A new Checkpoint Mechanism for Real Time Operating Systems

Santiago Rodríguez, Antonio Pérez, Rafael Méndez
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Universidad Politécnica de Madrid, Spain
Email: {srodri, aperez, rmendez}@fi.upm.es

Abstract

This paper presents an overview of a proposed protocol to provide application–transparent fault tolerant services in a Real Time Operating system. Fault tolerance is achieved by saving checkpoints of the processes belonging to a real time application. This approach proposes the extension of some real time system calls in order to save a recovery point when the user invokes them. This protocol allows a real time application designer to know the temporal specifications of every system call. Current real time applications are composed of several Real Time processes and they have to share data by using interprocess communication facilities provided by the operating system. The operating system has to take into account these interactions to ensure the consistency of checkpoints. This is done by tracking the communications performed since the last checkpoint and forcing dependent processes to perform a checkpoint at the same time.

1 Introduction

In order to improve computer reliability, several checkpointing based fault tolerant multiprocessors have been proposed in recent years. These approaches deal with transient and permanent errors for general purpose computing. Real Time environments put additional requirements ([10]) because they are specified by adding deadlines to every task in the system. So, the computational correctness of the system depends on both the logical results and the ability to meet the specified deadlines.

Some current checkpoint techniques take advantage of the memory hierarchy ([2]) to store active data (data accessed after the checkpoint) in the highest levels of the memory hierarchy and checkpoints in the lowest ones. In Cache Aided Rollback Error Recovery ([1]) active data are present in the computer registers and cache and checkpoints are saved in main memory. When a cache miss appears on a processor and a line has to be replaced, a checkpoint is done by storing the modified cache lines into main memory.

Dual schemes use different mapping for the device containing the active data and the recovery data (memory based schemes). These ones use different physical memory locations to store the active data and the checkpoints. These schemes are more useful for real time systems because either the application or the real time operating system can decide when to perform the checkpoint.

2 Objectives and Architecture

Our objective is to design an application transparent checkpoint protocol that will be performed by the real time operating system. Its main characteristics are:

- **Real Time Support.** The system has to support real time applications composed for several tasks (Unix processes) executing concurrently. Several of these tasks perform communications among them and data consistency ([3, 6]) stored in the checkpoint has to be ensured. A checkpoint established by a task forces its dependent tasks (they have communicated with the former one) to establish a checkpoint.

- **Fault masking.** The system has to be able to mask a failure and it has to continue its execution without intervention of the real time application.

- **Portability and transparency.** In order to obtain application transparency, the protocol is included in the operating system level. So, the execution of the recovery actions (checkpoint and rollback) will be performed by the operating system. Portability is achieved by using the Posix.1b standard ([5]) to include the fault tolerant services.

- A **Stable device** will be used to store the recovery data (checkpoint). This device ensures that data stored on it will not be corrupted by faults or external actions.

3 Protocol Analysis

As stated in previous sections, predictability is only achieved by controlling the instant when a checkpoint is done. Furthermore, portability is not achieved if the application has to perform

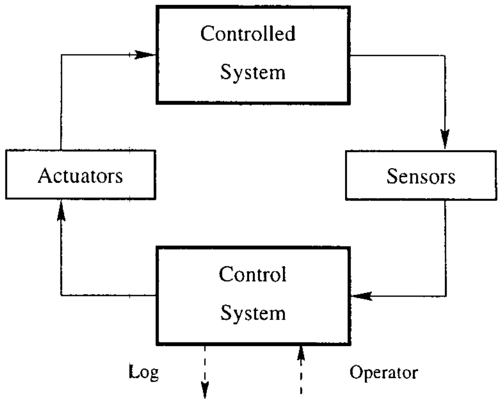

Figure 1: Physical Real Time Application Structure.

special system calls to establish a checkpoint or to perform a rollback. In our approach the operating system has to decide when a checkpoint has to be saved. Figure 1 shows the typical structure of a real time application. A real time application is closely related with its environment: it reads data from a controlled environment and modifies some parameters in the controlled system. The system is usually composed of several tasks executing in fixed intervals (periodic tasks). They read data from the sensors and write in the actuators. Aperiodic tasks ([9]), activated when an event appears, perform some write operations in the actuators.

As established in the previous section, the fault tolerant system is built by including new facilities in a Posix.1b extension to a Unix system, so data acquisition from the system and the environment modification has to be done by invoking system calls (*read* and *write*).

In our solution, additional code has been included in some selected system calls to save the state of a Unix process. These modifications are performed inside the operating system. Predictability is ensured because the temporal specification of the modified system calls will include the storing of the checkpoints.

