Fault-tolerant Dynamic Scheduling of Object-Based Tasks in Multiprocessor Real-time Systems

INDRANIL GUPTA*
DEPT. OF COMPUTER SCIENCE
CORNELL UNIVERSITY, USA

G. MANIMARAN *
DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING
IOWA STATE UNIVERSITY, AMES, IA 50011, USA

C. SIVA RAM MURTHY
DEPT. OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY
MADRAS 600 036, INDIA

gupta@cs.cornell.edu, gmani@iastate.edu, murthy@iitm.ernet.in

Abstract. Multiprocessor systems are fast emerging as a powerful computing tool for real-time applications. The reliability required of real-time systems leads to the need for fault-tolerance in such systems. One way of achieving fault-tolerance is by Primary-Backup (PB) approach in which two copies of a task are run on two different processors. In this paper, we compare and contrast three basic PB approaches - (i) primary-backup exclusive, (ii) primary-backup concurrent, and (iii) primary-backup overlapping - in the context of dynamic scheduling of object-based real-time tasks. The objective of this paper is threefold: (a) to extend the PB-based fault-tolerant approaches, hitherto applied only to conventional real-time tasks, to object-based real-time tasks, (b) to compare these three approaches, in terms of schedulability and implementation complexity, and (c) to propose a dynamic scheduling algorithm for object-based real-time tasks, which can be used in conjunction with any of these PB-based fault-tolerant approaches. We have conducted extensive simulation studies to evaluate the performance of these three approaches, for tasks with resource and precedence constraints, for a variety of task and system parameters. Our simulation studies reveal some interesting results about the relative performance of these approaches.

1. Introduction

Safety-critical real-time applications are required to be *predictable* and *reliable*. The capability for high performance and reliability offered by multiprocessor systems have made them emerge as a powerful computing tool for safety-critical real-time applications like avionic control and nuclear plant control. In order to satisfy the

* This work was done when the authors were at the Dept. of Computer Science and Engg., Indian Institute of Technology, Madras, INDIA.

predictability that a real-time system demands, scheduling assumes great importance in multiprocessor systems. The problem of scheduling real-time tasks on multiprocessors has attracted considerable research in the past. The problem of scheduling of real-time tasks in multiprocessor systems is to determine when and on which processor a given task executes [15]. This can be done either statically or dynamically. In static algorithms, the assignment of tasks to processors and the time at which the tasks start execution are determined *a priori*. Static algorithms are often used to schedule periodic tasks with hard deadlines. However, this approach is not applicable to aperiodic tasks whose characteristics are not known *a priori*. Scheduling such tasks require a dynamic scheduling algorithm.

In dynamic scheduling, when a new set of tasks (which correspond to a plan) arrive at the system, the scheduler dynamically determines the feasibility of scheduling these new tasks without jeopardizing the guarantees that have been provided for the previously scheduled tasks. A plan is typically a set of actions that has to be either done fully or not at all. Each action could correspond to a task and these tasks may have resource requirements, and possibly may have precedence constraints. Thus, for predictable executions, schedulability analysis must be done before a task's execution is begun. For schedulability analysis, tasks' worst case computation times must be taken into account. A *feasible* schedule is generated if the timing constraints, and resource and fault-tolerant requirements of all the tasks in the new set can be satisfied, i.e., if the schedulability analysis is successful. If a feasible schedule cannot be found, the new set of tasks (plan) is rejected and the previous schedule remains intact. In case of a plan getting rejected, the application might invoke an exception task, which must be run, depending on the nature of the plan. This planning allows admission control and results in reservation-based system. Tasks are dispatched according to this feasible schedule. Such a type of scheduling approach is called *dynamic planning based scheduling* [15], and Spring kernel [19] is an example for this. In this paper, we use dynamic planning based scheduling approach for scheduling of tasks with hard deadlines.

The demand for more and more complex real-time applications, which require high computational needs with timing constraints and fault-tolerant requirements, have led to the choice of multiprocessor systems as a natural candidate for supporting such real-time applications, due to their potential for high performance and reliability. Due to the critical nature of the tasks in a hard real-time system, it is essential that every task admitted in the system completes its execution even in the presence of failures. Therefore, fault-tolerance is an important issue in such systems. In real-time multiprocessor systems, fault-tolerance can be provided by scheduling multiple versions of tasks on different processors. Four different models (techniques) have evolved for fault-tolerant scheduling of real-time tasks, namely, (i) N-version programming [1], (ii) Primary Backup (PB) model [2, 12], (iii) Imprecise Computational (IC) model [8], and (iv) (m, k)-firm deadline model[1] [16].

In the N-version programming approach, N versions of a task are executed concurrently and the results of these versions are voted on. If N is 2, single fault can

be detected; if it is 3, single fault can be located. In [9], real-time task scheduling algorithms with fault detection and location capabilities have been proposed. In the PB approach, two versions are executed on two different processors, and an *acceptance test* is used to check the result. In the IC model, a task is divided into mandatory and optional parts. The mandatory part must be completed before the task's deadline for acceptable quality of result. The optional part refines the result. The characteristics of some real-time tasks can be better characterized by (m, k)-firm deadlines in which m out of any k consecutive tasks must meet their deadlines. The IC model and (m, k)-firm task model provide scheduling flexibility by trading off result quality to meet task deadlines.

The different methods employed for error detection often make one technique preferable to the other in certain applications [13]. The N-version programming approach can be applied to any application, but its resource utilization and hence the schedulability is very poor. The PB approach can be applied to most of the applications where acceptance test exist for checking the correctness of the results. IC and (m, k)-firm models are applicable in image processing and radar tracking applications.

Applications such as automatic flight control and industrial process control require dynamic scheduling with PB-based fault-tolerant requirements. In a flight control system, controllers often activate tasks depending on what appears on their monitor. If dynamic scheduling is employed in this system, when an airplane running on autopilot experiences wind turbulence, and the additional task generated due to disturbance cannot be executed while providing fault-tolerance, then the pilot has the option of taking over manual control of some or all functions of the airplane's navigational system.

