ISSN:0975-9646

REAL-TIME FAULT TOLERANCE AND IT'S IMPLEMENTATION USING RTAI 3.8

Ashis Kumar Mishra, Anil Kumar Mishra, Yogomaya Mohapatra
Department of Computer Science & Engineering, College of Engineering & Technology
Bhubaneswar, Pin-751003, Odisha, India

***Abstract*— the fault tolerance for real-time application avoidable by help of the protocol is CRR. However, existing checkpoint implementation support only non-real-time applications as the checkpoint overhead is usually non deterministic. In this paper, we represent an implementation of checkpoint scheme with the RTAI 3.8 supported by Linux, where services provided by the RTOS makes the checkpoint overhead, including the time to place recover the system from a failure is predictable. It also gives an optional view of the hard real-time definition and an appropriate perspective on why low cost general purpose computer can be an effective operating system available. The RTAI 3.8 for Linux describe here is a viable and effective open-free source software approach for adding hard real time capabilities to a widely available general purpose operating system. It keeps real time applications separated from non real time ones, achieving high efficiencies for both kinds of executions by affording appropriate synchronization and communication tools to allow an efficient interaction between two environments.**

***Keywords*— CRR, RTOS, API, FAULT-TOLERANCE, HARD REAL-TIME.**

1. INTRODUCTION

The study of temporal aspects of computations has been ongoing for several decades. However, in recent years the attention being paid to this subject has increased tremendously. A wide variety of system currently exists in which timeliness is an important requirement, and these are called real-time systems. More precisely, real-time systems are systems where the correctness of operation depends not only logical result of their computations, but also on the time at which the results are produced [8]. Consider a system in which data need o be processed at a regular and timely rate. For example, an aircraft uses a sequence of accelerometer pulses to determine its position. In addition, a system other than aeronautic ones requires a rapid response to events that occur at a no regular rates, such as an over temperature failure in the nuclear plant [1]. In some sense it is understood that these events require Real-Time processing.

Now consider a situation in which passenger approaches an airline reservation counter to pickup his ticket for a certain flight from New Delhi to Mumbai, which leaving in 5 minutes. The reservation clerk enters the appropriate information in to the computer and a few second later a boarding pass is generated. Is this Real-Time System? Indeed, all three systems- aircraft, nuclear plant, and airline reservation are Real-Time because they must process information within a specified interval or risk system failure. Although these examples provide an intuitive definition of real-Time system, it is necessary to clear define when a system is Real-Time and when it is not. The fundamental definition of Real-Time system engineering can vary depending on the resources consulted. The following definition have been collected and refined to form that is intended to be most useful to the practicing engineering, as opposed to the theorist. Real-Time systems are different from general purpose computing systems in several ways. The processes in a real-time system have time related attribute such as ready times, deadlines, computation times and periods [10]. Therefore, the worst case behaviour of real-time systems is more important than the average response time and user conveniences, which are important issues in general purpose computing systems.

The hardware of the general purpose computer solves problems by repeated execution of macro instruction collectively known as software. Software is traditionally divided in two types' system program and application program. A system program consists of software that interfaces with the underlying computer hardware, such as scheduler, device drivers, dispatcher, and program that acts as tools for the development of the application programs [9]. These tools include compilers, which translate high order language into a special binary format called object or machine code and linker, which prepare the object code for execution. An operating system is a specialized collection of system programs that manage the physical resources of the computer s. As such, a Real-Time operating system is a system program. An application programs written to solve specific problems. Such as a pay-roll preparation, inventory, and a navigation. Certain design considerations pay a role in the design of certain system programs and application software intended to run in Real-Time environment.

The notation of a system is a central to software engineering, and indeed to all engineering and warrant formalization. A system is a mapping of set of inputs in to a set of output. When the internal details of the system are not of interest, the mapping function can be considered as a black box with one or more inputs entering and one or more output exiting from the system. Every Real-world entity, whether synthetic or occurring naturally, can be modelled as a system. In computing system, the inputs represent digital data from hardware devices and other software system. The input are often associated with sensors, cameras, and other devices that provide analog inputs, which are converted to digital data, or provide direct digital input. The digital output of the computer system can converted to analog outputs to control external hardware devices such as actuators and displays. The time between the presentation of a set of inputs to a system (stimulus) and the realization of the required behaviour (response) including availability of all associated output, is called response time of the system.

