Integrating Real-Time Requirements and Fault-Tolerance

Célio Estevan Moron *

Grupo de Sistemas Distribuídos e Redes
Departamento de Computação
Universidade Federal de São Carlos
13565-905 São Carlos, S.P.
E-mail: dcem@power.ufscar.br

Resumo

Existem dois aspectos fundamentais no projeto de sistemas de tempo real que são: cumprir com as restrições de tempo real e ser confiável. Estes dois aspectos são conflitantes e geralmente são pesquisados separadamete. Para proporcionar tolerância a falhas por um lado, é necessário um tempo maior de execução; mas por outro lado os sistemas de tempo real requerem que as tarefas sejam completadas dentro de seus limites de tempo. Existe uma necessidade clara e urgente de mecanismos que sejam capazes de integrar os requisitos de tempo real e confiabilidade. Desta forma, seria possível abordar os requisitos de tempo real e tolerância a falhas ao mesmo tempo durante o projeto do sistema. Isto tornaria simples e eficiente o desenvolvimento de sistemas confiáveis de tempo real. A nossa abordagem se baseia na idéia que se o sistema é adaptável então a confiabilidade é alcançada mais facilmente. Foi desenvolvido um mecanismo, conhecido como "Real-Time Recoverable Action" (RTR-Action), o qual integra os requisitos de tempo real e confiabilidade.

Abstract

There are two fundamental aspects in the design of the real-time systems: to meet the real-time constraints and to be reliable. These two aspects are in conflict and usually are researched separately. To provide for fault-tolerance on one side, there is the need for a greater execution time; but on the other side real-time systems require tasks to be completed by their deadlines. There is a clear and urgent need for mechanisms that are able to integrate the real-time and the reliability requirements. So, it would be possible to address the real-time and fault-tolerance requirements together during the design of the system. This would make the development of reliable real-time systems simple and efficient. Our approach is based on the idea that if the system is adaptable then the reliability is easier to achieve. We have developed a mechanism, known as *Real-Time Recoverable Action* (RTR-Action), which integrates the real-time and the reliability requirements.

*This work was carried out while the author was following a Ph.D. program at the University of York, England; supported by a Brazilian Government Scholarship through CNPq - Conselho Nacional de Desenvolvimento Cientifico e Tecnologico.

1 Introduction

The fast and continuous development of hardware, as well as the demand for more sophisticated and reliable systems, are bringing about the need for a new generation of hard real-time systems. These newly-required systems are complex, dynamic and adaptable. Furthermore these emerging systems need to be highly reliable, because of the risk of human and financial losses, as well as the necessity of long periods without human intervention. Therefore it would be impossible to develop these systems using techniques currently available [8, 5].

There is a clear and urgent need for mechanisms that would make the development of reliable real-time systems simple and efficient. These mechanisms should be able to integrate the real-time and the reliability requirements together. This is to say, the system must not only deliver the right results but also do so at the right time.

Fault-tolerance is characterised by the introduction of redundancy, overhead, and a greater execution time. On the other hand, real-time systems are characterised by non-determinism, time constraints and the possibility of disaster in the case of time violation. There is an apparent conflict between these two characteristics. To provide for fault-tolerance on the one hand, there is the need for a greater execution time; but on the other hand real-time systems require that tasks are completed by their deadlines. Thus making the system fault-tolerant might, at the same time, increase the probability of more time errors. As these systems cannot compromise over fault-tolerance, the solution for this problem is to address both the real-time and the fault-tolerance requirements together at the design stage of the system.

Predictability can be measured by the fulfillment of the requirements of real-time systems. Predictability can be defined as the likelihood that the specification and design assumptions are not violated at run-time, provided the run-time conditions match the specifications.

There are two possible approaches to make a system predictable:

- considering that the future state of the system can be guaranteed; or
- considering that the future state of the system cannot be guaranteed.

In the first approach, the system uses off-line information to reconcile the competing demands of the various processes within the system; that is done by creating a schedule that guarantees that all processes meet their timing constraints. This approach carries some difficulties such as: a) even if we assume that complete information is available, we cannot make guarantees about external system behaviour; and b) the scheduler may require information that is either unavailable or just too difficult to collect; a predictable schedule only lasts as long as the world remains unchanged. This approach can only be used within small systems and has a limited use or value in a highly dynamic environment.