Table 1 shows the recovery actions performed when a system call is invoked. If check-

System Call	**Operation**
exit	*checkpoint* deleting
fork	*checkpoint* for both processes
exec	*checkpoint* deleting
write	*checkpoint*
open	*checkpoint*
creat	*checkpoint*
link	*checkpoint*
unlink	*checkpoint*
mknod	*checkpoint*
fchmod	*checkpoint*
chmod	*checkpoint*
fchown	*checkpoint*
chown	*checkpoint*
ioctl	*checkpoint*
fcntl	*checkpoint*
mq_open	*checkpoint*
mq_close	*checkpoint*
mq_unlink	*checkpoint*
sem_open	*checkpoint*
sem_close	*checkpoint*
sem_unlink	*checkpoint*
shm_open	*checkpoint*
shm_close	*checkpoint*
shm_unlink	*checkpoint*
aio_write	*checkpoint*
lio_listio	*checkpoint*

Table 1: System Calls and Recovery Actions.

pointing is frequently done, computer performance will be decreased because a great amount of temporal data will be stored. We try to reduce the performance loss in two ways: checkpoints are saved only at the end of a control cycle (the number of checkpoints is reduced) and only when a write is done (the number of system calls affected are decreased).

Checkpoint deletion is done when a Unix process changes the execution image or finishes its execution. So, when a Unix process performs an *exit* system call, it will be removed from the system process table, its assigned resources are freed and the recovery information has to be removed from the stable device. Some other system calls do not remove a process from the system, but change the execution image of a process (*exec* system calls). This case forces to remove the information stored in the stable device because the process that invoked this system call will change its execution image (text segment) and the data segment will be initialized with new data. On the other hand, the stack segment will contain the new stack (one subroutine activation) with the program entry point. So, the checkpoint stored for this process has to be deleted because the virtual addresses of the process after executing the *exec* system call will have no relation with the data stored at these addresses before executing the system call.

Another system call related to process creation is the *fork* system call. This one creates a new process in a Unix system and a recovery point will be saved for both the creator process (parent) and the new one. If a fault is detected after finishing this system call the system will need the first recovery point to rollback.

As stated above, the system will save a checkpoint when it modifies some variables in the controlled system. In a Unix system this is done by executing *write, aio_write* and *lio_listio* system calls. Other system calls related with the I/O system such as *open, creat, link, etc.* may modify the file system and a checkpoint is done at the end of these services. System calls related with message queues do not save a checkpoint, except the *mq_open* and *mq_unlink* calls. These system calls modify the file system and a checkpoint is done when the service is finished. The same actions are performed for the equivalent calls related to semaphores (*sem_open* and *sem_unlink*) and shared memory segments (*shm_open* and *shm_unlink*). *close*, *mq_close*, *sem_close* and *shm_close* also perform a checkpointing because they are freeing a resource that is being used to track dependencies (see section 4). When a checkpoint is done for one process some others have to perform a checkpoint to maintain data consistency. The dependency study is done in section 4.

3.1 Checkpoint Information

A checkpoint of a process has to contain the information that allows the operating system to be able to recover a consistent execution state. The information that has to be saved in a checkpoint is:

- User pages modified since the last checkpoint. Every page that has been modified since the last checkpoint is stored in the stable device. So, an additional bit has to be added for every entry in the page table to keep track of every page modified since the last checkpoint.
- Process table entry of the process. Some additional information related to the process is stored in the process table, i.e. opened files, opened message queues, scheduling state, etc. that are included in the checkpoint.
- Internal operating system structures related to the process that is being saved on the stable device. Because a process can communicate with other ones the structure related with message queues, semaphores and shared memory has to be stored into the stable device.

4 Dependency Tracking

A Real Time application is composed of several tasks that communicate among them. A solution to maintain consistency is shown in [7]. It allows to establish checkpoints in an application transparent way. This solution performs a global checkpoint and the operating system does not track dependencies. If a process performs a checkpoint, the operating system forces every process in the system to establish a recovery point, even when there has not been communication among processes since its last checkpoint. This approach is simple but introduces additional delays that may involve deadline misses.

Another solution to avoid the dependency tracking is to establish a checkpoint every time a process communicates with another one. This solution introduces high overhead in the checkpointing of a process because it saves a great amount of intermediate results.

Our solution tries to take into account the communications established among Unix processes belonging to one application and to maintain the recovery data consistency by checkpointing the dependent processes. These dependencies will be generated when a process uses some services provided by Posix.1b to communicate processes: message queues, Posix.1b semaphores, Shared Memory objects, Pipes and FIFO's and Signals. System calls related with this kind of interprocess communication need to perform several actions to track the dependencies. The complete study is done in [8] and as an example we will show the dependency tracking for message queues (figure 2).