A Real-Time system is a system that must satisfy explicit (bounded) response time constraints or risk severe

consequence including failure. A real-Time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness. In any case, note that by making unnecessary the notation of timeliness, every system. Real-Time systems are often reactive or embedded systems. Reactive system are those in which scheduling is driven by ongoing interaction with their environment. For example, a fire control system reacts to buttons pressed by a user. These systems respond to a series of external inputs, which arrive in an unpredictable fashion. The Real-Time system process these inputs, take appropriate decisions and also generate output necessary to control the peripherals connected to them. The design of a Real-Time system must specify the timing requirements of the system and ensures that the system performance is both correct and timely. There are three types of time Constraints :(1) **Hard**: A late response is incorrect and implies a system failure. (2) **Soft**: Timeliness requirements are defined by using an average response time. If a single computation is late, it is not usually significant, although repeated late computation can result in system failures.(3) **Firm**: Firm real-time systems have hard deadlines, but where a certain low probability of missing a deadline can be tolerated.

Scheduling and resources allocation in Real-Time systems are difficult problems due to the timing constraints of the task involved. The order in which the tasks are scheduled or dispatched has a large effect on the chances of the tasks meeting their timing constraints. Many of the Real-Time scheduling problems are known to be NP complete. A great deal of research has been conducted for scheduling in a variety of Real-Time system models.

2. LITERATURE SURVEY

2.1 Fault-Tolerance

Due to the catastrophic consequences of violating timing constraints in hard Real-Time systems, it is important to consider the effects of the operating environment on the system. Since the environment may cause various kinds of faults to be generated, it is essential that fault tolerance be incorporated in a real-time system when it is designed. A system is fault-tolerant if it continues to perform its specified tasks in the presence of hardware failures or software errors [8]. A fault-Tolerant system has to ensure that faults in the system (which are defects in hardware or software) do not lead to a failure (which is the non-performance of some action that is due or expected). Fault-Tolerance is achieved through the use of redundancy, which is the addition of information, resources, or time beyond what is needed for normal system operation [8]. One of the main requirements of a fault-tolerant system is reliability. A highly reliable system continues to perform correctly over long interval of time. For many reasons, the fields of real-time systems and fault tolerance have largely evolved independently of each other. One of the reasons is that more important aspects of Real-Time system such as scheduling were not well understood till recently. Another reason is that massive hardware redundancy was used as the main technique for tolerating faults. This technique is expensive and could be afforded only in large system such as space shuttles and aircrafts.

The fault tolerance requirements make a Real-Time system even more complicated because faults must be detected and tolerated within the timing constraints of the tasks. If faults trigger backup tasks for recovery purposes, the backup tasks must also be executed before the task deadlines. Due to these complexities, most Real-Time systems to date only dealt with timing constraints.

Transient faults in Real-Time systems are generally tolerated using time redundancy, which involves the retry or re-execution of any task running during the occurrence of transient faults [1]. Several studies have done for using time redundancy in embedded Real-Time system for tolerating faults. Pandya and Malek in have used time redundancy for tolerating single fault. In the event of faults, all unfinished tasks are re-executed. Authors have presented static and dynamic allocation strategies to provide fault-tolerance. Two algorithms have proposed to reserve time for the recovery of periodic Real-Time tasks on a uniprocessor. Authors have provided exact schedulability tests for fault-tolerant task sets. In their paper, time redundancy has been employed to provide a predictable performance in the presence of failures. However no study has been done about adding appropriate and efficient time redundancy into the schedule, which is the main contribution of this paper. In recent years, Rate-Monotonic (RM) scheduling policy has been used to schedule Real-Time tasks in a variety of critical applications. However, RM does not provide mechanism for managing time redundancy, so that Real-Time tasks will complete within their dead-lines even in the presence of faults. The goal of this paper is to add appropriate and efficient time redundancy to the RM scheduling policy for schedule periodic tasks.