The second approach relies on the fact that if we cannot guarantee the future state of the system, we need to make the system adaptable to possible changes in the environment. Large and complex real-time systems need to be adaptable, or at least to attempt to be so to some degree. In an adaptable system, timing constraints on behavioural tasks might be well specified, but an upper bound on execution time for each process might not be known. Process timing constraints may not be well understood, or may be so complex that they may be approximated by very crude and pessimistic approaches.

Thus, for small systems we can achieve predictability by means of off-line knowledge, but for a highly dynamic and complex real-time environment, the predictability should be achieved by system adaptability.

2 Use of Fault-Tolerance in Real-Time Systems

Reliability in real-time systems has been achieved in the past largely by using ad hoc implementation, without a general approach able to deal with a large class of systems. Most of these systems have been implemented using the recovery block mechanism. Hecht [12] has proposed a fault-tolerant flight control system using the recovery block mechanism. Another implementation using a distributed recovery block mechanism was carried out by Welch [21]. This implementation by Welch was the design of a radar tracking system using four processors. The Deadline Mechanism [13] was proposed as a suitable approach to be used within real-time systems. This mechanism basically makes two assumptions: that the alternate algorithm is correct; and that the worst case execution time of the alternate block is previously known. The Exchanges mechanism [4] was another mechanism aimed at cyclic real-time systems. It fits into the old model of cyclic executive, but not into the current multi-task model which reflects the actual nature of real-time systems.

More recent researches are aimed at using more effectively the resources of the system as a whole. The redundancy introduced by the fault-tolerant mechanisms can become an unacceptable waste of resources in large real-time systems. The trend is to better utilise the slack resources in order to improve the performance of the system. In this direction there is a work by Bondavalli et al. [1] which proposes the Self-Configuring Optimistic Programming (SCOP). The aim of the research is to improve the cost-effectiveness of the fault-tolerant design. It uses dynamic redundancy in order to allow a trade-off among reliability, response time and throughput according to the needs of the application. This approach is difficult to be applied in systems with tight deadlines. Another work in the same direction is the one by Bondavalli et al. [2] which proposes the Fault-Tolerant Entity for Real-Time (FERT). Its aim is to improve the run-time efficiency of the system without losing in the reliability requirement. The application modules are separated from the control module. The control module specifies the interaction among themselves and the scheduler. A FERT represents a unit of schedulable activity. The designer specifies the time and functional requirements, ignoring the redundant organisation. So, the FERTs can be viewed as a redundancy-management layer of design.

There has been also some attempts to integrate the hardware and software fault-tolerant techniques in order to make an effective use of the CPU resources. Burns et al. [3] for example, proposed the *fail-omission* nodes which guarantee that any output each node produces is correct in the value and time domains. This allows reliability to be achieved without unacceptable increase in the number of processors used.

The model of the robust object, which can tolerate hardware and software failures, fits nicely into real-time systems. However, the overhead imposed by this structure makes its utilisation impossible within hard real-time systems. There is little research into a general model for the use of robust objects in hard real-time systems.

3 Real-Time Recoverable Action

In a distributed or parallel hard real-time system processes execute concurrently on distributed nodes. These processes have to coordinate their operations under time constraints. Atomic action has traditionally been used in the design of fault-tolerant systems. We will extend the concept of atomic action to be used in hard real-time systems.

We are proposing a programming structure *Real-Time Recoverable Action* (RTR-Action) which will provide fault-tolerance for real-time systems. The objective is to create a high-level abstract structure, making it easier to enforce the real-time constraints and error recovery.

The aim of the proposal is to extend the concept of action to the real-time control process environment, thus making possible the utilisation of the mechanism to achieve recovery from violations of the time constraints.

A system is composed of a set of processes which are to be executed on a parallel system. Each processor may have one or more processes. Processes can be either periodic or aperiodic. Processes communicate by message passing. Processes may preempt one another depending on their priorities. There is a fixed number of processes which are allocated to processors statically. Each process has a dynamic priority associated with it. This priority changes as the execution of the process is carried out, reflecting the dynamic behaviour of the system.