The fault-tolerant scheduling of object-based real-time tasks is a problem of growing interest and assumes significance due to the following reasons. Real-time systems software is inherently large and complex. The complexity in the development of software for such systems can be managed by using object-based design and methodology [21]. However, even though reusable software components contained in the object-based implementation of an application have advantages such as information hiding and encapsulation, execution efficiency may have to be sacrificed due to the large number of procedure calls and contention for accessing shared software components. Further, making object-based applications reliable is a challenging problem as protocols have to be developed to maintain the consistency of object data stores and method calls, in the presence of faults. The problem of fault-tolerance in object-based real-time systems is currently a wide topic of research [7].

For conventional task models, three different PB-based fault-tolerant approaches exist. The two most popular PB approaches are the *Primary-Secondary Exclusive* (PS-EXCL) and the CONCURrent approaches. PS-EXCL[2] is the most widely used PB approach where the primary and backup copies of the tasks are excluded in space (processor) and time [2, 12]. CONCUR proposes a concurrent execution of the primary and backup copies of each task [6, 7]. This approach obviously

involves unnecessary use of resources if faults rarely occur. A third approach is possible, namely, OVERLAP [7, 20]. This approach is a combination of PS-EXCL and CONCUR and is flexible enough to exploit their advantages according to the system parameters.

The objective of this paper is threefold: (i) to extend the PB-based fault-tolerant approaches, hitherto applied only to conventional real-time tasks, to object-based real-time tasks, (ii) to compare these three approaches, in terms of schedulability and implementation complexity, and (iii) to propose a dynamic scheduling algorithm for object-based real-time tasks, which can be used in conjunction with any of these three approaches. This paper provides valuable inputs for the system developers to choose an appropriate fault-tolerant approach depending on the application requirements. To the best of our knowledge, there has been no prior work which deals with dynamic scheduling and fault-tolerance for the object-based task model.

The rest of the paper is organized as follows. Section 2 describes the object-based task model and the application of the three PB-based fault-tolerant approaches for this task model. In Section 3, we present the dynamic scheduling algorithm to be used with any one of the fault-tolerant schemes. In Section 4, we discuss the simulation results of these approaches. Finally, in Section 5, we make some concluding remarks.

2. Object-based Task Model and PB-based Fault-tolerant Approaches

In this section, we show how to extend the three PB approaches to achieve fault-tolerance for object-based real-time tasks. We use the object-based task model of [21]. [7] discusses a fault-tolerant approach for a real-time object model called the *RTO.k* model, but does not focus on the scheduling aspect.

2.1. Programming Model

The complexity in developing object-based real-time applications is conquered by decomposing them into a set of *programs*. The programs are further divided into *classes*, where each class corresponds to a reusable software component. The way in which the application is divided into different programs and the way in which each program is divided into classes, depends on the characteristics of the application. It is assumed that the *classes* or *reusable software components* are implemented as either abstract data types (ADTs) or abstract data objects (ADOs) [21, 22]. The deterministic object-based task model is used, where the worst case execution time of each method can be fixed at schedule time.

2.2. The Object-based Task Model

The application is designed as a number of object-based tasks. These tasks may arrive at different times at the scheduler. The structure of each task is shown

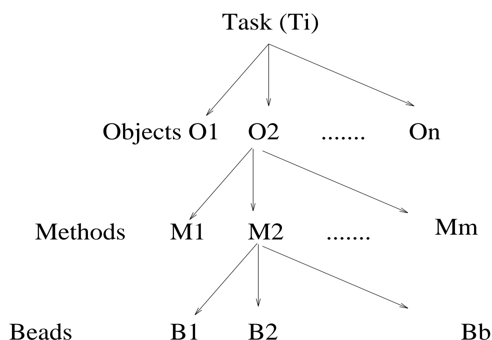

Figure 1. Object-based task model

in Figure 1. Each task contains a set of objects which are instances of software components spread over a set of programs. The software components are either ADTs or ADOs. Different tasks may access the same method or methods of the same object and hence contention to access the software components may occur. Note that these access contentions may exist between tasks that have arrived at different times also. One way to resolve such contention is to *clone* (replicate) the software component on another processor so that more than one data item can be processed at a time. However, *stateful software components* are those which have a state associated with them making cloning a costly operation. It is assumed that only stateless software components are cloned. Some software components represent the resources of the system (like, say, access to a port) and are called *environment dependent software components.* It is assumed that such software components are also not cloned. As the number of stateful software components and environment dependent software components increases, the schedulability of the system decreases.

Each software component or object has a set of methods operating on data encapsulated in the object. At run-time, the objects communicate with each other through method calls. If the caller and the callee methods are assigned to the same processor, then the method call can be implemented by a *local procedure call* (LPC). If they are on different processors, a remote procedure call is used. A remote procedure call can be either an *asynchronous remote procedure call* (ARPC) or a *synchronous remote procedure call* (SRPC). In an SRPC, the caller gets blocked after making the call and remains blocked until the callee returns. In an ARPC, the caller can continue execution after making an RPC until the point that it will need the results from the callee.

The methods of the different software components may also require to use the global *resources.* This is apart from those provided by the environment-dependent software components. These are modeled as *resource requirements* of the methods. These resource constraints may change during the execution of the method, but only at known points. The points at which the external procedure calls are made or return, or at which the resources required by the method change are known as

preemption points. We use the semi-preemption model wherein two consecutive preemption points constitute a non-preemptible entity called the *bead*. Thus every method is the combination of one or more such beads, each bead having some resource requirements (or constraints). Each bead is composed of two parts: (i) a computation code, followed by (ii) a set of output actions.

The output actions correspond to the method calls made at the end of the bead or any other output actions (like writing to a log, freeing some global resources, etc). Thus the *bead* becomes the smallest schedulable entity in the system. The output actions which are part of a bead's execution are actually global memory writes done by the processor. In a distributed system, these output actions would have been scheduled on the communication channel and would not be a part of the bead execution time.

