations, experience with the GOTHIC system has convinced us that programming reliable applications can take advantage of the built-in atomic operations features of the stable storage. In summary, the prototype has shown that it should be possible to design a fault-tolerant computer from a standard open architecture by associating a stable storage board with each processor. That is the main reason why this approach is generalized here in the FTM in order to construct a general purpose fault tolerant system.

2.2 Basic approach in FTM

To design a fault tolerant system four constituent phases have to be implemented [13]: error detection; damage confinement and assessment; error recovery; fault treatment and continued system service.

In all fault tolerant architectures redundancy is needed to ensure continuation of service after error detection and confinement of the faulty component. When using static redundancy, multiple components are doing the same work at the same time. That allows to mask one failure without reconfiguration of the system and operating system intervention. Examples of such systems are the TANDEM S2 which uses three way replication (Triple Modular Redundancy) and the STRATUS [11]. At the contrary, when using dynamic redundancy, similar components are doing different tasks at the same time and the load of a faulty component must be handled by a safe one in addition of its own work. For instance in TANDEM NonStop [5], peripheral devices are managed by process pairs. One process is running on the active processor, the other on the backup processor. This

implies that the active process regularly sends its state to the backup by an interconnection subsystem, a message passing bus in the TANDEM example. Dynamic redundancy cost is lower than static redundancy one, as in normal operation all the processors are executing different tasks ; but message sending introduces some checkpointing overhead. However faults are not masked by the hardware and recovery needs an operating system reconfiguration.

Our approach is to build the FTM hardware architecture from *standard open machines* using dynamic redundancy in the building of processing elements. To reduce checkpointing overhead we use a dual ported stable storage board as an interconnection subsystem. A stable storage board is associated with each processor which can locally manipulate stable objects with fast access time. In normal operation, the stable storage is not shared between a processor and its backup. When a stable storage board detects the crash of its associated processor it restores a consistent state of stable objects and warns the backup processor. Then the backup processor dynamically recreates system services from stable objects and user tasks from checkpoints.

2.3 Hardware design rules

In this section, we derive general rules which describe how to construct a FTM architecture in a modular manner by associating a stable storage board to existing machines. Our purpose is to build a FTM architecture which can tolerate a single failure of any component (processor, disk, bus), with the exception of stable storage board which is reliable and has the ability to tolerate an internal single fault (see 3.3).

The FTM system has to provide continued service to users. Thus, both a backup component and the faulty one must be independent as far as hardware faults are concerned. For instance, one processor and its backup must not be connected to the same communication bus, since for ordinary machines, the bus may be frozen by a faulty processor. Peripheral devices (disks) must be also accessible in more than one way.

Rule 1 allows us to build a minimal FTM machine which can tolerate any single component hardware fault:

> **Rule 1:** *A minimal FTM architecture is built from two standard uni-processors, independently powered, and connected by stable storage boards.*

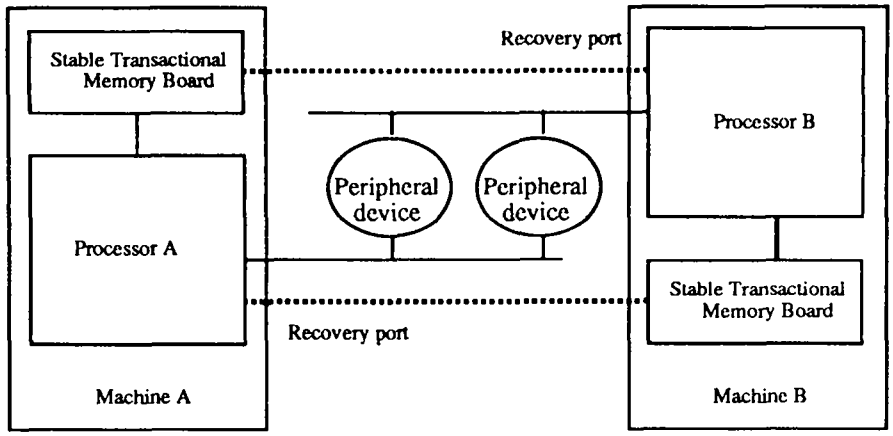


Figure 2: A minimal FTM configuration

A minimal FTM configuration is shown in Figure 2; this architecture allows to switch off independently one of the two machines in case of a failure to permit manual maintenance to take place.

The Stable Transactional Memory board (STM) is the evolution of the SSB of GOTHIC. Its main features (see section 4) can be summarized as autonomy and atomicity. The STM is able to take decisions. For instance, it can decide that a processor accessing it is faulty and then initiate a reconfiguration. The STM provides an atomic transaction facility on a set of objects (stable structures). Thus a consistent state of a set of objects can be recovered after a processor failure.

The STM board is dual ported to permit access by another processor. In the event of its associated processor failing, the STM switches to the other port and sends an interrupt to the backup processor which can recover the stable objects and restart computations from them. Detection of a processor crash is done by a watch-dog which has to be reset regularly by the processor.

The previous minimal configuration can be extended by using loosely-coupled multiprocessor machines instead of uniprocessors. A second rule for constructing a multiprocessor FTM is the following:

Rule 2: *A multiprocessor FTM configuration is built from two standard multiprocessor machines in which processors are paired by their STM recovery port.*

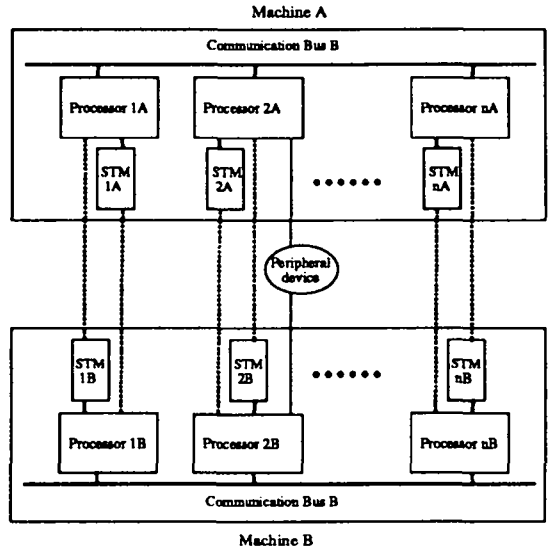


Figure 3: A FTM multiprocessor configuration

It is desirable that the two FTM machines act like a single multiprocessor architecture in normal operation to facilitate system management and to permit freely distribution of computations among the processors. We need thus a fast communication link between the buses which offers a minimal overhead for transferring messages from one machine to the other. Consequently, we can derive a third rule:

Rule 3: *The FTM machines can be connected by a fast Fault Tolerant Link (FTL) which is used to implement a virtual multiprocessor bus.*

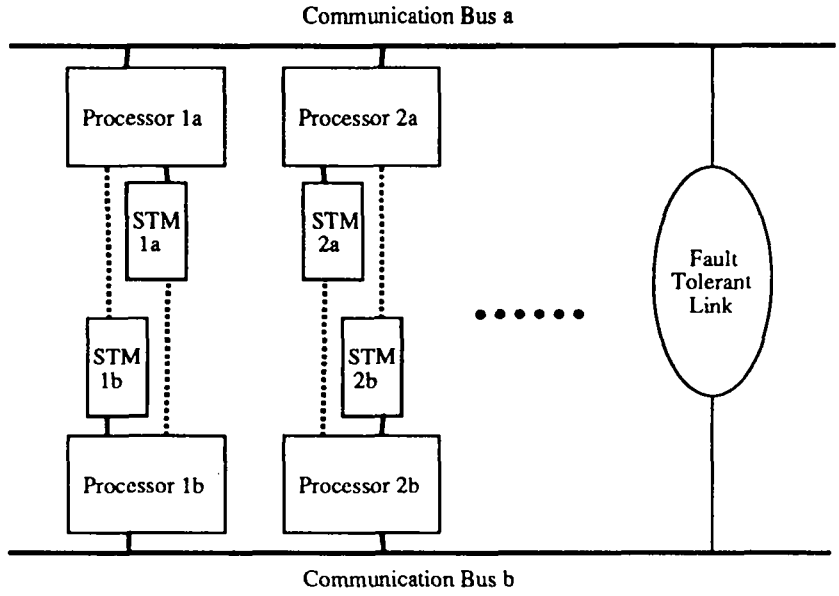


Figure 4: Virtual multiprocessor configuration

Should a bus of one FTM machine fail, the FTL isolates the faulty machine from the safe one. An FTL failure results in FTM partition. The architecture becomes equivalent to the architecture of Figure 3. If the FTL fails no side effects must occur on the FTM machine to permit service continuation. In section 3.3 we describe a specific implementation of the FTL which offers transfer times of the same order as the communication buses.

From these three rules, many implementations can be derived, depending on the number and type of the FTM machines. To build an instance of the FTM architecture, two boards, the FTL and the STM, have to be designed and connected to the buses of the FTM machines. However the STM and the FTL will not differ significantly from one implementation to another one. Only the bus interfaces have to be implemented.

3 The FTM prototype

In the last section we have given the general design principles of our FTM. We now detail one physical implementation of the architecture.

The multiprocessor bus of the FTM machines is the message passing MULTIBUS II (iPSB) [16]. The bus is driven by a specific integrated controller, the MPC. The processor does not have an electrical access to the bus. This gives the bus better protection from a hardware processor failure.

3.1 The Pair of Stable Nodes

The two FTM machines are associated in such a way that any processor of one machine is coupled to the corresponding one of the other machine by its STM recovery port. These two processors and their STMs form a pair of stable nodes (see Figure 5). A *stable node* is a virtual component, which includes an STM and a virtual processor. The virtual processor is either the main processor (in normal operation) or the backup processor (when the main one fails). The backup processor is then shared by the two stable nodes. For instance, the stable node 1a is composed in normal operation from the processor 1a and the STM 1a. If the processor 1a fails, the backup processor 1b becomes the virtual processor of the stable node 1a.

The major advantage of the pair of stable nodes is that it statically binds the backup processor to a stable node. Thus, the communication subsystem is simplified since the physical address for a message to a stable node is either the main processor or the backup one.

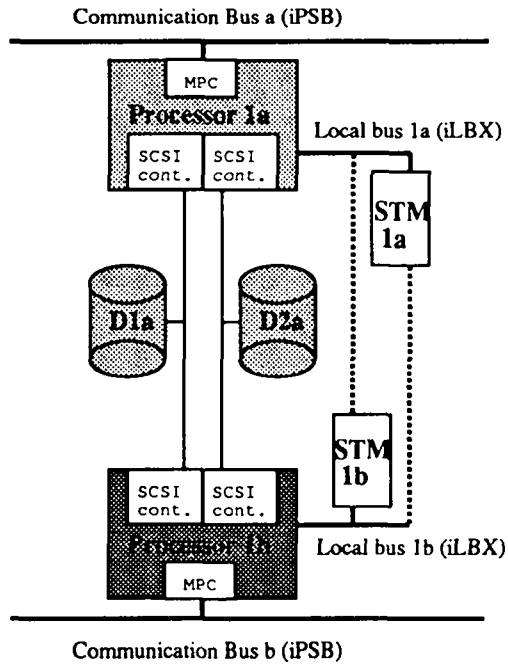


Figure 5: Structure of a pair of stable nodes

3.2 The mirrored disk subsystem

A mirrored disk is implemented using two physically independent disks which are connected to two different SCSI buses. Each processor board possesses a

separate SCSI controller for each SCSI bus. This enables any single failure in either a disk, a SCSI bus or a SCSI controller to be tolerated.

A mirrored disk is associated with a stable node. The mirrored disk is not shared by the two stable nodes. Only the virtual processor of the stable node owns the mirrored disk and can modify it. For instance, in Figure 5, one stable disk is associated with the stable node 1*a*.

3.3 The structure of the Stable Transactional Memory (STM)

Stable storage must provide operations which execute atomically (all or nothing property). To implement them a two-phase commit protocol is used internally to atomically update two different memory devices:

Phase 1 The data is written to the first memory device,

Phase 2 When the first phase has successfully completed, data is copied from the first memory device to the second one.

If a processor failure occurs during phase 1, the previous version of the data, which is still on the second memory device, is copied back to the first. The update is not successful. If a failure occurs during phase 2, the first memory device will still contain the new version of data, so phase 2 can be restarted. Only on completion of phase 2 is the update considered successful.

The STM is an evolution of the SSB of GOTHIC [2][15] described in 2.2. The STM contains two memory banks and an intelligent controller. The main hardware evolution of the STM is the ability to tolerate an internal error of the STM itself. The components of the STM, memory banks and intelligent controller, possess two properties: non-stop and maintainability.