4.1 Message Queues Dependencies

P and P' are two processes performing *mq_send*, *mq_receive* and *mq_setattr* system calls. Let's assume that at t_1 a recovery point is saved for P and at t_2 and t_3 a checkpoint is established for P' (see figure 2.a). When a checkpoint is done the private pages of the process and the complete message queue state is saved in the stable device. After P saves a checkpoint in t_1 it performs a *send* or *receive* operation in the queue (S/R), so the message queue state changes from the state in t_1 (E) to state E'. Assume that a fault is detected in t_{RB} and a rollback is needed for process P.

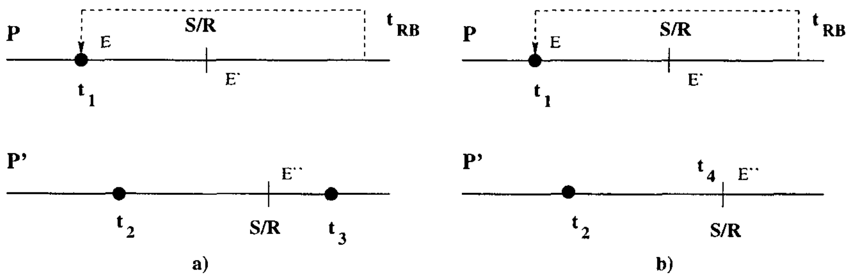

Figure 2: Message queues dependencies.

Then, the state of process P is restored to the state in t_1, but the message queue state is restored to the last saved state in t_3 (E''). If the operation performed in S/R is a *mq_send* system call, the result of the inconsistent rollback is that P sends the same message twice to the queue. If the operation performed in S/R was a *mq_receive* call P will try to receive a message that will never be sent. To solve this inconsistent rollback, it is necessary to force P to establish a recovery point when P' does it.

Figure 2.b shows a different situation, because the second checkpoint is not done for P' in t_3. At t_{RB} a rollback is needed for process P, then private data of P are restored to t_1 and the state of the message queue is restored at the state saved in t_2 and P' execution goes on. This rollback is not consistent because P will again execute an operation on the queue (S/R) and if P performs a *send* call this message will never be received, or if P performs a *receive* call P will never receive a message that has been sent.

The conclusion is that a process that has executed a *mq_send*, *mq_receive* or *mq_setattr* operation on a message queue after establishing a recovery point is dependent to every process that has performed one of these primitives. To keep track of the dependencies a message queue modified flag is added per message queue opened for a process. When a process modifies a message queue (*mq_send*, *mq_receive* or *mq_setattr* operation) this flag is set and when a checkpoint has to be done for a process, the operating system has to look for every process that has modified the same message queue. When the checkpoint has finished every message queue descriptor is cleared.

5 Evaluation Results

In order to validate our approach we have designed models that allow to compare the behaviour of our proposal with another current policy (global checkpoint).

- **Global Checkpoint.** There is no dependency tracking among processes. If a process is going to establish a checkpoint every process in the system establishes a checkpoint and no dependencies are generated.

- **Proposed Model.** As stated in section 4 dependencies among processes are tracked to reduce the number of checkpoints that have to be saved and the deadlines that are not met in the real time application. So, when a checkpoint is established for a process in the system, only its dependent processes are forced to establish a checkpoint.

A comparative study has been done for both models building two simulation programs based on queue networks for each of them. Both models execute a real time application composed by several processes. Two parameters are specified for every process in both