We observe that the object-based task model is analogous to the conventional task model. The *bead* in the object-based model is analogous to the *task* of the conventional task model. The precedence constraints among the beads are imposed by the procedure calls and continuation of code of the same method, and the resource constraints determined by the contention for objects as well as the resource requirements of individual beads. This means that any scheduling algorithm *or* fault-tolerant approach which is applicable to the conventional task model must be applicable to the object-based model also with a few changes. With this, we are off to adapting the three fault-tolerant schemes - PS-EXCL, CONCUR, and OVERLAP to the object-based task model.

2.3. PB-based Fault-tolerance for Object-based Task Model

2.3.1. Problem Statement Our fault-tolerant schemes for object-based tasks will try to tolerate the following types of faults: (1) a fault in the software design itself, leading to a fault in the execution of the bead, (2) a transient fault in a processor, leading to a fault in the execution of the bead, (3) a run-time fault in the output action of a bead, (4) a processor crash, and (5) object data store loss or method crash caused by local memory crash in a processor.

The problem is to design a fault-tolerant scheme so that any application program in the object-based task model that is scheduled on a multiprocessor system prone to these faults will be able to run consistently, correctly, and timely (within the deadline) under the assumption that not more than one fault occurs per bead.

2.3.2. Adapting PB Approaches to Object-based Task Model The starting point for our solution to this problem are the observations that the *bead* is the smallest non-preemptible unit in this object-based task model, and that the consistency, timeliness, and correctness of an object-based application depends on (i) all beads being executed correctly according to the bead precedence graph, (ii) the correctness of the *output actions* of these beads [7], and (iii) consistency of the object data store at the beginning of every bead execution.

If these conditions are satisfied in the presence of the above listed faults, the correctness and hence fault tolerance of individual methods and object-based tasks will be automatically guaranteed. To handle faults, we use the two-pronged approach of fault *detection* and *recovery* from faults.

Fault Detection: The first three types of faults are the most difficult to detect. They are *detected* in the following ways:

- Faults of types 1 and 2 are handled by including an *acceptance test* (AT) at the end of the computation code and before the output actions of a bead.
- Faults of type 3 can be detected by the processor executing the output action. If the *write* to the global memory (which is what the output action is) completes successfully, the output action is correct, otherwise it is faulty.
- Faults of types 4 and 5 can be detected as a processor crash.

Fault Recovery: This is done by using the PB approach. Our fault-tolerant strategy maintains 2 active versions of every fault-tolerant object (object data store segment and code) or fault-tolerant method on the two different processors. These two versions are called the *primary* and the *secondary* (or *backup*) versions. All beads of a fault-tolerant method or object are replicated and scheduled on two processors so that at run-time, if one of the beads fails, the other bead will complete the operations and do the requisite output actions. The assumption made here for this fault-tolerant scheme to be successful is that if the primary version of a fault-tolerant bead fails, its secondary version completes successfully. Thus, the worst case computation time of each fault-tolerant bead B now consists of

1. A global memory read of the finishing status of the earlier bead(s) in the precedence graph : $r(B)$
2. A computation code : $c(B)$
3. AT, followed by a write of the AT's result into a global variable $ATRrecvd(B)$, present for every bead - the time for these two are included in $c(B)$
4. Set of output actions, followed by a write of the OSN's result into a global variable $OSNrecvd(B)$, present for every bead : $w(B)$

The *read* (1) is needed due to the following reason. A version of a fault-tolerant method or object can be inconsistent if the last executed bead belonging to that version terminated prematurely because of a fault. So, at the beginning of every bead, a *read* is done, from the global memory, of the results of the preceding bead(s) in the precedence graph. Thus, if a fault occurs in a bead of one version of a fault-tolerant object or method, when the *next* bead copy (either primary or backup) of this inconsistent version (on this processor) starts executing, it will read off the consistent state from the global memory and then move on to the computation code, guaranteeing correct bead execution. So, the output actions of the primary copy of any fault-tolerant bead also includes global memory writes giving information

of local method variables and object data store variables that have been modified during the bead execution. These writes actually constitute a store checkpoint and the read actually constitutes a compare checkpoint.

Here, we give an explanation of two global variables associated with every bead and one global variable associated with every bead copy. The former two are required for the check-pointing and the latter for setting the version type of a bead copy. For every bead B, we have two variables namely *ATRrecvd(B)* and *OSNrecvd(B)*, which are initialized to FALSE. These indicate the result (success/failure) of the AT and the output actions respectively of the primary copy of B. Thus, if the primary copy of B executes its AT successfully, it sets *ATRrecvd(B)* to TRUE. The same holds for output actions and *OSNrecvd(B)*. These are to intimate to the backup copy of B the results of the AT and the output actions performed by bead B's primary copy. Thus a write to any of these global variables would be a store checkpoint and a read from any of them is a compare checkpoint. In addition, each copy of bead B has a variable $verflag$ which takes values PRIMARY and BACKUP and stands for the version type of that particular copy. These are set by the scheduler or the resource reclaiming algorithm depending on the fault-tolerant technique. These variables are created and initialized by the scheduler and can be destroyed when both the versions of a bead finish. *verflag* is the global variable that is used to 'mark' the version type of a bead copy.

Have we satisfied the three conditions set down at the beginning of this section for the timeliness and correctness (consistency) of the object-based task model ? The *read* at the beginning of every bead satisfies the third condition. In the following three sections, we describe the application of the three PB approaches to object-based tasks to satisfy the first two conditions. The fault-tolerant techniques replicate beads in a similar way to the tasks' replication. The two copies (primary and backup) of every bead ensure that every bead executes correctly and the intended output actions of the beads are successful and hence help to maintain fault-tolerance for the execution of the entire object-based tasks.

Note that because of faults of the types 1 and 3 which arise from the software design stage, *all the method invocations need not be fault-tolerant.* Only some methods or objects, namely those that are likely to fail can be made fault-tolerant. We will refer to such objects or methods as *fault-tolerant* objects or methods, respectively, and their beads as *fault-tolerant beads.* Thus this scheme has sufficient flexibility.

2.3.3. PS-EXCL Fault-tolerant Scheme to Object-based Tasks In this approach, for each fault-tolerant bead B, its backup copy is scheduled to start execution only after its primary copy finishes execution. This time exclusion is maintained during run-time also. Figure 2 shows the primary and backup copies as they are scheduled onto the dispatch queues. Their execution at run-time is described below.