The memory banks are made from a set of memory chips. In order to tolerate soft (non permanent) errors when reading memory, an Error Correcting Code (ECC) mechanism is used to correct a false bit. Soft errors are the result of spontaneous alterations of the memory chip and occur from time to time. The ECC is able to correct an error on one bit and detect errors on twos. If one chip fails, the error is hard (permanent), the ECC is able to correct the false bit, deliver the good result to the processor, but can not write it back to the memory chip. Hence the ECC can not recover subsequent soft failures. In such cases the *bit steering* technique [10] is used to copy the contents of the faulty memory chip to a spare one on the board. The fault is reported to the kernel and the operator is asked to replace the faulty chip. Using the previous mechanisms a chip failure is masked to the processor. Another solution would be to copy the value of the other bank to the faulty one and abort atomic actions (transactions). Hovewer this is not always possible. For instance if

bank 1 is alterated in phase 2 of the commit protocol a safe value can not be retreived from bank 2.

The intelligent controller has direct access to both memory banks. Every processor access goes through the controller. If the controller fails, the contents of memory may be destroyed. To prevent this, it is necessary to design a fail-stop controller, made from two identical VLSI chips (see Figure 6). Both chips are always doing the same work. Their outputs are compared before being sent to the memory banks. If any difference is found, the comparator disconnects the circuits and declares the fail-stop controller to be in failure. If the chips outputs are the same, access to the memory banks is permitted.

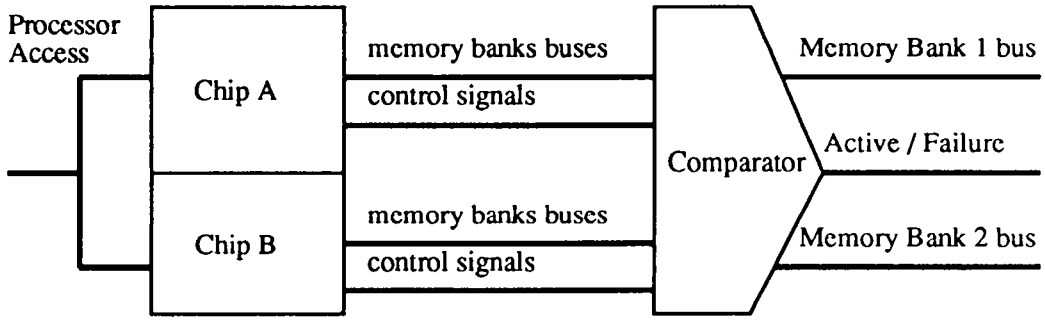


Figure 6: The fail-stop intelligent controller

Access to the STM must be still ensured, even in the event of controller failure, to permit service continuation. A second controller has to be able to replace a faulty one. The STM possesses two access paths which are connected directly to the main and backup processors by their local iLBXII bus. Both access paths are equivalent since the backup processor, when active, has the same function as the main one. To implement non-stop, the idea is to associate a fail-stop controller with each access path (see Figure 7). The memory banks are connected to both controllers. In normal operation, the controller associated with the main access is active: it executes processor commands and verifies access to stable objects (see 4.1). If the processor fails, the main controller becomes passive and control is transferred to the backup controller. A failure of the main controller is equivalent to a failure of the processor - the backup controller, and processor, become active. The backup controller aborts current internal STM operation and restores a consistent state of objects (state restoration algorithms are described in section 4.5).

To optimize STM performance, the two controllers can be used to implement some internal parallelism. For instance, functions like copy from bank 1 to bank 2

can be performed by the passive controller. If the passive controller fails the primary controller has to restart its work.

The STM is built from two electrically independent board to permit manual maintenance on one board. The two boards are identical and contain a memory bank and a controller. One board is connected to the main processor local bus, the other one to the backup processor. During maintenance of one board the owning processor is halted; at restart the replaced board is updated with the contents of the safe one.

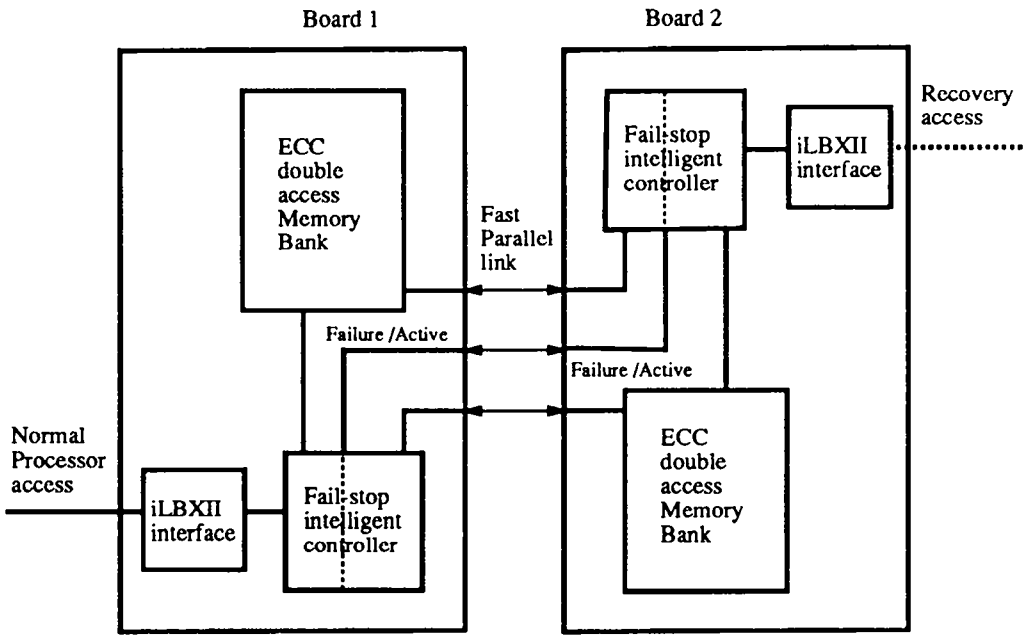


Figure 7: The STM structure

3.4 The FTL structure

The function of the FTL is to connect the two FTM machines to provide a virtual multiprocessor bus. The simplest way of implementing the FTL is to use a local area network, like the ethernet, and route messages from one machine to the other. However, local network transfer times are very slow compared to MULTIBUS II. To achieve faster communication between FTM machines, we implement the FTL using two boards connected by a 32 bit parallel bus (see Figure 8). The FTL boards are themselves connected to the local bus. To route messages we use one of the processors which can be considered as a gateway to the other FTM machine.

To avoid synchronization problems between the communication bus and the FTL, a small RAM memory buffers the messages. Input and output buffers are distinct in order to isolate the bus from a faulty FTL board. Moreover the boards can be switched on and off independently. When a message is received from the communication bus, it is written in the output buffer of the local FTL board. Then a DMA chip copies it to the input buffer of the other FTL board.