2.2 Causes of Hardware Transient Faults

- Limitations in the accuracy of electromechanical devices (such as the positioning servomechanism for the reading heads of a disk drive).
- Electromagnetic radiation received by interconnections (such as long buses acting like receiving antennas).
- Power fluctuations or glitches not properly filtered by the power supply.
- Effects of ionizing radiation on semiconductor devices. This last cause is currently the most important challenge to device designers and requires some more explanation. It has been only recently that the effects of ionizing radiation have been recognized as a source for "soft" faults in computer memories. "Soft" means that the information held in a memory device has changed, but no irreversible change in the device has occurred. Information in computer memories is stored as the presence or absence of charge in capacitors. When an energetic particle creates electron- whole pairs in the vicinity of a capacitor, some of the added charge carriers are collected by the capacitor. If the added charge is sufficiently large, the information stored is changed. The amount of charge that represents a bit and the "critical" charge that is needed to change it has decreased with miniaturization and the advent of VLSI technology. Transient failures in semiconductor memories due to ionizing radiation were not significant until the introduction of 16K bit

and 64K bit memory chips. Two main causes have been detected as far as sources of ionizing radiation which affect the operation of digital computers.

- Trace amounts of natural radioactive elements in metallic and ceramic packaging materials.
- The effects of cosmic rays. Although the effect of radioactive materials in packaging materials can be reduced by further purification and better system design, it is not clear how the effects of cosmic rays can be avoided.

Fault-Tolerance is the tendency to function in the presence of hardware or software failure. In Real-Time system, Fault-Tolerance includes design choices that transform hard Real-Time into soft ones. There are often encountered in interrupt driven system, which are can provide for detecting and reacting to a missed deadlines. Fault-Tolerance designed to increase reliability in embedded system can be classified as either **Spatia**l or **Temporal**.

2.3 Spatial Fault-Tolerance

The reliability of most hardware can be increased using redundant hardware. In one common scheme, two or more pair of redundant hardware devices provided inputs to the systems [9]. Each device compares its output to its companion. If the results are unequal, the pairs declare itself in error and the output ignored. An alternative is use a third device to determine which of the other two is correct. In either case, the penalty is increased cost, space, and power requirement. Voting scheme can also be used in software to increase the algorithm robustness [1]. Often like inputs are processed from more than one source and reduced to some sort of the best estimate of the actual value. For example, an aircraft position can be determined via information from satellite positioning systems, inertial navigation data, and ground information. A composite of these reading is made using simple averaging on a kalman filter.

2.4 Temporal Fault-Tolerance

Involves techniques that allow for tolerating missed deadlines. Of two temporal Fault-Tolerant is more difficult to achieve because it requires careful algorithm design [1].

2.4.1 Checkpoints

One way to increase Fault-Tolerant is to use checkpoints. In this scheme, intermediate results are written to memory at fixed location code for diagnostic purposes. These location are called check points, can be used during system operation and during verification. If the checkpoints are used only during testing, then the code is known as test **Probe** [5]. Test probes can introduce subtle timing errors.

2.5 Recovery-Block Approach

Fault-Tolerance can be further increased by using checkpoint in conjunction with predetermined reset points in software [7]. These reset points mark recovery block, the check point are tested for "reasonableness". If the results are not reasonable, then processing resumes with prior recovery block. The points, of course, are that some hardware devices (or another process that is independent of the one in question) have provided faulty inputs to the blocks. By repeating the processing in the block, with presumably valid data, the error will not be repeated. In the process block model, each recovery block represents a redundant parallel process to the block being tested. Although this strategy increases system reliability, it can have severe impact on performance because of the overhead added by the checkpoint and repetition of the processing in a block.

2.6 Software Black Boxes

The software black box is related to checkpoints and used in certain mission-critical systems to recover to prevent future disasters. The objective of a software black box is to recreate the sequence of the events that led to the software failure for the purpose of identifying the faulty code. The software black box recorder is essential a checkpoint that records and stores behavioural data during program execution, while attempting minimize any fact on that execution. The execution of a program functionalities results in a sequence of module transition such that the system can be described as a module their interaction. When the software is running, it passes control from one module to the next is considered a transition [6]. Call graphs can be developed from these transitions graphically using N×N matrix, where N represents the number of modules in a system. When each module is called, each transition is recorded in a matrix, incrementing that element in a transition frequency matrix. From this, a posterior of transition matrix can be derived that records the likeness that transition will occur. The transition frequency and transition matrices indicate the number of observed transitions and the probability that some sequence is missing in these data [7]. Recovery begins after the system has failed and the software black box has been recovered. The software back-box decoder generates possible functional scenarios based on the execution frequencies found in the transition matrix. The generation process attempts to map the modules in the execution sequence to functionalities, which allows for the isolation of the likely cause of failure.