The primary copy of B works as follows. It does the computation, the AT and writes the result of the AT into *ATRrecvd(B)*. If the AT fails, the primary terminates execution. If the AT is correct, the primary tries to do the output actions, writes

the correctness of the output actions into *OSNrecvd(B)* and terminates. Note that *OSNrecvd(B)* indicates the completion of the computation and output actions of bead B. So if the primary copy fails *OSNrecvd(B)* will remain false when the secondary copy starts execution.

When the secondary copy of B starts execution, it first checks if the primary has finished correctly by checking *OSNrecvd(B)*. If *OSNrecvd(B)* is true, it exits as the primary has finished successfully, otherwise it executes its version of the computation code, does the AT and the output actions.

Thus this fault-tolerant scheme ensures that the computation code of every bead is executed correctly and output actions of every bead are correct.

While the store checkpoint of the primary copy has been made into the ATR and OSN writes (for computation and communication respectively), the compare checkpoint at the backup copy's beginning has been made a part of the *read*.

Figure 2 shows how the two versions of bead B will look like when they are scheduled. Note that the total (worst case) execution time of each version (copy) of a bead B under PS-EXCL is $(r(B) + c(B) + w(B))$.

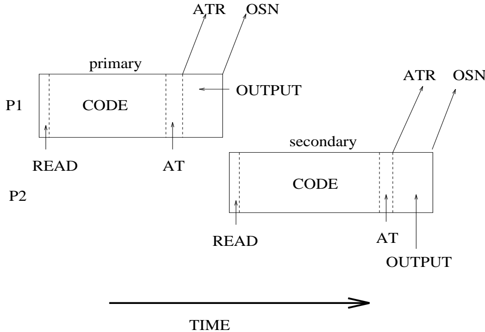

Figure 2. PS-EXCL: Primary and secondary versions of a bead (as scheduled)

2.3.4. CONCURrent Fault-Tolerant Scheme to Object-based Tasks In this approach, the two versions of any given fault-tolerant bead B are scheduled (and run) concurrently or simultaneously. Figure 3 shows the primary and backup copies as they are scheduled onto the dispatch queues. Their execution at run-time is described below. The two versions are marked as primary and secondary arbitrarily at run-time by the resource reclaiming algorithm. The primary copy of B works as follows. It does the computation and the AT, writes the result of the AT into *ATRrecvd(B)*. If the AT fails, the primary terminates execution. If the AT is cor-

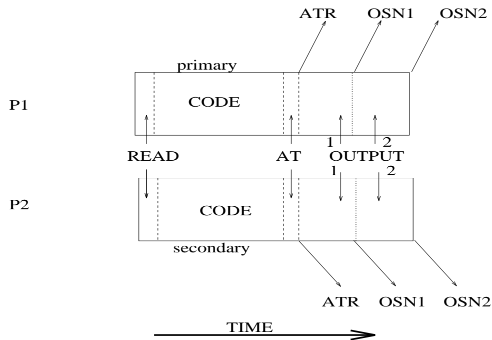

Figure 3. CONCUR: Primary and secondary versions of a bead (as scheduled)

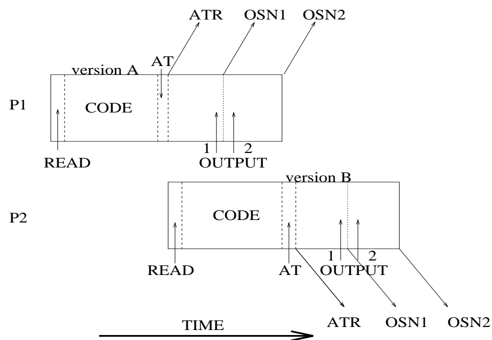

Figure 4. OVERLAP: Primary and secondary versions of a bead (as scheduled)

rect, the primary tries to do the output actions. It writes the correctness of the output actions into *OSNrecvd(B)* and terminates.

The secondary copy of B works as follows. On starting execution, it first checks if the primary copy of B has finished correctly. If yes, it exits, otherwise it executes its version of the computation code, does the AT and then checks if the primary has executed completely and correctly. If yes, it exits, otherwise it checks if the primary has failed. If the primary has failed, it does the output actions else it waits for a time-out period equal to the worst case output time. Note that at the end of this time-out, as the secondary does not start before the primary, the primary would certainly have finished its execution (both computation and output), either successfully or after a fault. Then the secondary checks if the *OSNrecvd(B)* is still FALSE (meaning a fault has occurred in the primary). If no, it exits otherwise, it completes the output actions. Note that the total (worst case) execution time of

each version (copy) of a fault-tolerant bead B under CONCUR is $(r(B) + c(B) + 2 * w(B))$. The worst case execution time of a bead includes twice the output time as a fault can occur during the output actions. This did not arise in PS-EXCL as the two copies are time excluded. The total (worst case) execution time for a non-fault-tolerant bead B remains $(r(B) + c(B) + w(B))$.

Thus this fault-tolerant scheme also ensures that the computation code of every bead is executed correctly and output actions of every bead are correct. The store checkpoint of the primary copy and the compare checkpoint of the backup have been made into the ATR and OSN writes and reads (for computation and communication) respectively. Figure 3 shows how the two versions of a bead will look like when they are scheduled.

2.3.5. OVERLAP Fault-tolerant Scheme to Object-based Tasks In this approach, the two versions of any given fault-tolerant bead B are scheduled (and run) in an overlapping manner. Figure 4 shows the primary and backup copies as they are scheduled onto the dispatch queues. Their execution at run-time is described below. The scheduling of the two versions of a fault-tolerant bead is done run-time, the bead copy starting execution first becomes the primary and the other copy the secondary.

The compare checkpoint of the secondary is a part of the *read* at the beginning of the bead copy. The ATR and OSN arrows shown become outputs for the primary (store checkpoints) and reads for the backup copy (compare checkpoint). *verflag* is set by the resource reclaiming algorithm. In this way, this fault-tolerant scheme ensures that the computation code of every bead is executed correctly and output actions of every bead are correct.