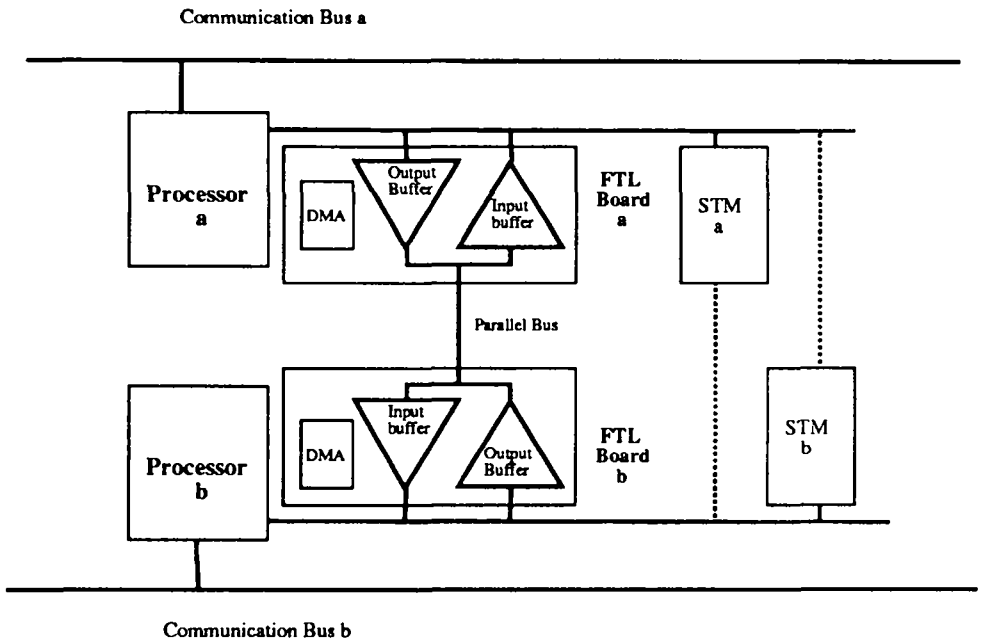


Figure 8: The FTL structure

What performances can be expected from the FTL implementation? A synchronous SCSI bus (8 bits) has a bandwidth 4 Mbytes/s. Thus using a SCSI technology to design the FTL (32 bits), a bandwidth of 16 Mbytes/s is possible. The maximum bandwidth of the Multibus II communication bus is 40Mbytes/s. Hence the minimal transfer time of one 1 Kbytes page between two processors of the same machine is 25 µs. Communication between two processors of two FTM machines which are not connected to the FTL board needs 3 transfers: processor → FTL processor, FTL → FTL, FTL processor → processor. The transfer time from one FTL board to another is 62.5 µs. Thus the maximum overhead in transferring one page between two processors of the two FTM machines is $\frac{25 + 62.5 + 25}{25} = 4.5$ and the minimum is $\frac{62.5}{25}$=2.5.

3.5 Power supplies

All computer systems work better when power is on! We can distinguish two causes of power failure: malfunction of one power supply and external network power failure.

In the FTM architecture, each FTM machine possesses its own power supply unit. Thus, failure of a unit implies halting only one FTM machine. The same solution is applied to disks to allow flexibility in maintaining them. Concerning the STM boards, they must not halt if one power supply fails. Therefore, each STM board is powered by two distincts power units (see Figure 9): the power supply of the FTM machine to which the STM board is connected and a power supply reserved for all STM boards of one machine.

To tolerate network power failures, another source of power has to be provided. If the system and applications must never halt, whatever the duration of the failure, an alternative power source must be provided. To tolerate short and medium length break-downs, battery power units can be sufficient. When a power failure occurs, the system can be still powered by batteries and works normally. If the duration of the break-down reaches the limit of the batteries a consistent state of the operating system and applications is saved on the disks and the FTM is halted safely, permitting a restart when power is available.

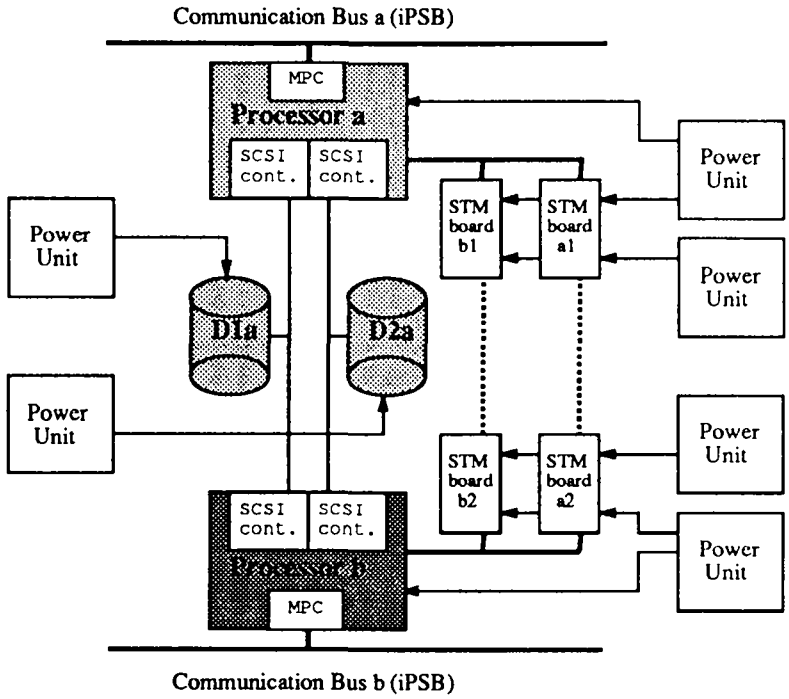


Figure 9: Power supplies of the FTM prototype

4 STM functionality

The purpose of this section is to describe in detail the functions of the STM whose structure has been introduced in section 3.3. Our experience in the design of the GOTHIC kernel led us to define the precise functions that a stable memory board should possess in order to build a reliable kernel:

* Atomic transactions on multiple objects
* Simple programming interface
* Autonomy
* Protection
* Fast access

The STM board is a flexible hardware device which provides fast atomic operations on a set of data structures. To permit fast access the STM is directly connected to the processor busses. Thus the internal memory has to be protected from processor failures. In a classical random access memory, any word is accessible at any time. To help protect memory, we can limit the number of words which are accessible at any one time. The solution provided by the STM is to record logical structures named STM objects so as to provide memory protection.