The parts of the system to be made fault-tolerant are identified by the use of the RTR-Action. A sub-set of processes can participate in an RTR-Action by means of an entry command in the body of each of these processes. The processes can communicate only inside of the RTR-Action and only with the processes that participate in it. This set of entry points establishes the recovery line of the RTR-Action and imposes stronger constraints than those already existing in the processes; these constraints represent the combination of the constraints of the set of processes. There is also the declaration of the exit point of each process in the RTR-Action.

3.1 Computational Model

The characteristics of the next generation of real-time systems are discussed in [20, 6]. These systems are believed to be large, complex, distributed, adaptive, to contain many types of timing constraints, to operate in a non-deterministic environment and to have a long system lifetime. The aim in building these systems is to integrate a large number of subsystems into a distributed system in order to reduce costs, over capacity of computer power and wiring. To this end, we view a real-time system as one that consists of a set of hard real-time tasks that must meet all their deadlines, and a set of soft real-time tasks which if they do not meet their deadlines will not provoke a disaster.

We also assume that: *the system is always able to execute at least the hard real-time set of tasks in the worst situation.* This implies that as many of the soft real-time tasks as possible will also be guaranteed. In complex real-time systems, normally the number of tasks with hard deadlines is smaller than the number of those with soft deadlines. We note that such an assumption is a rather minimal one and should be present in systems where guaranteeing deadlines are done off-line.

3.2 Communication Model

Real-time actions are little addressed in the literature because of the restrictions imposed by the overhead. If we talk about real-time action in a parallel environment the factor of communication delay will be added to the overhead. The atomic commitment limits the concurrency inside the action. In order to minimise the limitation in the concurrency, we opted for the use of a parallel system connected by a fast point-to-point network. The system is composed of a set of processors; where each processor has a large primary memory and links of communication to other processors. The class of processor which has these characteristics is the tranputer-like class. Within this class of processors, the following ones are currently available: INMOS T4xx, T8xx, and T9xxx; Intel i860; Texas TMS320C30 and TMS320C40.

Our parallel system will allow us to extract three major advantages which are:

- *Higher communication rates* - The processors have high rate communication links, and moreover, the possibility of parallel communication will increase the communication rate.
- *Scalable number of processors* - In order to match the specific capacity of a particular application we need to adjust the number of processors allocated to it.
- *No memory bottleneck* - The large primary memory allows us to keep all necessary data on it.

Each processor has its own local clock. There is an upper bound clock drift and we will call it ϵ.

3.3 Fault Model

Although we do not discard the importance of fault-avoidance and fault-removal, our technique is focused on fault-tolerance for real-time systems. More specifically, our technique is aimed at the logical and the time fault design in process control systems.

As our main objective is to deal with design time faults, we are not supporting node fault. However, our technique can be used together with other existing real-time fault-tolerant techniques in order to provide node fault-tolerance. The consequence of not supporting fault-tolerance for node crash is that we do not need to keep the state of the processes in a stable storage (disk copy). Instead, the state of the processes is kept in a protected part of the primary memory.

We assume that there is no malicious fault in our process control environment.

3.4 Timing Model

The *time-constraints specification* might be enforced either at the compile-time or at run-time. At the compile-time we have the advantage of not having overhead in the execution; however, the actual behaviour of the system becomes very difficult to predict. This approach is not easily applied to systems which have a great number of aperiodic tasks. On the other hand, the enforcement of the time constraints at run-time has the advantage of allowing reaction to unpredictable events; however, the price paid is a general overhead in the system.

Our approach is to deal with the real-time constraints at run-time. A real-time system is viewed as a set of processes, where each process is associated with a set of constraints. The process may contain one or more of the following constraints:

- Criticality - This can be either Hard or Soft. If omitted it is assumed to be soft.
- Start - The absolute time at which the execution should begin. If omitted it is assumed to be the earliest possible time.
- Deadline - The absolute time at which the execution should/must finish. If omitted it is assumed to be infinite.
- Duration - The time during which the process executes. If omitted it is assumed to be infinite.
- Period - The interval between successive executions of a periodic process.
- Priority - A value indicating the importance of this process.