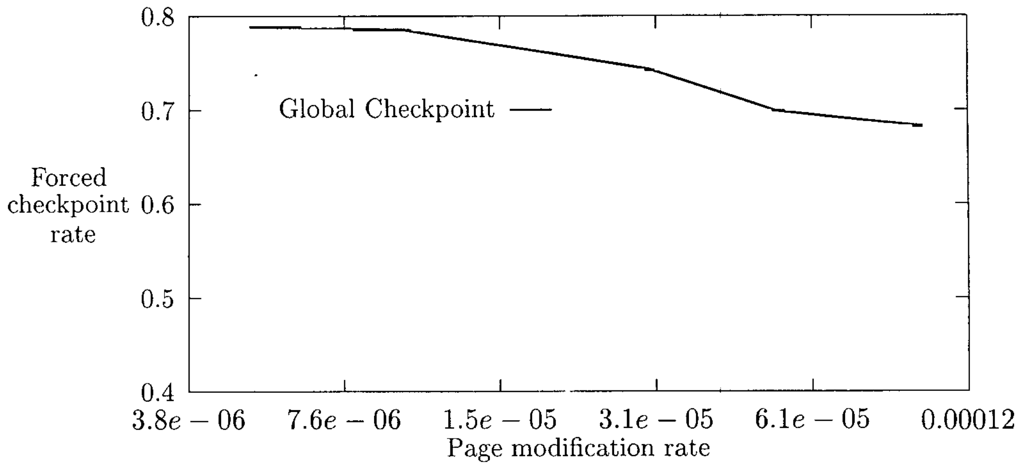

Figure 3: Forced checkpoints rate in the global checkpoint model.

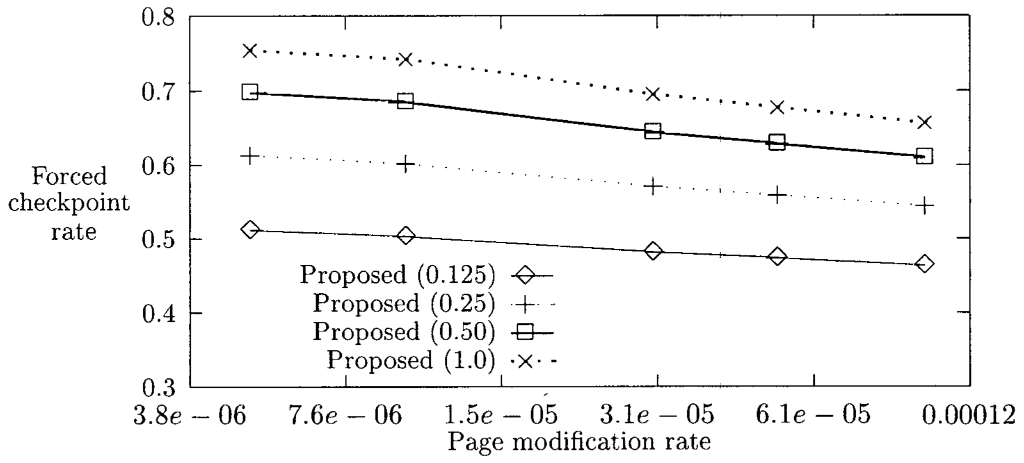

Figure 4: Forced checkpoints rate in the proposed model.

models: activation period and computation time for each activation of it. For our proposed model a third parameter is included: a rate in the range [0-1] that specify the probability of establishing an interaction with another process of the system.

The main aspect that will affect the performance of the model is the number of checkpoints that are forced to be done by a Unix process because another one has to perform a checkpoint. Figures 3 and 4 show the rate of forced checkpoints in the global checkpoint model and in the proposed model respectively. Our model shows that even when the communication rate is the highest (1.0) the behaviour is slightly better than with the global model. Note that communication 1.0 means that every process in the application communicates every period with every process in the application. This is a completely dependent application and it can not be found in actual systems. For the usual communication rates (from 10% to 25%) the proposed model is quite better that the global one. Our approach reduces in a 30% the forced checkpoints that generates the global checkpoint model.

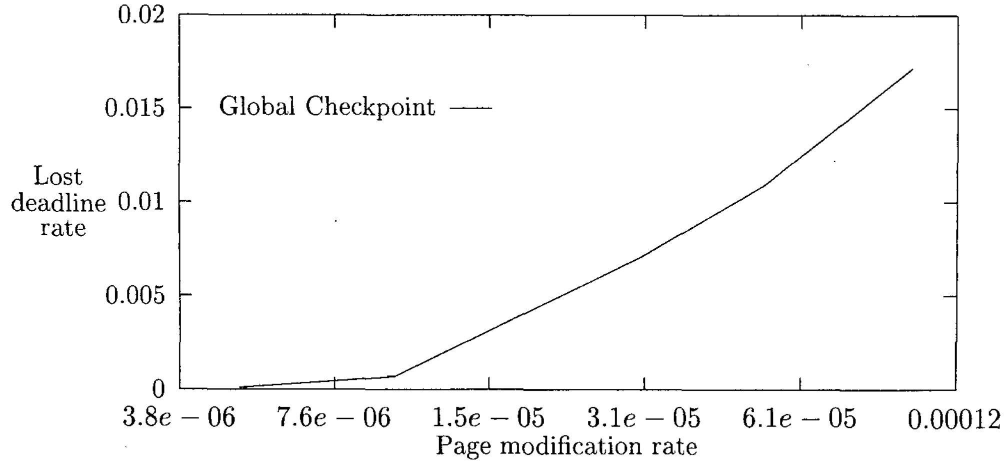

Figure 5: Lost deadlines in the global checkpoint model.