All operations on objects have to be atomic in order to recover a consistent state of memory after a processor failure. The *transaction* is the basic mechanism provided by the STM so as to manage objects atomically. Any object operation (read, write, creation, destruction) *must* be performed whitin an STM transaction. Multiple transactions may be simultaneously active in STM to permit sharing of the STM by several processes. Transaction management is detail in section 4.2. In the following, the user of the STM facilities is refered to as the STM client (e.g. the kernel).

4.1 STM object management

One of the major problems in the design of the STM is to represent an STM object. In the GOTHIC SSB (described in section 2.1), an object is a set of non-contiguous words. A tag is associated with each memory word of an object and contains a link to the next word in the object. The protection mechanism required that a read or update operation has to read sequentially all words of the object, even if few words are really accessed. Thus, access to the SSB was not easy to program.

In the FTM, an STM object is a *contiguous set of words.* To *read* or *modify* any word of an STM object, the program has to explicitly *open* the object. Any attempt by the processor to access a word which does not belong to a currently opened object is signaled as an exception. This protection mechanism is

implemented by the intelligent controller. The controller records which of the objects stored in STM are currently opened. In order to *create* an STM object, it is necessary first to allocate the space needed within the STM address space and, second to *declare* the object to the intelligent controller. Note that the STM client is responsible for the space allocation policy of the STM memory.

An STM object is represented by its contents and a *descriptor* which descriptor contains the lower and upper bounds of the memory region allocated to the object together with some information about the object state (see Figure 10). When an object is opened, the bounds are loaded into two comparators so that to check the validity of a processor address (see Figure 11).In other words we want to be able to access any word of an open object. The controller checks if the incoming address lies within the bounds of the object. Up to k objects may be opened simultaneously. This choice facilitates the implementation of $(k\text{-}1)$-ary operations such as obj := f(Obj_1, ..., Obj_{k-1}).The value k is fixed by the implementation of the controller (VLSI chip), but should be greater than 3.

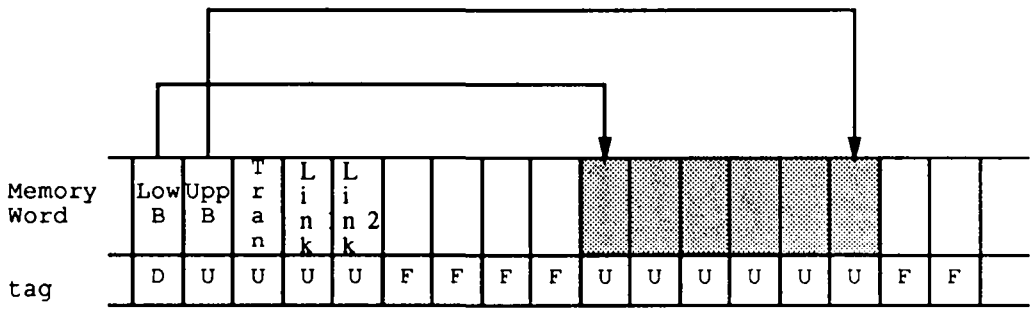


Figure 10: Structure of an STM object

Object descriptors of all STM objects can not be recorded in the STM controller and are stored in the STM memory itself. A fixed size object descriptor is allocated by the client at object creation. The object descriptor and the object itself need not be contiguous in memory. We introduce here after the object commands provided by the STM. To declare an STM object the processor sends a *Declare_STM_object* command to the controller:

```
Declare_STM_object (Obj_descr_add, Low_bound_add, Upper_bound_add);
```

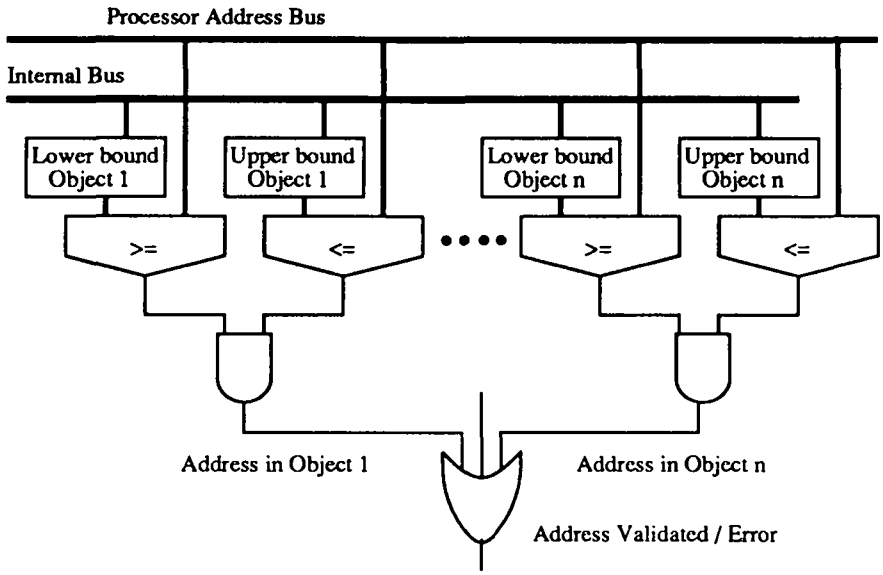


Figure 11: Address checking by the STM controller

When receiving the Declare_STM_object command, the controller first verifies that the allocated memory is free in order to prevent existing objects being overwritten and then initializes the object descriptor. To implement this protection mechanism, a *tag* is associated with each memory word to describe its contents. The tag has four states:

F	The memory word is free and can be allocated,
D	The memory word is the start of an object_descriptor,
T	The word is the start of a transaction_descriptor (see 4.2),
U	The word is in use, either in an object or in a descriptor.

The *Open_STM_object* command gives a processor access to an STM object:

```
Open_STM_object (Obj_descr_add);
```

When receiving this command, the controller checks that the parameter points on the beginning of an object_descriptor. A set of control registers (bound registers, comparators, etc) is allocated and the object descriptor is loaded into these registers. If no register set can be allocated (k objects are currently opened), an exception is signaled to the processor. Two commands so as to close and delete an object are provided:

```
Close_STM_object (Obj_descr_add);
Delete_STM_object (Obj_descr_add);
```

The *Close_STM_object* command frees the controller register set held by the object descriptor and forbids any subsequent access to that object until it has been reopened[2]. The *Delete_STM_object* removes an object from the STM and frees the associated space.

4.2 STM transactions

Object operations described previously must be performed within *transactions* so as to be recoverable. Similary to STM objects, a transaction is represented by a *transaction descriptor* (Figure 12). A transaction descriptor is analogous to an object descriptor, it has to be allocated by the STM client and be declared to the controller by a *Declare_STM_transaction* command:

```
Declare_STM_transaction (Trans_descr_add);
```

2 To ensure correct behavior of this protection mechanism, the processor cache has to be disabled when accessing STM memory. Otherwise the processor could access data from the cache, while the corresponding object is closed. Consequently access to STM memory is slower than access to cached RAM memory.