4 Scheduling the RTR-Action

In order to make possible the scheduling of the RTR-Action, there is a need for an adaptable scheduler able to respond to sudden changes in the environment. This is due to the fact that the action is a heavier structure than the simple process. The action is performed by a set of processes, and this is why there is a need for a scheduler that is more sensitive to changes.

Existing schedulers work by making a pessimistic assumption about the conditions of the system, and they lack adaptability whilst relying heavily on an off-line knowledge of the system. The use of this kind of scheduler for the RTR-Action would render it extremely inefficient, and the guarantee of the time requirements would be very difficult to achieve.

We have designed the Milestone Least Laxity Scheduler Algorithm [16, 18, 17], which is an adaptable scheduler made possible because of the information passed by the application. A solution has been found to the main problem presented by the traditional Least Laxity Scheduler Algorithm, namely its inability to cope with a transient overload of the processor. The solution is an *alert* mechanism which is triggered according to the information available about the current execution of the processes. So, the *alert* mechanism can foresee the danger of a deadline being missed and then the scheduler can take action to increase the priority of these processes, in order to reverse this trend.

The RTR-Action is guaranteed indirectly by guaranteeing each of the processes that participate on it. However, the time constraints of the processes change during the execution of the RTR-Action, assuming an astringent value which is the composition of the several individual time constraints.

In a distributed or parallel system, in addition to scheduling the processes, we have to schedule the communication messages. If these systems are real-time then this scheduling can affect the correct functioning of the system. As our model uses a point-to-point communication network, instead of using complex protocol, we can schedule the communication messages straightaway using the laxity of the sender process. Thus, we need just to place the message in the proper queue classified by its laxity, without imposing extra

overhead. Although the laxity is calculated locally, it has a global significance because it represents the urgency of the process execution. So, when the message arrives at the receiver node, it will be dealt with according to its laxity.

5 Basic Operations to Manipulate the RTR-Action

The RTR-action mechanism provides basic operations to coordinate the concurrency and the management of the state of the participant processes of the action. A particular part of a process (or processes) can be made atomic using the operations provided. The processes participating in the RTR-Action are declared in the following way:

RTR-Action *Action_Name* **participating** *List_of_Participating_Processes*

The first process declared in *List_of_Participating_Processes* is called *coordinator* and its role is to coordinate the commitment of the RTR-Action. To define the RTR-Action we opted to use a set of independent basic operations instead of a rigid and inflexible structure. The user has the freedom to choose how to construct the RTR-Action, by defining its structure using the basic set of operations. Each of the processes in the *List_of_Participating_Processes* must use a basic operation to declare the entry point of the RTR-Action. When the block of the action has been executed, the user tests the acceptance test; depending on the result of the operation the process commits or uncommits. After the process has committed or uncommitted, it must wait for the others to do the same. An operation is needed to synchronise the commitment of the processes. This operation blocks the process until all processes have committed or uncommitted. In case an error is found, an operation is needed to recover the saved state of the processes. Also an exit operation is needed to leave the structure of the RTR-Action. And finally an operation to abort the execution of the action is provided.

5.1 Action

This basic operation allows the participant process to begin the RTR-Action. It basically saves the state of the participant process in order to be able to roll back the computation. The syntax of this basic operation is:

Action (*action_name*)

5.2 Commit

If after the execution of the block command of the RTR-Action everything is correct, then the participant process must tell this to the coordinator. This is done using the basic operation *Commit*. This announcement will set the status of the participant process as COMMITTED. The syntax of this basic operation is:

Commit (*action_name*)

5.3 Uncommit

If after the execution of the block command of the RTR-Action an error is detected, then the participant process must inform the coordinator. This is done using the basic operation *Uncommit*. This announcement sets the status of the participant process as UNCOMMITTED. The syntax of this basic operation is:

Uncommit (*action_name*)