Figure 4 shows how the two versions of a bead will look like when they are scheduled. Note that the total (worst case) execution time of each version (copy) of a fault-tolerant bead B under OVERLAP is $(r(B) + c(B) + 2 * w(B))$. The total (worst case) execution time for a non-fault-tolerant bead B is $(r(B) + c(B) + w(B))$.

2.4. Criteria for Comparison of the PB Approaches

A fault-tolerant algorithm is most viable when it ensures that for a task set with resource and/or precedence constraints among the tasks, all of the above types of faults are tolerated at run-time while the number of tasks accepted by the system is increased. This can be primarily obtained by:

- Reducing the pre-run schedule length for a task set.
- Reducing the post-run schedule length for a task set: this can be obtained by avoiding unnecessary execution of task copies in the absence of failures, and also exploiting the early completion of tasks. This is called the issue of resource reclaiming [10, 18].

These criteria ensure that more tasks will be scheduled even if they have tight deadlines. Also, newly arriving tasks will be more likely to be accepted if already

scheduled tasks finish quickly. Therefore these are the two criteria in our comparison of the PB approaches. The advantages and disadvantages, we present, for the PB approaches will be based on these criteria. These criteria clearly reflect in the results of the simulation study in Section 4.

3. Dynamic Scheduling Algorithm for Object-based Real-time Tasks

In this section, we first describe a dynamic scheduling algorithm for the object-based task model. This can be used in conjunction with any of the three PB approaches to fault-tolerance. This is followed by an example of the application of this scheduling algorithm along with the PB approaches.

3.1. System Model

We assume a multiprocessor system with m processors and a shared global memory. Each processor has a local memory (for storing the codes and other local variables associated with the execution of the tasks) and a set of associated resources, which can be accessed only by itself. The shared global memory has a set of (logical) resources which are accessible by all the processors in either exclusive or shared mode.

The access time of the shared memory for any processor needs to be bounded to guarantee predictability in the system. This is assumed to be provided say, through an interleaved access scheme to the global memory for each processor, similar to the TDMA scheme used to guarantee bounded message delivery time in distributed systems. Hence, the global memory access clashes among the processors are taken into account in scheduling by just including the maximum access time for each shared memory read/write into the worst case computation time of the tasks.

Dynamic scheduling algorithms can be either centralized or distributed. In our simulation, we assume a centralized scheduling scheme. In a centralized scheme, all tasks arrive at a central processor called the *scheduler*, from where they are distributed to the other processors of the system. The communication between the scheduler and the processors is through *dispatch queues*. Each processor has its own dispatch queue. This organization, shown in Figure 5, ensures that the processors will always find some tasks in the dispatch queues when they finish execution of their current tasks. The scheduler runs in parallel with the processors, scheduling the newly arriving tasks and updating the dispatch queues.The scheduler has to ensure that the dispatch queues are always filled to their minimum capacity (if there are tasks left with it) for the parallel operation. This minimum capacity depends on the worst case time required by the scheduler to reschedule its tasks upon the arrival of a new task. If a permanent processor failure is detected, the scheduler excludes the failed processor from the scheduling algorithm i.e., no further tasks are scheduled on that processor. The scheduler is susceptible to becoming a bottleneck or even single-point failure. This can be prevented by making the

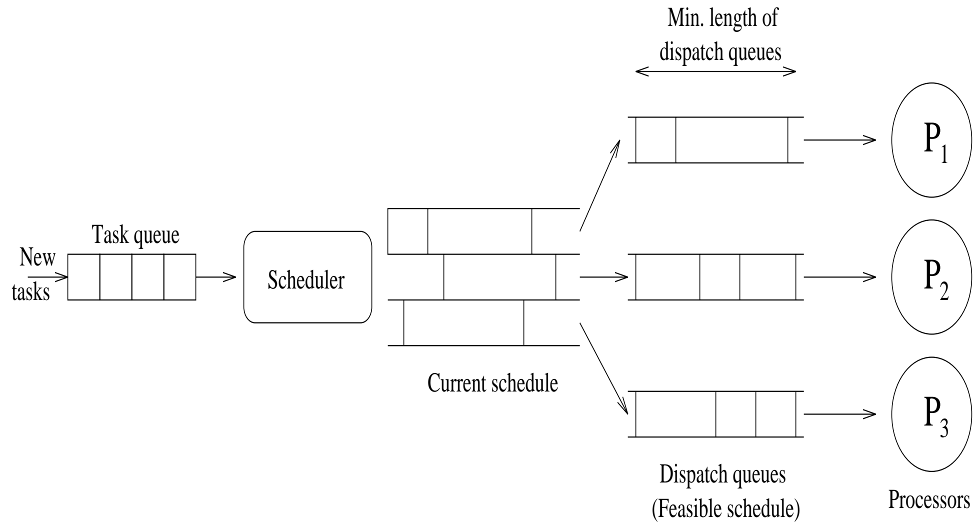

Figure 5. System model

scheduler consist of more than one processor and having the scheduling algorithm run across these processors to achieve both faster execution and fault-tolerance.

Resource reclaiming [10, 18] is the strategy used to reclaim resources when: (i) a task completes earlier than its worst-case computation time or (ii) the backup copy of a task does not get executed as the primary has already completed execution successfully. The resource reclaiming algorithm, invoked at the completion of every task on a processor, seeks to schedule the next task in the dispatch queue (DQ) ahead of its scheduled start time. We use the RV algorithm [10] for resource reclaiming. Here, the scheduler builds a restriction vector (RV) for each task T that it schedules. The RV is an m-component vector (m being the number of processors), where each entry $RV_i(T)$ is the last task scheduled prior to T on processor P_i which has a resource conflict or precedence relation with T. When a task finishes execution, it runs the RV algorithm. It checks the first task in the DQ's of all idle processors and starts that task immediately if all the tasks in its RV have finished execution.