2.7 User-level Implementation of checkpoint for Multithreaded Application on Windows NT

The existing user-level checkpoint scheme supports only a certain portion of multi-threaded programs on windows operating systems, which are based on single thread programs. Here we focuses on studying a checkpoint scheme to support inter- thread synchronization and quantitative variation of threads for multithreaded process. Unlike other proposed schemes in which thread is suspend by another at checkpoint, this checkpoint scheme employs a strategy by which a thread suspends itself. Therefore, it is free of nondeterminacy of thread, suspension point, thereby ensuring correct rollback recovery. The checkpoint scheme supports also various synchronization objectives such as Mutex, Critical section and Event, as well as Semaphore, WaitableTimer and Thread.

The multithreading paradigm becomes a prevalent programming model for application software, attributing to its capacity that enables a process to execute multiple tasks simultaneously and to exploit effective processor execution resources. Fault-Tolerance in such a multithreading environment also becomes essential requirement. As an effective approach to fault-tolerance, checkpoint recovery should play an important role in reliability of multithreaded

applications. For single threaded programs on Windows, many results have already been reported.

In the checkpoint recovery mechanism, a process state is saved on a stable storage at a proper moment during normal execution. The saved process state is called a checkpoint. When a failure occurs, the program restarts and proceeds from a saved checkpoint.

With respect to a user –level implementation of checkpoint for multithreaded process on windows, inter-thread synchronization and quantitative variation of threads are distinctive features of multithreaded process distinguishing from single-threaded process. In the existing checkpoint schemes, all threads are suspended regardless of whether or not they are in the middle of system API call. This technique can be problematic for successful rollback recovery. We address these issues and propose a new user-level checkpoint scheme with reliable rollback recovery capability.

2.7.1 The impact of thread suspension point on rollback recovery

Process state includes two parts: user space state and process environment. At user level, process environment cannot be dumped directly, because it is located in OS kernel. So it is necessary to assume that application program interacts with system only through Win32 API. As program start up, the checkpoint library is injected into the address space of the process [7] to wrap system API [8] and to create a checkpoint thread responsible for check pointing and recovering. During the normal execution, the program threads calls to system API are intercepted. According to intercepted information, the checkpoint thread can determine process environment at the moment of check pointing. Just before dumping process state at check pointing, check pointing scheme should manage to make all program threads pause to guarantee consistency of the dumped process state. The suspension points of program thread impose an important impact on later roll-back recovery. When the suspension point of a program thread locates in API code segment 1 (ACS1) or kernel code segment (KCS), it is possible to cause failure of later recovery.

If the suspension action is forced at a moment when a program thread is executing API code segment (ACS1), a suspension point will located in API. During recovery execution, it is possible for thread to access to kernel object through a stale handle in its stack, which will result in a failure of recovery execution.

When suspension action is imposed on a program thread that executing kernel code, the thread does not pause. So it will not meet the condition that all threads must pause. Even if a program thread could pause kernel code, the suspension point would be in middle of KCS. Thread will begin to execute from the point AP2 at the later recovery. So kernel code segment from KP to AP2 will not be executed, which can induce a failure in recovery.

2.8 Motivation

Several real-time applications have already been mentioned where fault tolerance is an essential requirement. To further demonstrate the need for fault tolerance, consider the application of real-time system intensive care unit of hospitals. Such systems performs various monitoring task which give early warning of life-threatening situations, such abnormal blood pressure, heart rate, etc. it is essential that such real-time system continue to operate even in the presence of faults. Fault-tolerant medical systems include heart-lung machines used during open-chest surgery and artificial hearts. A system used for medical application which include a wide range of real-time performance requirements, for example, acquisition, processing and immediate display of images in an operating room.