5.4 Status

After the participant process has committed or uncommitted the RTR-Action, it needs to wait for the other participant processes to finish the execution of its basic operation. The basic operation Action_Status is the synchronisation point of the Action and will return the status of the whole RTR-Action. This status will be COMMITTED if all participant processes have committed the RTR-Action or UNCOMMITTED if not. The syntax of this basic operation is:

Action_Status (*action_name*)

5.5 Restore

If the status of the RTR-Action is UNCOMMITTED the state of the participant processes must be restored. The basic operation Restore allows backward error recovery. The state of the process can be restored to the previous state saved at the beginning of the RTR-Action.The syntax of this basic operation is:

Restore (*action_name, alternate_name*)

5.6 RestoreF

This basic operation is used in the same situation as Restore, but instead of Backward, it provides forward error recovery. The state of the process can be restored to a known safe state. This is done executing the *recovery_procedure*. The syntax of this basic operation is:

RestoreF (*action_name, recovery_procedure*)

5.7 End

If all processes commit the RTR-Action, then the action can be ended. This is done using the basic operation *End_Action* which discards the state of the process and resets all variables for the next execution of this RTR-Action. The syntax of this basic operation is:

End_Action (*action_name*)

5.8 Abort

The status of the RTR-Action can be uncommitted for the second attempt of execution or some special circumstances can be detected. In this situation, the RTR-Action can be aborted by using the basic operation *Abort_Action*. The syntax of this basic operation is:

Abort_Action (*action_name, abort_procedure*)

The RTR-Action will be stopped and the abort_procedure will be executed. If the abort_procedure is omitted then the RTR-Action will be stopped and the state restored to the saved state.

6 Time Constraints inside the RTR-Action

The real-time constraints are defined in the process and consequently are extended to the RTR-Action. Table 1 shows the real-time constraints defined in the processes and their significance in the RTR-Action. There are some constraints that have no meaning in the RTR-Action because they are imposed at process level. These are the start time, duration and period. The other three constraints criticality, deadline and priority, which do have meaning in the RTR-Action, are the composition of the constraints of the processes that participate in it. So, these constraints inside the RTR-Action are more severe. Take for example a RTR-Action with three participating processes where the priorities are 1, 5 and 7 respectively. As all the participating processes will leave the RTR-Action at the same time, processes with priority 5 and 7 may delay the leaving of the process with priority 1. So, in order to avoid this delay, during the execution of the RTR-Action, the priority of all participating processes should be equal to the priority of the process with the highest priority.

CONSTRAINT	*meaning inside the RTR-Action*
criticality	hard if at least one process is hard, otherwise soft
start time (st)	X
deadline (dl)	min dl(P1), dl(P2) ... dl(Pn)
duration (dt)	X
priority (pt)	max pt(P1) ... pt(Pn)
period (pd)	X

Table 1: Constraints inside the RTR-Action

where:
X: indicates that there is no meaning to the constraint in this level

7 Backward and Forward Error Recovery

The RTR-Action uses backward error recovery in a way similar to that of Randell's conversation [19]. The run-time environment will enforce the real-time constraints of the set of processes that participate in the structure. Each of the participant processes has a primary block which performs the normal processing and an alternate block which performs a second processing attempt. The limitation of the number of alternates is because the scheduler must guarantee the primary block and all alternate blocks. With several alternate blocks this guarantee will require an excessive spare time and make the fulfillment of the time constraints more difficult. Inside the primary and alternate blocks, there is a mechanism which provides information about the actual execution point that can be checked with the constraints of the processes.

The forward error recovery of the RTR-Action is embedded in the basic operation which restores the state of the participant processes. The state can be restored to the previous saved one or can be restored by the execution of a specified procedure. For more details see section 5.5 and section 5.6.

The general structure of the RTR-Action, using backward error recovery with the basic operations presented in the last section, is as follows:

Action (*Action_Name*)
wcet_primary: value;
wcet_alternate: value;
Begin
... primary block
if (acceptance test == TRUE) **Commit** (*Action_Name*);
else **Uncommit** (*Action_Name*);
if (**Action_Status** (*Action_Name*) != COMMITTED)
Restore (*Action_Name,alternate_name*);
else **End_Action** (*Action_Name*);
end

Alternate *Alternate_Name*
Begin
... alternate block
if (acceptance test == TRUE) **Commit** (*Action_Name*);
else **Uncommit** (*Action_Name*);
if (**Action_Status** != COMMITTED) **Abort_Action** (*Action_Name*);
else **End_Action** (*Action_Name*);
end

The general structure of the RTR-Action for forward error recovery is similar to that of the backward error recovery. However, instead of using the basic primitive **Restore**, the basic primitive **RestoreF** must be used having as parameter *recovery_name*.