3.2. Dynamnic Scheduling Algorithm

A fault-tolerant mechanism for object-based tasks will not be complete without the development of an effective dynamic scheduling algorithm for object-based tasks which can be used in conjunction with the fault tolerant mechanism. Most algorithms in the real-time scheduling area of literature today [14] solve the dy-

namic/static scheduling problem for the conventional task model. On the other hand, object-based tasks involve a number of methods belonging to different objects and method calls between methods. This leads to a different (*object-based*) task model. As such, an immediate need to bridge this gap arises: a need to develop algorithms for scheduling object-based tasks on multiprocessor systems.

Most of the existing literature in object-based scheduling tries to overcome the difficulties in using reusable software components like execution overhead due to large number of procedure calls and contention for shared software components [15, 21-24] except [26], where software components are assigned and scheduled on the processors based on inter-task parallelism and processor utilization computed using heuristic techniques. An algorithm to assign reusable software components which exploits parallelism with minimum number of processors by the introduction of ARPCs is described in [25]. A model for pre-run-time scheduling of object-based distributed real-time systems that are composed of ADTs and ADOs is proposed in [21, 23]. In addition, they present an incremental scheduling approach which constructs an initial schedule and modifies it by enhancing concurrency through ARPC and cloning, till a feasible schedule is obtained. The work in [24] presents compiler techniques for identifying concurrency among software components via ARPCs and cloning in the context of an incremental scheduling algorithm. [22] considers static scheduling of periodic tasks having precedence constraints among them, compared to the multiple independent tasks considered in [24].

In this section of the paper, we propose an algorithm for the dynamic scheduling of object-based real-time tasks on multiprocessors. In our algorithm, we use ARPC and cloning to achieve better schedulability. The algorithm is based on Spring scheduling algorithm and is shown in Figure 6. The complexity of steps 2 to 6 is $O(n)$ each where n is the total number of bead copies to be scheduled. Step 7 is $O(Kn)$. As K is usually a small number [14], this is $O(n)$. Step 8 is also $O(n)$. Thus the entire scheduling algorithm has a complexity of $O(n)$.

The scheduling algorithm has four steps: (i) setting the precedence constraints among beads, (ii) clustering the beads into entities, (iii) allocation step, and (iv) scheduling step.

3.2.1. Precedence Constraints among the Beads while Scheduling Initially, we are given the original set of precedence constraints among the beads (without considering their fault-tolerant versions) as a set P'. P' corresponds to the precedence constraints among the beads as per the method calls and the method bead orders in each method. It consists of pairs of the form $(A > B)$, where A and B are beads belonging to the tasks and '>' means 'should be executed before'. We shall call the precedence relation graph among the beads defined by P' as the *Comprehensive Entity Invocation Graph* or the *Initial Bead Precedence Graph.* We construct a precedence relation set P from P' by taking into account the precedence and fault-tolerance constraints among the beads. This is done as follows (A and A' refer to the two fault-tolerant copies of bead A; A is the primary and A' the backup).

$P = \phi$
For each $(A > B)$ in P',
  $P = P \cup \{(A > B)\}$
  if(A is fault-tolerant and B is not fault-tolerant)
    $P = P \cup \{(A' > B)\}$
  else if(A is not fault-tolerant and B is fault-tolerant)
    $P = P \cup \{(A > B')\}$
  else if(A and B are both fault-tolerant)
    $P = P \cup \{(A > B'), (A' > B), (A' > B')\}$

If(PS-EXCL scheme is being used)
  For each fault-tolerant bead B in P'
    $P = P \cup \{(B > B')\}$

We shall call the graph among the beads defined by P as the *Bead Precedence Graph*. Our scheduling algorithm uses this graph.

3.2.2. Allocation Step - Clustering the Beads into Entities Here we cluster the beads into *entities*. An *entity* is a bunch of beads all of which have to be allocated to one processor only. Without cloning of methods, the *entities* will just be the software components themselves. However, with cloning, we create an entity for each

1. Fault-tolerant copy of each stateful or environment dependent ADT (as it cannot be cloned),
2. Each fault-tolerant copy of each method invocation exported by each stateless ADT.

For example, consider an ADT O_i having two methods M_{i1} (invoked once) and M_{i2} (invoked twice). Let us first consider the case where neither the ADT nor any of its methods is fault-tolerant. If O_i is a stateful or environment dependent ADT, there will be only one entity for all the beads of O_i, that is, all invocations of M_{i1} and M_{i2}-thus all beads of all invocations of these methods will be scheduled on the processor to which this entity is allocated. However, if O_i is not stateful or environment-dependent, three entities will be created - two for each invocation of M_{i2} and one for the invocation of M_{i1}. Now, if any of these methods or the ADT is fault-tolerant, there will be similar duplicate entities for the fault-tolerant versions. For example, in the second case, if M_{i2} is fault-tolerant, we will have to create two pairs of entities, each pair corresponding to a fault-tolerant invocation of M_{i2}.

3.2.3. Declustering Heuristic and Allocation Step Let M_{ij} and M_{pq} be two communicating methods. M_{ij} denotes the jth method of entity E_i and M_{pq} denotes the qth method of entity E_p. Let M_{ij} call M_{pq}, $NMC(M_{ij}, M_{pq})$ times. The net advantage of declustering the two methods M_{ij}, M_{pq} is:

$$NADV_{ij,pq} = \Sigma_{k=1}^{k=NMC(M_{ij},M_{pq})}[pt_k + E(M_{pq}) - Max(pt_k, cr_k + cs_k + E(M_{pq})] \quad (1)$$

where $pt_k = \sum_l B_{ijl}$ is the sum of the execution times of the beads of M_{ij} that can be executed in parallel with the kth call to M_{pq}. cs_k and cr_k denote the amount of data communicated from M_{ij} to k^{th} call of M_{pq} and back during the returning of the method call (each of cs_k and cr_k stands for the sum of the worst case read + write times for the communication(s) in question). The first term in $NADV$ (equation 2) denotes the time needed to execute M_{ij} and M_{pq} if they are assigned to the same processor. The second term denotes the time they may take if they are executed on different processors. The difference of the two gives the gain/loss in declustering the two methods. For two communicating entities E_i and E_p, the net gain/loss in declustering them is

$$CNADV(E_i, E_p) = CNADV(E_p, E_i) = \sum_j \sum_q [NADV_{ij,pq} \times NM(M_{ij})] + Y \tag{2}$$

where $Y = \sum_q \sum_j [NADV_{pn,ij} \times NM(M_{pn})]$

The higher the value of $CNADV(E_i, E_p)$, the better it is to decluster the two entities E_i and E_p. Note that the values of $CNADV$ for all pairs of entities can be found in time *O(number of beads in the bead precedence graph)* by making a depth/breadth-first search of the graph.