Yet another use of fault-tolerant real-time scheduling is in the field of robotics. Wilfong states that the requirement of reliability of an automated system "is important for safety in an environment where robots and other automated equipment could be hazardous to humans working nearby" the author also adds that reliability is important for economic reasons, since faults can cause an expensive loss. Since many takes automated system have timing constraints, fault-tolerant real-time scheduling is an important requirement in such systems. Even through various kinds of faults can be tolerated by adding redundancy to the system, simply adding redundancy is not sufficient. The additional resources have to be managed such that all timing constraints are met, and faults are guaranteed to be tolerated. To manage the available resources (commonly) in order to achieve timing and fault-tolerance guarantees specialized scheduling algorithms are required.

Many real-time systems focus on the use of hardware redundancy to provide fault-tolerance. The main advantages of using hardware redundancy are that permanent hardware faults can be tolerated. However, hardware redundancy also has some draw-backs. First of all, hardware redundancy mainly targets permanent faults; using massive hardware redundancy to tolerate such faults is excessive. Many real-time systems with a need for fault tolerance cannot incorporate extensive hardware redundancy. An example of such system is mobile robots on factory floors. According to Singh and Murugesan, in their introductory article for a special issue on fault-tolerant system, "no system design can provide fault tolerance for conceivable failure scenario. The trick is to achieve the desired level of dependability by building in protection against the most likely failures, within the given design constraints".

Hardware redundancy has several other disadvantages including heavier weight, larger volume, and greater power consumption and as a result, increased cost. An other disadvantages of using only hardware redundancy is that correlated faults, which occur simultaneously in all hardware modules, cannot be tolerated. For example, electromagnetic radiation may simultaneously affect all hardware modules in a space shuttle.

2.9 Related Work

Check pointing Rollback and Recovery (CRR) is one of the popular temporal redundancy techniques used to achieve fault tolerance in real-time system [1]. However, as performing a check pointing also takes time and consumes resources, we must take into account the check pointing overhead to better predict the satisfaction of constraints in real-time application.

There are two main functions in a CRR protocol that need to be implemented, i.e., the check pointing function where checkpoints are taken periodically and the recovery function where systems are recovered from faults by rolling back to pervious check points. Pervious work in check pointing implementation, such as [2-4], normally accomplish these to function by utilizing multi-threaded processes on general purpose operating system, where main function thread has blocked by check pointing and recovery threads frequently.

3. PROBLEM DEFINITION

How ever, implementation built on non-real-time OS do not provide deterministic pre-emption and inter-process communication, because a kernel space thread cannot be interrupted by other kernel space thread or by user space threads. The OS kernel is "locked" once a kernel function is executing. This usage of locks introduces non-deterministic latencies for both check point and recovery tasks, which are not tolerable in real-time applications.

3.1 Objective

Here, we implement the checkpoint scheme with Real-Time application Interface (RTAI 3.8) which is a popular open source real-time patch for non real-time Linux. We treat main function, check pointing function, and recovery function as real-time tasks with different priorities so that the time to save a checkpoint and recovery from a fault becomes deterministic. It is implemented by using the RTAI 3.8 real-time interruptions and scheduling mechanisms. The check pointing library built on RTAI 3.8 can hence be adopted by real-time application to provide fault tolerance.

3.2 Work Plan for Library Implementation with RTAI

3.2.1 RTAI

The Real-Time Application Interface (RTAI) modifies the general purpose Linux Kernel so that the patch operating system can use the interrupt Abstraction approach to add deterministic real-time characteristic. Specifically, with an additional interrupt abstraction layer on top of general purpose Linux, RTAI can intercepts hardware interrupts before they go to the Linux kernel. RTAI then apply real-time scheduling policies to decide which task shall be run first. Comparing with general purpose Linux. RTAI's task scheduler uses fully preemptive scheduling based on fixed-priority scheme and hence provides predictable behaviour for hard real-time tasks.

For nice feature of RTAI is that it provides a technique named LXRT which allow users to develop and run hard real-time tasks in user space using the same API that is provided in kernel space RTAI. This practice makes the development, debug and test of real-time applications much easier than in the kernel mode. This is the method that we use to implement the check pointing scheme on RTAI.

For each real-time application running on the RTAI Linux, which is called "main function", there are two associated tasks, i.e., the checkpoint task and the recovery task. These two tasks are performed through cooperation of four main modules: a check pointing module, a fault detection module, a fault recovery module, and a main function module. All the modules are implemented as real-time tasks supported by RTAI pre-emption and real-time scheduling services. Since the scheduling is based on priorities, the assignment of priorities set as below given show that higher number representing higher priorites. Main function has priority1, checkpoint has priority2, Fault Recovery has priority2 and Fault Detection has priority3.