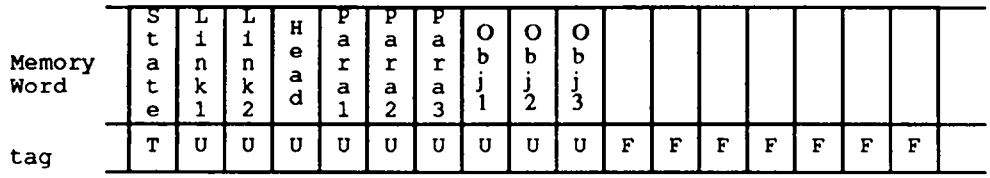


Figure 12: Structure of a transaction descriptor

The *Declare_STM_transaction* command is also recoverable and has to be done itself within a transaction. Transaction nesting is not allowed so that a transaction descriptor cannot be used until its creating transaction commits. To permit STM initialisation by the STM client an initial transaction descriptor is internally created at the bottom of STM memory.

There are five basic STM commands for transaction management:

```
Declare_STM_transaction (Trans_descr_add);
Delete_STM_transaction (Trans_descr_add);
Begin_STM_transaction (Trans_descr_add);
Commit_STM_transaction ();
Abort_STM_transaction ();
Change_STM_transaction (Trans_descr_add);
```

The *Delete_STM_transaction* command deletes a transaction descriptor. The transaction has to be inactive or an error is signaled to the processor. The command *Begin_STM_transaction* activates a transaction (see Figure 13). Operations are only applied to the contents of memory bank 1. All objects accessed by a transaction are linked together in a double link list. The *head* field of the transaction descriptor points to the first object descriptor in the list. A simplified implementation (written in the C language) of internal operations performed by the controller when opening an object within a transaction is the following:

```
void Open_object (Type_object_descriptor Obj_desc)
{
 if (Obj_desc.Trans == nil) then
 {
   /* Object never accessed */
   /* We have to link it in front of the access list  */
   Trans_desc.Head->link2 = &Obj_desc;
   Obj_desc.link1 = Trans_desc.Head;
   Obj_desc.link2 = nil;
   Trans_desc.Head = &Obj_desc;

   /* Tag the object with the opening transaction */
   Obj_desc.Trans = &Current_trans_desc;
   return (OK);
 }
```

```
else
if (Obj_desc.Trans == &Current_trans_desc) then
{
  /* The object have already been accessed by this transaction */
  /* There is nothing to do */
  return (OK);
}
else
{
  /* The object is currently accessed by another transaction */
  return (Conflict_detection);
}
}
```

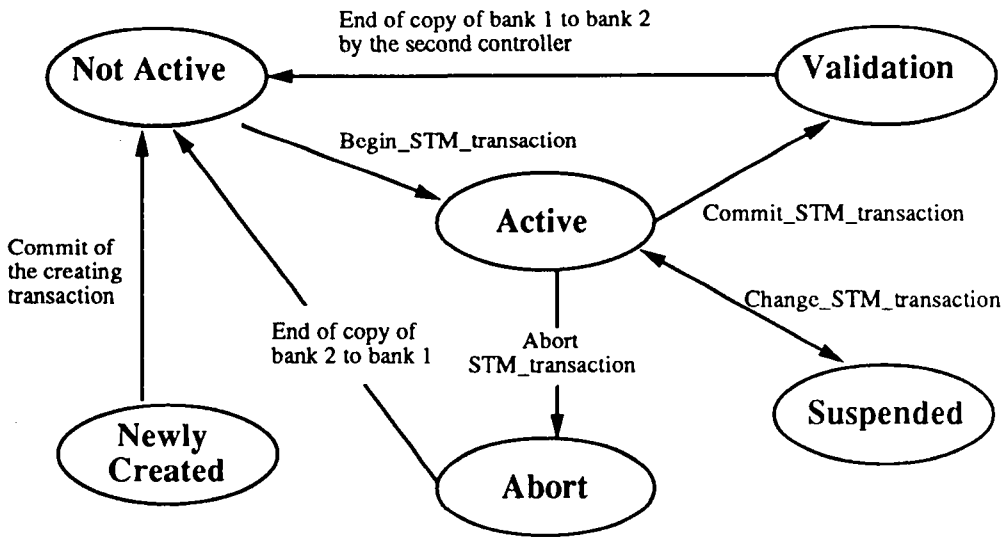


Figure 13: State machine for a transaction

To successfully end an active transaction the command *Commit_STM_transaction* is sent to the controller. This command takes no parameters and refers to the current transaction. The objects accessed by the transaction are identified using the linked list and then are internally copied from the first bank to the second bank. This copy can be done by the passive controller, while the main one continues servicing other processor commands in parallel.

Abort_STM_transaction cancels the execution of the current active transaction. All modified objects, created and deleted descriptors are restored to their original state by copying from bank 2 to bank 1. Note that when a transaction is aborted, objects which have been deleted by it have to be recovered to their prior state. This implies that the memory space of a deleted object and its descriptor cannot be over-written by another object while the transaction is active. In order to ensure roll-back, the effective destruction of the object is done at commit time and its area

is unusable until then. When the processor fails, no assumptions can be made about the consistency of the objects which were open: the transaction which manipulates these objects has to be aborted to recover a consistent state.

In order to have multiple transactions simultaneously active in the STM, it is possible to save the current transaction and switch to another one previously suspended with the *Change_STM_transaction* command. To save the transaction, registers sets and the *transaction status* are copied by the STM controller to the transaction descriptor. The transaction status consists of the following information (see Figure 12):

- State of the transaction which may be: *newly created, not active, active, suspended, validation, abort* (see Figure 13),
- A reference to objects opened at switch time. The transaction descriptor contains K specials fields store pointers to the object descriptors(Obj1, Obj2, Obj3 in Figure 12). When the transaction is restored, these objects will be automatically reopened,

4.3 Group of transactions

When designing the kernel it is useful to group different transactions, so that either all transactions commit, or all transactions abort. For instance, several processors may need to update atomically several RAM variables which are on different STMs. To implement the atomicity of a *group of transactions*, a distributed commit protocol is needed. Each transaction of the group follows the protocol:

Phase 0	Perform the transaction work.
Phase 1	If the transaction succeeds, send *Prepare_to_commit* to a coordinator, otherwise send *Abort*.
Phase 2	Receive the final decision from the coordinator and either commit or abort the transaction.