When the first bead belonging to an entity E_i comes up for scheduling, the following heuristic $\rho[p]$ is calculated for the entity with respect to all processors $p = 1..m$ (m being the total number of processors).

$$\rho[p] = \frac{G}{H} + \frac{ProcLoad[p]}{max_{q=1}^{m}(ProcLoad[q])} \tag{3}$$

where $G = \sum_{\text{All entities } E_j \text{ on proc. p}} CNADV(E_i, E_j)$ and
$H = max_{q=1}^{m}(\sum_{\text{All entities } E_j \text{ on proc. q with which it } E_j \text{ has communication}} CNADV(E_i, E_j))$

The second term in the above heuristic takes care of equal load distribution on all processors and the first term seeks to minimize communication among beads executing on different processors. This heuristic is used as follows. The entity E_i is allocated to processor p with the minimum value of $\rho[p]$. Thereafter all beads belonging to that entity are scheduled on that processor only.

3.2.4. Scheduling Step The Spring scheduling strategy is used. At every invocation of the scheduling algorithm, all the beads to be scheduled are ordered in non-decreasing order of their deadlines in a *Bead Queue (BQ)* At every step of the scheduling algorithm, the myopic algorithm heuristic H is applied to schedule the beads based on their precedence and resource constraints. However, the calculation of EST of a bead B will involve an additional term apart from the bead resource and precedence constraints. If B is the first bead of a method invocation, $EST(B)$ will also involve the earliest time that the entity to which that bead belongs will become free. This is modelled in a similar way as the resource constraints. Once a bead has been chosen from the BQ according to the H heuristic, it is first checked whether the entity to which the bead belongs has been allocated to any processor.

If not, that entity is allocated to a processor using the heuristic ρ as explained in the earlier section. Then the bead is scheduled as early as possible on the processor to which this entity has been allocated. RV's are also constructed for each of the scheduled beads. Note that the amortized cost of calculating $\rho[p]$ for all beads is *O(Number of beads in bead precedence graph).*

4. Simulation Studies

To evaluate the performance of the three algorithms for object-based tasks we conducted extensive simulation studies. The performance metric used is the *guarantee ratio* defined as the ratio of number of tasks found schedulable by an algorithm to the number of tasks considered for scheduling. The parameters used in the simulation are given in Table 1. Each point in the performance curves (Figure 7) is the average of several simulation runs each with 100 object-based tasks with a 95% confidence level. The values indicated for the parameters are used in all the following graphs unless otherwise stated.

The object-based tasks were generated as follows from the above parameters. Each object-based task generated consists of a method which calls other methods belonging to other objects and so on. *MToMRatio* is used to determine the sharing of methods. The actual number of methods in the system is chosen as $MToMRatio \times$ *Total number of method invocations in the task graph of task sets arriving at a time.* *SofRatio* is used to determine the sharing of software components by chosing the number of software components to be $SofRatio \times$ *Total number of method invocations in the task graph of MaxTask task sets arriving at a time.* For a given number of methods in the object-based task graph, the more the value of $SofRatio$ the more is the number of software components (objects) and the lesser is the contention among the methods for accessing the objects.

Each bead is chosen to have a computation time uniformly distributed between *MinBeadCompTime* and *MaxBeadCompTime*. The read time of a bead is the product of *CCRatio*, *RdRatio* and a number chosen uniformly between *MinBeadCompTime* and *MaxBeadCompTime*, The write time of a bead is the product of a uniform number between *MinBeadCompTime* and *MaxBeadCompTime*, *CCRatio* and *WrRatio*. In addition, the cost due to reclaiming is added as ($RecCost \times NumProcs$) to the total bead computation time. The resource requirements of a bead are determined by *UseP* and *ShareP*. The probability of a bead failing at run-time is determined by *FaultProb*. The actual execution times (both computation and output) of a bead at run-time are determined using a multiplicative factor chosen uniformly between *min_aw_ratio* and *max_aw_ratio*.

EnvProb is the probability of an object being environment dependent or stateful and hence not clonable. The deadlines of the tasks are chosen using a laxity lying uniformly between *min_laxity* and *max_laxity*. An average of *MaxTask* tasks arrive

1 On receiving a new set of tasks, compute the
   $cutoffline = currenttime + schedulingtime$.
2 Include the beads which start after the $cutoffline$ among the beads to be scheduled.
3 Make a depth-first search of the Initial Bead Precedence Graph, turning it into the Bead Precedence Graph by making modifications to it as described in section 3.2.1). Also, for each bead,
   calculate its worst case execution time by calculating the worst case communication times according to
   (a) the amount of output required at the beginning and the end of every bead,
   (b) whether they are fault-tolerant or not and depending on the fault-tolerant algorithm being used,
      and adding it to the bead's worst case computation time.
4 Make a depth-first search of the Bead Precedence Graph. For each bead,
   (a) Create a new entity corresponding to that bead (if not already created)
   (b) If the bead has a communication with one or more beads (read/write), update $CNADV(i, j)$, where E_i is the entity this bead belongs to and E_j is the entity the bead which communicates with this bead belongs to.
5 Make a bottom-up pass through the Bead Precedence Graph to obtain the individual bead deadlines. This step can also be combined with the next step.
6 Order the beads to be scheduled in the Bead Queue(BQ) in non-decreasing order of their deadlines.
7 Repeat until all beads are scheduled or no schedule is possible
   (a) Calculate the H heuristic for the first K beads in the BQ.
   (b) Select the bead with the least H-value.
   (c) If the entity to which that bead belongs has not been allocated to a processor
      Find the allocation heuristic ρ of the bead's entity for each processor and allocate the entity to the processor with which it has the least heuristic value.
   (d) Schedule the bead as early as possible on the processor to which its entity has been allocated.
8 If (all beads have been scheduled before their deadlines) then
      at the cutoff time, put the newly scheduled beads onto the dispatch queues of the processors. Construct the RV's for all scheduled beads
   else
      At the cutoff time, put back the old scheduled beads into the dispatch queues.