4. IMPLEMENTATION

Our checkpoint library is developed in user space with LXRT. Under LXRT, these tasks can be conveniently coded and tested in user space, and at the same time benefit from the real-time characteristic. The implementation based on the deterministic preemption ability offered by the RTAI. With the RTAI scheduler, real-time tasks with higher priority will be able to pre-empt lower-priority tasks, and hence have deterministic timing behaviours.

The first development step is to use the API's provided by LXRT to create each function module as a real-time task associated with a priority specified. Specifically, we use two RTAI functions: rt_task_init_schmod and rt_make_hard_real_time to create a real-time task. There are two things happening after these two functions are called. At first, a task is created and is assigned a priority. In LXRT, however, SCHED_OTHER is the standard Linux default scheduler performs non-preemptable and non-priority scheduling on tasks. So the second function is to switch the scheduling to SCHED_FIFO, which is intended for special and time-critical applications that need precise control over the way in which runnable processes are selected for execution. Processes scheduled with SCHED_FIFO are assigned static priorities in the range from 1 to 99, which mean that when a SCHED_FIFO process becomes runnable, it will immediately preempt a running SCHED_OTHER processor a SCHED_FIFO process of lower priority [5]. A FIFO policy is applied to processes of the same priority. Pre-empted SCHED_FIFO processes remain at the head of their priority queue and resume execution again once all higher-priority processes become blocked, which obviously can help us to predetermine our running order and realize real-time performance.

We have four tasks running concurrently in a system. The main function is then created as a real-time task with priority1, which means it is the lowest priority and can be pre-empted by other higher priority tasks. In order to perform the check pointing functionality, we create a check pointing task with priority 2. Meanwhile, since a checkpoint will be taken periodically, we need to set a real-time timer and make the check pointing task as periodically real-time task by calling the function start_rt_timer to start a real-time timer, and then rt_task_make_periodic to make the timer a periodical one. Then when the time reaches the period, the timer wakes up the check pointing task. There are two possible situations when the check pointing task is up: (1) when the current running task is main function. Since the check pointing task has higher priority, it preempts the running main function and start taking checkpoint. After the checkpoint is taken, another function rt_task_wait_period will be called such that the check pointing will be sent back to sleep and wait for next coming period. The real-time scheduler will then resume the execution of the main function; (2) if the current task is the

fault detection or fault recovery. Since the check pointing task has lower priority, the scheduler will simply block the task until the higher priority tasks finish.

To achieve fault recovery, we need to create two real-time tasks, i.e. the recovery task with priority 2 and fault detection task with priority3. The fault detection task is also periodc. When the timer reaches the fault detection interval, the fault detection tasks preempts all running tasks and sends a “keep alive” signal to the main function. If no response is received, it will report a fault occurrence by sending an RPC signal to the fault recovery task then block itself.

Different from the check pointing and fault detection tasks, the fault recovery task is event-driven instead of time driven. Specifically, it starts as infinity loop and waits for a fault event. When the recovery task receives “fault occurrence” signal from the fault detection task, it calls the function rt_task_resume so that the real-time scheduler put it in the front of the running for execution. The task will read the pervious checkpoint from the persistent storage, and recover the application state accordingly. After the recovery procedure finishes, the recovery task then calls rt_task_suspend function to suspend itself again infinite loop, until the next fault occurrence event arrives.

It is worth that the check pointing frequency has impacts on system performance. In a particular, more frequent check pointing speed up the recovery when the failures occur, and therefore improves the system availability and accelerates the execution time. However, check pointing also takes time and consumes resources. It increases the fault-free execution time and can jeopardize the satisfaction of timing constraints. The check pointing task hence may need to communicate with non-real-time Linux processes to receive adaptive checkpoint

Interval in formation. For instance, a central controller located in a remote process may decide the proper checkpoint interval and send the value to the check pointing task through communication network. The challenge for adaptive checkpoint interval in real-time application is that we need to guarantee that new checkpoint interval can be applied to the application and be effective within predictable time.