The STM provides an efficient implementation of the group commit protocol. A new state, *Prepare_to_commit,* is introduced in the transaction automata (see Figure 14). In this state, no object is or can be opened. Thus, if the processor fails, objects are still consistent and the transaction does not have to be aborted. To enter this state, the command *Pre_Commit_STM_transaction* must be sent to the controller. The final decision to either commit or abort is sent by the coordinator. The copy from bank 1 to bank 2 (or bank 1 to bank 2) is delayed until receipt of the *Commit_STM_transaction* (or *Abort_STM_transaction*) command.

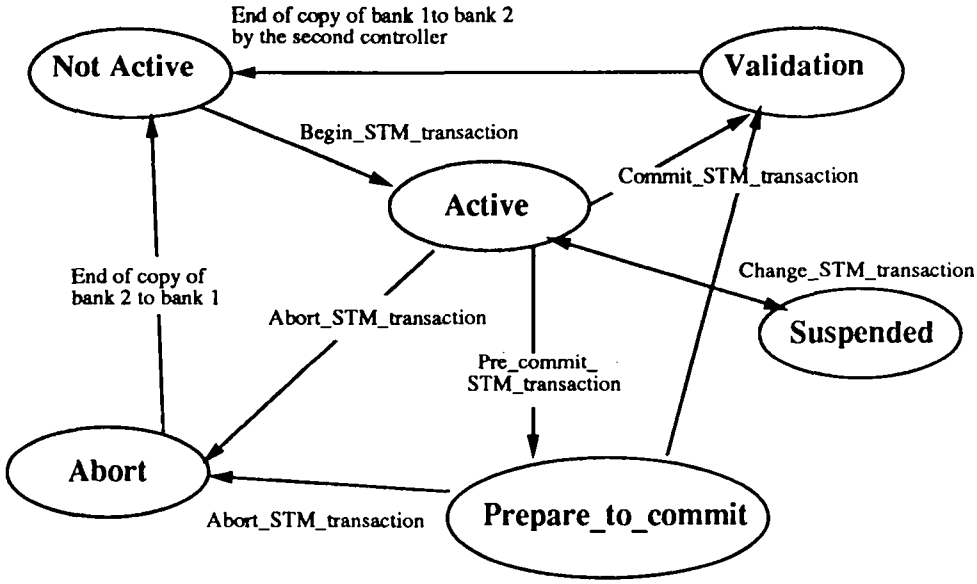


Figure 14: Complete state machine for a transaction

4.4 STM commands synchronization

A client must await the completion of a command sent to the STM in order to detect a possible error. The STM offers two programming modes: polling and interrupt. The mode is specified by setting a bit in the command word. The interrupt mode is quite useful for long commands such as *STM_commit_transaction* where it enables the processor to run another task while the STM copies the objects to bank 2.

Some synchronization is needed when multiple transactions want to share the same object. If one transaction attempts to open an object already used by another one, the STM controller will signal an exception. To access the object, the transaction has to wait until the commit of the transaction which owns the object. At the end of a transaction commit, the STM has the ability of sending an interrupt to the processor (kernel) so as to inform that the object is free to awaken one or more of the waiting clients. This wakeup is completely transparent to the transactions bidding for the object. There is no overhead if objects are not shared.

4.5 Coping with controller failures

A controller failure can be considered equivalent to a processor failure. To roll-back to a consistent state, the backup controller has to copy the previous version of modified objects from bank 2 to bank 1. The most complicated problem occurs when a failure arises while creating a descriptor (object or transaction

descriptor) since descriptor creation requires the execution of multiple basic operations to insert the descriptor into the descriptor list. To solve this STM algorithms use a robust structure [17] - a double linked list which allow to insert an element safely. Now we illustrate this point.

In Figure 15 we detail the basic steps for creation of a descriptor and its insertion in the descriptor list. If a failure occurs during the creation process, the backup controller must restore the initial state of the list as shown in Figure 15.a. In state 1 the backward link of descriptor b is modified to point to the non-created descriptor *c*. In state 2, we complete the initialization of descriptor *c*. If a controller failure occurs during states 1or 2, the backup controller is able to find descriptor *c* and remove it. To get the final state, the forward link of descriptor *a* is just moved from descriptor *b* to descriptor *c* by a single atomic step.

After a failure, the backup controller performs a consistency check of the internal descriptor (objects, transaction) lists of the STM. If some inconsistency is detected, a consistent state is restored as explained previously.

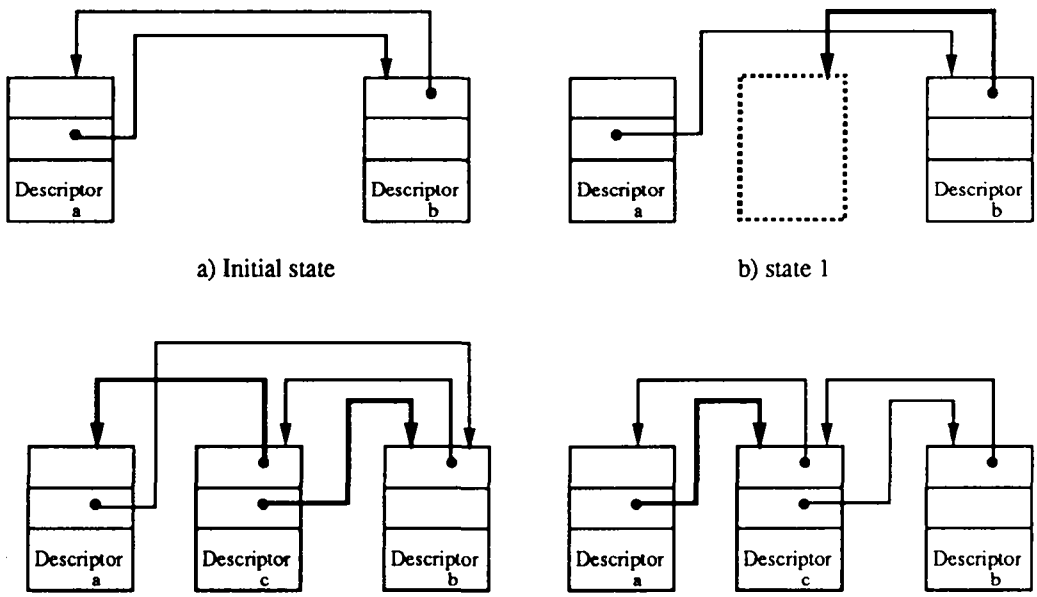


Figure 15: Double linked list element insertion

5 Summary and conclusion

This paper has reviewed the main aspects of the FTM architecture which is built by associating stable transactional memory board with processors of a standard machine. We have presented design rules of FTM architecture and an instance of its implementation based on market industry products.