The worst case execution time of the RTR-Action is calculated by adding the worst case execution time of the primary (**wcet_primary**) and the worst case execution time of the alternate block (**wcet_alternate**). In most of the executions only the primary block will execute and the execution time of the alternate will then be reused by other processes. This is done automatically by the adaptable scheduler.

8 Atomic Commitment

The processes which participate in the RTR-Action execute in a parallel system and must coordinate when meeting the real-time constraints imposed on them. Inside the RTR-Action there are two attempts of execution of the action block, one by means of the primary block and another by the alternate block. After the execution of the RTR-Action, there is a situation such that all or none of the components perform correctly. So the system will always be left in a consistent state.

In [15] we have the development of the so called *timed atomic commitment*. The timed atomic commitment is the traditional atomic action together with the enforcement of a deadline on the decision and performance of the action. Because of the fact that the deadline may expire while the action is still executing they use a more complex protocol of action commitment than the traditional one.

In contrast, our model provides the guarantee that the hard tasks will meet their deadlines. So, we have the major advantage of being able to use in the RTR-Action the same algorithm as that used in the traditional atomic commitment. As a result of this, some execution time is saved and this represents a precious commodity in real-time systems.

Each process of the set of processes which participate in the RTR-Action has two variables associated with it: *process_status* and *action_status*. The variable *process_status* represents the status of the process which can be COMMITTED or UNCOMMITTED. The variable *action_status* represents the status of the RTR-Action which can be UNDEFINED, COMMITTED or UNCOMMITTED. This variable is initialised as UNDEFINED meaning that a decision about the status of the RTR-Action has not been reached. When a decision is reached and received by this process the variable *action_status* will be COMMITTED if the RTR-Action has committed, otherwise it will be UNCOMMITTED. For the commitment of the RTR-Action we are using the traditional Two Phase Commit Protocol [10, 14]. The specification of the traditional atomic commitment is given by Hadzilacos [11]. This protocol introduces a commit coordinator which in our case corresponds to the first process participating in the RTR-Action. All participant processes in the RTR-Action have a communication path with the coordinator. The participant processes enter the RTR-Action and go into a state such that the participant processes can either redo or undo the RTR-Action. The participant processes perform the commands needed and commit or uncommit the RTR-Action according to the acceptance test.

9 Overhead of the RTR-Action

In real-time systems, the overhead inherent in the implementation of fault-tolerant mechanisms apparently conflicts with the need to fulfill the time constraints. In order to validate our mechanism we need to show that the mentioned conflict was solved. It means that the overhead imposed by the mechanism is an acceptable value within the real-time environment. We carried out a simulation of the adaptable scheduler to better understand its behaviour, and we could do the same with the whole RTR-Action, but we certainly would not touch on the central point of the mechanism. So, the only option left was to implement the real-time recoverable action in a hardware parallel platform. To build the whole operating system was not viable due to the time available. The final solution was to utilise an industrial real-time kernel and add to it the whole RTR-Action and its structure.

We have implemented the RTR-Action with the Milestone Least Laxity Scheduler in the Real-Time Kernel RTXC/MP [1]. We are running the RTXC/MP kernel in a motherboard with Transputers (T800 - 20 MHz).

The amount of time used by the kernel to execute the kernel calls of the RTR-Action was measured in our implementation. The results can be seen in Table 2. These measure-

[1] RTXC/MP is a trademark of Intelligent Systems International.

ments were carried out using the timer calls of the kernel which have accuracy on the order of micro-seconds. More precisely, they were carried out by measuring the time before and after the execution of the kernel call. The results shown in the table correspond to the average of these differences.