RTAI provides a set of real-time Inter Process Communication (IPC) mechanisms that can be used to transfer and share data between tasks in both the real-time and Linux user space domains. These mechanisms include real-time FIFO’s, mailboxes, semaphores, and RPC’s (Remote Procedure Call). In this implementation, we use the mailboxes for check pointing task to receive message from normal Linux tasks.

Specifically, when the check pointing task is resumed by the periodical timer and before it takes a checkpoint, it checks the mailbox queue to see if there is a message indicating the checkpoint intervals. If a new checkpoint interval is detected. The check pointing task finishes saving its current checkpoint first and then calls function next_period. This function resets the time which will be the caller periodical task’s next running period. Since the check pointing task can be guaranteed to obtain the CPU periodically, the adaptive checkpoint interval hence able to applied within a deterministic time range. In fact, if a checkpoint reset message is in the Mailbox queue, and the pervious checkpoint interval is Y, the new value will be effective is no later than 2Y time.

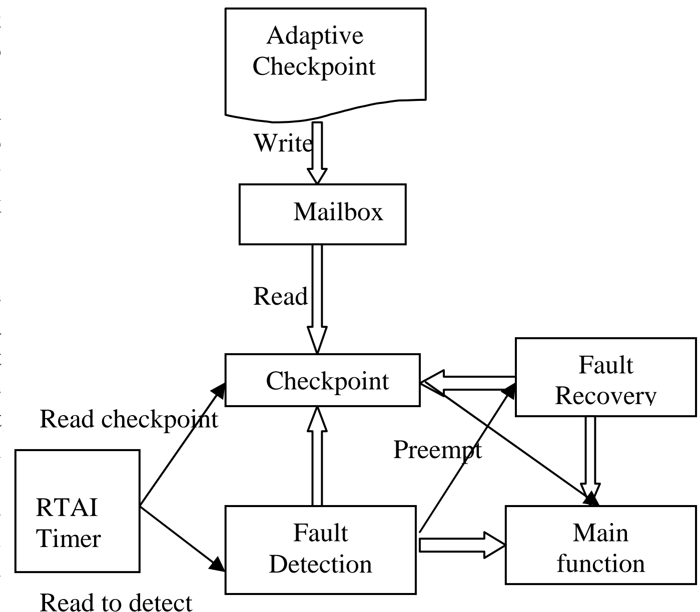


Fig.1 Checkpoint Architecture on RTAI

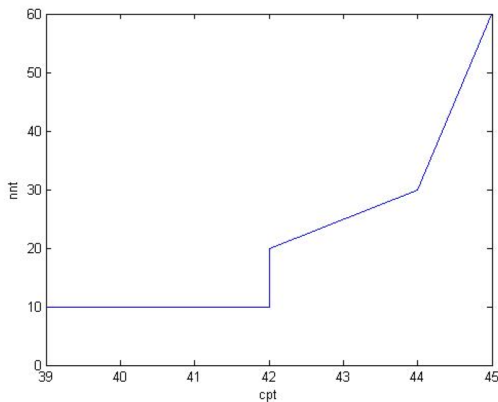
4.1 EXPERIMENT RESULTS

The experiment settings are as follows: a Pentium Dual Core 1.6GHz CPU and 3GB RAM. The system is running on a Ubuntu Linux with kernel version 2.6.18 and an RTAI 3.8 patch. In our experiments, we develop a simple application that adds 1 to the current values starting from 1 until we force it to terminate. The check pointing operation is hence to save the current accumulation value into a file, and the recovery operation is to retrieve the checkpoint (pervious accumulation value) and continue adding values to that.