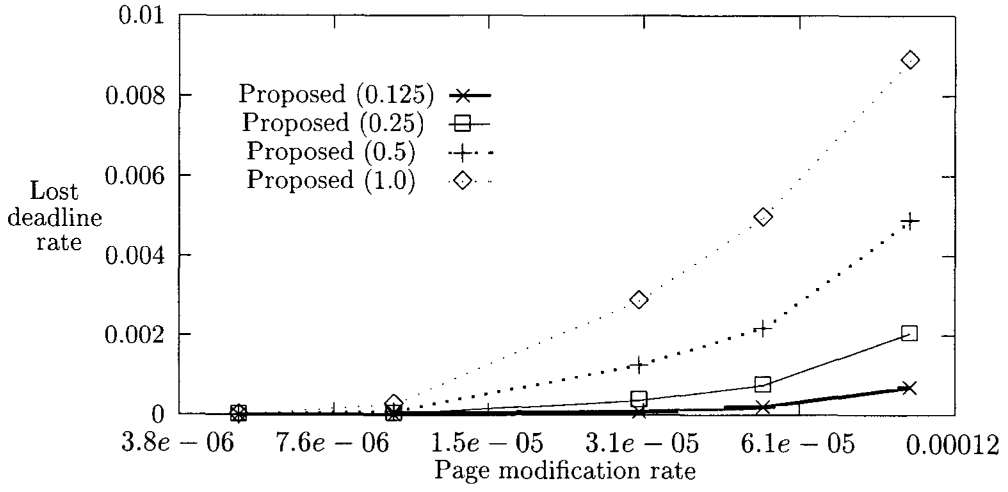

Figure 6: Lost deadlines in the proposed model.

One of the main objectives of a real time application is to meet the deadlines. Figures 5 and 6 show the lost deadlines rate for the global and the new proposed protocol (for communication rates of 0.12, 0.25, 0.5 and 1.0). Some conclusions can be deduced from both figures:

- The behaviour of the proposed model is better even when the communication rate is 1.0.

- The proposed model does not introduce any lost deadline when the communication rate is 0.0 (completely independent process). This rate is not present in figure 6 because actual systems has interactions among processes of the same application.

- Communication rates used in an actual application goes from 10% to 25 % ([4]). In these cases the new proposed model lose only a 10% of the deadlines lost in the global checkpoint model.

6 Conclusions

This paper has presented a protocol for establishing recovery points in a real time environment. It allows the real time application designer to know the time used for executing a system call including the time spent in saving recovery points on a stable device. This mechanism tracks dependencies and only the necessary information is saved to optimize the extra delay and the use of the stable device.

The performance studies are being done by performing some simulations that compare our model with the global checkpoint one. Some results have showed that our model has better performance metrics than global checkpointing:

- The number of forced checkpoints is highly reduced comparing with the global model. So, the busy percentage of the stable device is reduced in the same rate for the proposed checkpoint model.
- The number of deadlines not met is reduced in a 90 %.
- Even for extremely high communication rates the new proposed model behaviour is better than the global checkpoint model.

References

[1] R. E. Ahmed, R. Frazier, and P. Marinos. Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessors systems. In *FTCS-20*, pages 82–88. IEEE, June 1990.

[2] N. S. Bowen and D. K. Pradhan. Processor – and memory – based checkpoint and rollback recovery. *IEEE Computer*, 26(2):22–31, February 1993.

[3] L. M. Censier and P. Feautier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[4] K. Fowler. Inertial navigation system simulation program: Top level design. Technical Report CMU-SEI-89-TR-38, Carnegie Mellon University, Pittsburgh, PA 15238, USA, January 1990.

[5] IEEE. *IEEE Standard for Information Technology: Portable Operating Systems Interface (POSIX 1.b)*. IEEE, 1994.

[6] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, Wien, second edition, 1990.

[7] K. Li, J. F. Naughton, and J. S. Plank. Real-time concurrent checkpoint for parallel programs. *ACM SIGPLAN Notices*, 25(3):79–88, March 1990.

[8] S. Rodríguez. *Sistema Operativo de Tiempo Real con Tolerancia a Fallos mediante Puntos de Recuperación*. PhD thesis, Dpto. de Arquitectura y Tecnología de Sistemas Informáticos. Universidad Politécnica de Madrid, July 1996.

[9] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

[10] J.A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, October 1988.