PRIMITIVE	*TIME (microseconds)*
Action	128
Restore	233
RestoreF	64
Commit	64
Uncommit	64
Action_status	64
End_Action	64
Abort_Action	64

Table 2: Overhead of the RTR-Action's Kernel Call in a Single Processor

We can see that for a single processor the overhead imposed by the RTR-action is quite reasonable. In the case where the action commits, the result is a total overhead of 256 micro-seconds (Action, Commit, End_Action). In the case where the action uncommits the results is a total overhead of 553 micro-seconds (Action, Uncommit, Restore, Commit, End_Action).

The overhead of the kernel call Restore depends on the size of the stack of the process. Table 2 was built using a stack of 128 bytes. In order to better understand the effect of different stacks on the overhead, we built Table 3.

STACK	*TIME (microseconds)*
1 Kbytes	324
512 bytes	275
256 bytes	246
128 bytes	233

Table 3: Overhead of the Kernel Call Restore with Differents Stacks

9.1 Overhead for More than One Processor

Table 4 shows the overhead of the action for the situation where the participant processes are being executed in more than one processor. The overhead of the kernel calls Action, Restore, Commit, Uncommit, End_Action and Abort_Action remains the same. The execution of the commit protocol involves a communication delay. So the Action_status kernel call has this additional overhead. This overhead is imposed by each participant process that is running in a different processor.

In our computational model there are several communication links and this brings the possibility of parallel communication. In order to our system achieve a better performance its configuration needs to be in such a way that the commitment protocol can be executed in parallel. This is to say that each participant process should have an exclusive link that would allow it to communicate with the coordinator processor.

PRIMITIVE	*TIME (microseconds)*
Remote Commit	344
Remote Uncommit	344
Remote Restore	577
Remote RestoreF	408

Table 4: Overhead for the remote commit/uncommit

10 Conclusion

The fulfillment of the real-time requirements and fault-tolerance in real-time systems have been addressed without regard for the integration of these conflicting aspects. The fault-tolerance aspect can interfere in the fulfillment of the real-time requirements and vice versa. Hence, it is very difficult to tackle efficiently just one of these aspects; what is needed is a uniform, high level structure which is able to address the real-time and fault-tolerance requirements together during the design of the system.

We designed a programming structure *Real-Time Recoverable Action* (RTR-Action) which provides fault-tolerance for real-time systems. The objective was to create a high-level abstract structure, making it easier to enforce the real-time constraints and error recovery. The atomic action has traditionally been used in the design of fault-tolerant systems. We extended the concept of atomic action to be used in hard real-time systems, thus making possible the utilisation of the mechanism to achieve recovery from violations of the time constraints. An adaptable scheduler was used to help in the detection of the timing errors, as well as in the scheduling of the whole set of processes including the ones that participate in the RTR-Action.

We have designed the Milestone Least Laxity Scheduler Algorithm, which is an adaptable scheduler made possible because the information passed by the application. The use of schedulers currently available to schedule the RTR-Action could make it very inefficient and the guarantee of the time requirements would be very difficult to achieve. Our adaptable scheduler is able to cope with a transient over load of the processor. The guarantee of the hard tasks is achieved by using an *alert* mechanism, which can foresee the danger of a deadline being missed and then react by increasing the priority of these processes.

We are using the same adaptable scheduler to schedule the processes as well as the messages exchanged through the point-to-point communication network. This approach, which uses the integrated scheduler, proved to be very effective in allowing us to deliver the RTR-Action with a low overhead.

The great advantage of our model is that, as we can guarantee the hard tasks, we do not have the possibility of a deadline being missed during the execution of the RTR-Action. In contrast with other *timed atomic commitments* we can use the traditional atomic commitment instead of a more complex one.

The crucial question of the overhead was kept at an acceptable level. It is not easy to determine an ideal value which is convenient for everybody and can be used for all applications; it is application dependent. There is little discussion about overhead in the literature to be compared.

We can mention the overhead of a real-time action [7]. It does not have support to recovery from fault, but it uses a reserve of resources. A limitation in this implementation is the connection of the network by an Ethernet. The overhead of this action is around 70 milliseconds.