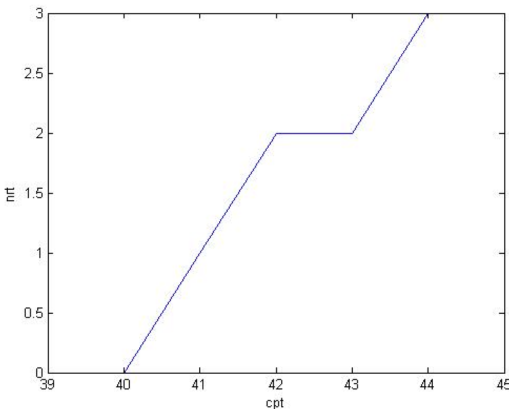
The experiment is to show that the time to take a checkpoint is predictable in our implementation. To test it in a tress environment, we create disturbing threads in the background. Specifically, when the check pointing task starts executing, we run various number of normal Linux dummy threads (priority 0) and lower priority real-time dummy threads (priority=1) in the following order: first, we test the check pointing overhead with no disturbing threads. Here we have take 4 modules to implement the detail result. The production which contains the counter which generates task. The other modules are rollback, error checking and recovery module. Here the Faults are artificial generated by the user. Whenever the user press the CTRL key at that particular time the fault is generated at the system environment. At that particular time the error checking module report that faults was found. Than the rollback module again start from pervious checkpoint. The recovery modules immediately check that faults and repair it as well as it also display the time take to repair the faults. From more than one reading we obtained that it is predetermined.

```
Fult Repair, Time Consule is 39 Microsecond
The Count Value is: 103
The Rollback Value is: 103
^ZFult Created
Fult Repair, Time Consule is 49 Microsecond
The Count Value is: 104
The Count Value is: 105
The Rollback Value is: 105
^ZFult Created
Fult Repair, Time Consule is 48 Microsecond
The Count Value is: 106
^ZFult Created
Fult Repair, Time Consule is 40 Microsecond
The Count Value is: 107
The Count Value is: 108
^ZFult Created
Fult Repair, Time Consule is 39 Microsecond
The Rollback Value is: 105
The Count Value is: 109
```

(1) Time taken to repair a faul t



(1) Normal Linux threads Vs Check-pointing Time



(2) Number of Real-Time Thread Vs Check-Pointing Time

5. CONCLUSION AND FUTUREWORK

In this paper we implemented the checkpoint Rollback Recovery in RTAI 3.8 real-time operating system. The preemptable interrupt service provided by the RTAI makes the checkpoint overhead predictable, so that the checkpoint scheme is feasible to be applied in a real-time application to provide fault tolerance. The experiment result performed on a real system indicates that the checkpoint overhead is close to constants. Our future work is to extend this work to distribute and virtualization environment, where global system states are maintained through synchronized checkpoint protocols. The deterministic synchronization overhead hence needs to be guaranteed by utilizing real-time-aware and inter-process techniques.

REFERENCES

[1] D. Moss, R.G. Melhem, S. Ghosh, “A Nonpreemptive Real-Time Scheduler with Recovery from Transient Faults and its Implementation” IEEE Trans. Software Eng., 29(8): 752-767, 2003.

[2] H. Lee, H. Shin and S. Min. “Worst case timing requirement of real-time tasks with time redundancy”. In Proc. Real-Time Computing Systems and Application. 1999. 410-414.

[3] J-M Yang, D-F. Zhang, X-D. Yang. “User-level Implementation of Check-pointing for Multithreaded Applications on Windows NT”. In proceedings of the 12th Asian Test Symposium.2003.

[4] http://www.aero.polimi.it/~rtai.

[5] http:// www. Realtimelinuxfoundation.org.

[6] W.R. Dieter, J.E Lumpp, Jr. “A User-level Check -pointing Library for POSIX Threads Programs”. 29 th Annual International Symposium on Fault-Tolerant Computing Systems. 1999.

[7] W.R. Dieter, J.E.Lumpp, Jr. “User-level Check-pointing for Linux Threads programs”. USENIX Annual Technical Conference.2001.

[8] Lineo, Inc. “RTAI Programming Guide 1.0”. September 2000.

[9] Y.K. Chen, E. Debes and R. Lienhartetal. “Evaluating and Improving Performance of Multimedia Application on Simultaneous Multithreading Architecture”, IEEE International Conference on Parallel and Distributed Systems, Dec 2002.

[10] F. Karablieh and R.A. Bazzi. “Heterogeneous Check-pointing for Multithreaded Application.” 21th IEEE Symposium on Reliable Distributed System (SRDS’). pp. 140-149, October, 2002

[11] H. Y Zhang, D.S. Wang, W.M.Zheng. “Check-pointing and Rollback Recovery for Windows NT Application.” Journal of computer Research Development, 2001, 38(1), pp.50-55.

[12] J. Srouji, P. Schuster, and M. Bach B etal. “A Transparent Checkpoint Facility on NT.” 2nd USENIX Windows NT symposium, August 1998.