We are currently working on the implementation of the intelligent controller VLSI chip. Its main advantage is to permit an easy implementation of a STM board for any kind of machines just by associating memory chips and a bus interface to the VLSI controller. The STM is emulated by software so as to ease the kernel development and test extensively internal STM algorithms.

The FTM kernel is based on the MACH kernel [19] and is designed in order to support reliable non-stop services. Reliable services are executed on a pair of stable nodes. The kernel provides access to STM facilities (stable objects and transactions) through an interface composed of a set of primitives written in the C^{++} object-oriented language so as to hide to the user the low-level STM hardware interface. The most significant reliable service designed so far on the FTM kernel is a reliable distributed virtual memory [4] which integrates in a uniform view all physical memories of the system (RAM, STM and disks).

Acknowledgments

The authors wish to thank P.A. Lee, M. Jegado and V. Issarny for their helpful comments on earlier drafts.

References

[1] J.P. Banâtre, M. Banâtre, G. Lapalme and Fl. Ployette. The Design and Building of ENCHERE, a Distributed Electronic Marketing System.. *Communications of the ACM*, vol. 29, (1), pp. 19--29, January 1986,

[2] J. P. Banâtre, M. Banâtre and G. Muller. Ensuring data security and integrity with a fast stable storage. *Proc. of 4th International Conference on Data Engineering*. pp. 285--293, Los Angeles, February 1988, IEEE,

[3] J.P Banâtre, M. Banâtre and G. Muller. Main Aspects of the GOTHIC Distributed System. *European Teleinformatics Conference on Research into Networks and Distributed Applications*. pp. 747--760, Vienna, Austria, April 1988,

[4] M. Banâtre, G. Muller, B. Rochat and P. Sanchez. A Reliable Distributed Virtual Memory on top of the Mach kernel. *OSF Micro Kernel Applications Workshop*. Grenoble, France, November 1990,

[5] J. Bartlett. A NonStop Kernel. *Proc. of 8th ACM Symposium on Operating Systems Principles*. December 1981,

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

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

[8] G.F. Clement and P.K. Giloth. *The Evolution of Fault-Tolerant Computing*. vol. 1, edited by Avizienis A., Kopetz H., et Laprie J.C., Springer Verlag, 1987,

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

[10] W.R. Hardell, D.A. Hicks, L.C. Howell, W.E. Maule, R. Montoye and D. P. Tuttle. Data Cache and Storage Control Units. *IBM RISC System/6000 Technology*. pp. 44--51, IBM, 1990,

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

[12] B. Lampson. Atomic Transactions. *Distributed Systems and Architecture and Implementation: An Advanced Course*. pp. 246--265, vol. 105, Springer Verlag, Lecture Notes in Computer Science, 1981,

[13] P.A. Lee and T. Anderson. *Fault Tolerance : Principles and Practice*. vol. 3, edited by J.C. Laprie A. Avizienis, H. Kopetz, Springer Verlag, New York, 1990,

[14] C. Morin. *Conception et réalisation d'un service d'appel de multiprocédure à distance.* Thèse de doctorat, Université de Rennes I, 1990,

[15] G. Muller. *Conception et réalisation d'une machine multiprocesseur sûre de fonctionnement.* Thèse de doctorat, Université de Rennes I, June 1988,

[16] *Spécial MULTIBUS II.* vol. 17, edited by Métrologie, DMA, Mars 1985,

[17] D.J. Taylor, D.E. Morgan and J.P. Black. Redundancy in Data Structures: Improving Software Fault Tolerance. *IEEE Transaction on Software Engineering.* pp. 585--594, vol. 6, IEEE, November 1980,

[18] J. Wensley. SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control. *Proc. IEEE.* pp. 1240--1254, October 1978,

[19] M. Young, R. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. *Proc. of 11th Operating System Review.* pp. 63--76, Austin, November 1987,

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

1990 :

PI 560 A SIMPLE TAXONOMY FOR DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS
Michel RAYNAL
Novembre 1990.

PI 561 MULTIMODAL ESTIMATION OF DISCONTINOUS OPTICAL FLOW USING MARKOV RANDOM FIELDS
Fabrice HEITZ, Patrick BOUTHEMY
Novembre 1990, 50 Pages.

PI 562 EFFICIENT GLOBALCOMPUTATIONS ON A PROCESSOR NETWORK WITH PROGRAMMABLE LOGIC
J M. FILLOQUE, E. GAUTRIN, B POTTIER
Novembre 1990, 14 pages.

PI 563 EQUATIONAL SETS OF TREE-VECTORS
Anne GRAZON, Jean-Claude RAOULT
Novembre 1990, 20 Pages.

PI 564 MULTIFRAME-BASED IDENTIFICATION OF MOBILE COMPONENTS OF A SCENE WITH A MOVING CAMERA
Edouard FRANCOIS, Patrick BOUTHEMY
Décembre 1990, 30 pages.

PI 565 NAIVE RESERVE CAN BE LINEAR
Pascal BRISSET, Olivier RIDOUX
Novembre 1990, 18 pages.

PI 566 METHODES D'INTEGRATION TEMPORELLE EN TRAITEMENT D'ANTENNE
Olivier ZUGMEYER, Jean-Pierre LE CADRE
Décembre 1990, 54 pages. Rapport n° 1

PI 567 METHODES PARAMETRIQUES POUR LA DETECTION DE SOURCES EN MOUVEMENT [1]
Olivier ZUGMEYER, Jean-Pierre LE CADRE
Décembre 1990, 42 pages. Rapport n° 2

PI 568 QUOI RETENIR D'UN ARBRE DE CLASSIFICATION ? UN ESSAI EN QUANTIFICATION D'IMAGE NUMERISEE
Israël César LERMAN, Nadia GHAZZALI
Décembre 1990, 36 pages, Projet CADO.

PI 569 VARIABLES RELATIONNELLES CODAGE ET ASSOCIATION
Israël César LERMAN, Mohamed OUALI-ALLAH
Décembre 1990, 40 pages

1991 :

PI 570 DESIGN DECISION FOR THE FTM : A GENERAL PURPOSE FAUTL TOLERANT MACHINE [1]
Michel BANATRE, Gilles MULLER, Bruno ROCHAT, Patrick SANCHEZ
Janvier 1991, 30 pages

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0949-6399