Another example of overhead that we can mention is the CHAOS [9] abstract objects. There are the ObjFastInvoke which involves no transfer of data and the ObjInvoke which involves the transfer of control or data information. The overheads of the local sharable object are: for ObjFastInvoke 1.1 milliseconds and ObjInvoke 6.0 milliseconds. The overheads of the remote sharable object are: for ObjFastInvoke 1.1 milliseconds and ObjInvoke 6.4 milliseconds.

Another way of looking at this question is by comparing the overhead obtained for the whole RTR-Action with the overhead of a kernel call of the RTXC/MP. We can use a kernel call wait/signal handshake between two tasks which spends 159 microseconds. The overhead of the RTR-Action in the centralised case is around 0.5 milliseconds, which corresponds to approximately three mentioned kernel calls. In the parallel case, the overhead is around 1.5 milliseconds, which corresponds to approximately nine mentioned kernel calls.

References

[1] Felicita Di Giandomenico A. Bondavalli and Jie Xu. Cost-effective and flexible scheme for software fault tolerance. *Computer Systems Science and Engineering*, 18(4):234–244, October 1993.

[2] J. Stankovic A. Bondavalli and L. Strigini. Adaptable fault tolerance for real-time systems. *ESPRIT BRA 6362 Predictably Dependable Computing Systems 2*, pages 117–145, September 1993.

[3] J. McDermid A. Burns and A. Wellings. Tolerating hardware and software fault in hard real-time systems: An integrated approach. *Technical Report, Department of Computer Science, University of York*, 1993.

[4] T. Anderson and J.C. Knight. A framework for software fault tolerance in real-time systems. *IEEE Transactions on Software Engineering*, 9(3):355–364, May 1983.

[5] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[6] A. Burns and A. J. Wellings. Criticality and utility in the next generation. *The Journal of Real-Time Systems*, 3(4):351–354, 1991.

[7] V. F. Wolfe S. B. Davidson and I. Lee. Rtc: Language support for real-time concurrency. *Real-Time Systems*, 5(1):63–87, 1993.

[8] R.L. Glass. Real-time: The lost world of software debugging and testing. *Communications of the ACM*, 23(5):264–271, May 1980.

[9] K. Schwan P. Gopinath and W. Bo. Chaos - kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, 1987.

[10] J.N. Gray. Notes on data base operating systems. *Lectures Notes in Computer Science*, 60:393–481, 1978.

[11] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. *Fault-Tolerant Distributed Computing, B. Simons and A. Z. Spector (Eds.), Lectures Notes in Computer Science*, 448:201–208, 1990.

[12] H. Hecht. Fault-tolerant software for real-time applications. *Computing Surveys*, 8(4):391–407, December 1976.

[13] R.H. Campbell K.H. Horton and G.G. Belford. Simulations of a fault-tolerant deadline mechanism. *Ninth Annual Symposium on Fault-Tolerant Computing - FTSC9*, pages 95–101, 1979.

[14] B. Lampson. Atomic transactions. *Distributed Systems Architecture and Implementation: An Advanced Course, Lectures Notes in Computer Science*, 105:246–265, 1981.

[15] S.B. Davidson I. Lee and V. Wolfe. Timed atomic commitment. *IEEE transactions on Computers*, pages 573–583, May 1991.

[16] C.E. Moron and H. Zedan. Towards an adaptable scheduler for real-time systems. *15th Technical Meeting of the World occam and Transputer User Group*, pages 154–166, April 1992.

[17] C.E. Moron and H. Zedan. Adaptable scheduler using milestones for real-time systems. *Technical Report - YCS 191, Department of Computer Science, University of York*, January 1993.

[18] C.E. Moron and H. Zedan. On guaranteeing hard real-time tasks. *Nineteenth EUROMICRO Conference*, pages 485–490, September 1993.

[19] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[20] J. A. Stankovic and K. Ramamritham. What is predictability for real-time systems. *Real-Time Systems*, 2(4):247–254, 1990.

[21] H. O. Welch. Distributed recovery block performance in a real-time control loop. *Proceedings of the IEEE Real-Time Systems Symposium*, pages 268–276, 1983.