Figure 6. Dynamic scheduling algorithm for object-based real-time tasks

at the scheduler at an average frequency of *TaskFreq* with exponential distribution.

NumProcs is the number of processors. K is the window lookahead in the myopic scheduling algorithm. *distance* is the distance used in the PS-EXCL algorithm. *ReplProb* is the probability of a method (or object) being chosen to be replicated. This might be offset by the *EnvProb*.

4.1. Effect of Number of Processors

Figure 7a shows the effect of varying *NumProcs* in the system from 2 to 16. All the three algorithms show an increasing guarantee ratio, which saturates and then begins to decline slowly. Two interseting aspects are worth noting in this graph. First, at low *NumProcs*, PS-EXCL performs better than CONCUR, in fact, as well as OVERLAP. This is because the chances of (near) simulataneous scheduling of both the copies is very less. Second, at high *NumProcs*, CONCUR and OVERLAP show the same performance as a lot of processor space is available and hence most of the time in OVERLAP scheme, both copies of a given bead get scheduled and run simulataneosuly. In short, OVERLAP reduces to CONCUR.

4.2. Effect of Fault Probability

Figure 7b shows the effect of varying *FaultProb* in the system from 0.0 to 1.0. CONCUR's guarantee ratio does not vary with *FaultProb* as always both the copies of every bead are executed. PS-EXCL's guaranatee ratio falls with increasing *FaultProb* as lesser procesor time can be reclaimed and most of the backup copies execute as the primary copies fail.

4.3. Effect of Communication to Computation Ratio

Figure 7c shows the effect of varying *CCRatio* in the system from 0.2 to 1.0. As the *CCRatio* rises, the amount of communication (output) load in the system rises and all the three algorithms show a decreasing guarantee ratio. At high *CCRatio* values, the load in PS-EXCL becomes lesser compared to OVERLAP and CONCUR as the worst case execution time of a fault tolerant bead copy is $(r(B) + c(B) + w(B))$ in PS-EXCL and $(r(B) + c(B) + 2 \times w(B))$ in CONCUR and OVERLAP. Hence, PS-EXCL begins to overtake CONCUR in performance at a $CCRatio = 0.8$.

4.4. Effect of Environment/Stateful Probability

Figure 7d shows the effect of varying *EnvProb* in the system from 0.0 to 1.0. The readings shown are for values $ReplProb = 0.0$ and $TaskFreq = 300$. The aim is to see the performance of the dynamic scheduling algorithm as more software components become stateful or environment dependent. As all the three algorithms reduce to just the dynamic scheduling algorithm (without any fault-tolerance) and

show the same performance, only one graph is shown. As expected, guarantee ratio falls with increasing statefulness and dependence of software components on the environment because cloning becomes lesser. This graph thus shows that cloning leads to better schedulability.

Table 1. Simulation parameters

Parameter	Explanation	Values used
MaxCompTime	Bead's maximum worst case computation time	50
MinCompTime	Bead's minimum worst case computation time	30
CCRatio	Communication to computation ratio	0.2
WrRatio	Write ratio	1.0
RdRatio	Read ratio	1.0
UseP	Probability of a bead using a given resource	0.5
ShareP	Probability of a bead using a given resource in SHARED mode	0.5
RecCost	Cost of RV algorithm per processor	1.0
min_laxity	Minimum laxity	1.3
max_laxity	Maximum laxity	1.5
FaultProb	Probability of a bead encountering a fault at run-time	0.3
min_aw_ratio	Minimum aw_ratio	0.6
max_aw_ratio	Maximum aw_ratio	0.65
EnvProb	Prob. of an object being environment dependent or stateful	0.5
MToMRatio	Determines the sharing of methods	0.7
SofRatio	Determines the sharing software components	0.5
ArpcProb	Probability of a method call being an ARPC	0.5
NumProcs	Number of processors	6
MaxTask	Average number of tasks arriving at the scheduler at one time	2
TaskFreq	Average period of a task arrival at scheduler	225
K	Window size in the myopic algorithm	4
distance	Distance factor used in PS-EXCL	4
ReplProb	Prob. of an entity chosen to be replicated (fault-tolerant)	0.5
NumResources	Number of global resources in the system	5
NumResQty	Number of instances of each resource	3

4.5. Effect of Software Ratio

Figure 7e shows the effect of varying *SofRatio* in the system from 0.0 to 1.0. The readings shown are for values $ReplProb = 0.0$ and $TaskFreq = 300$. The aim is to see the performance of the dynamic scheduling algorithm with varying contention for software components. Note that all the three algorithms show the same performance as they reduce to the dynamic scheduling algorithm. As expected, guarantee ratio rises with increasing *SofRatio* as the number of software components rises and thus the contention for accesing them decreases among the different methods, thus increasing schedulability.

4.6. *Effect of ARPC Parallelism*

Figure 7f shows the effect of varying *ArpcProb* in the system from 0.0 to 1.0. Performance of all the algorithms improves with improving ARPC parallelism.

4.7. *Effect of Read-Write Costs*

Figure 7g shows the effect of varying *RdRatio* in the system from 0.2 to 1.8. *WrRatio* is always tasken as $(2 - RdRatio)$ to keep the task-load constant. The aim of this experiment of to see the variation in the performance of the three algorithms when they are used in multiprocessor systems which vary in the way a global write or read is done. For example, a global write by a processor may involve a write to the memory of the processor to which the write is being done to and the global read will thus be just a read from the processor's local memory. This would correspond to high values of *WrRatio* and low values of *RdRatio* as a write would be costlier than a read. On the other hand, a global write may be done by a processor on to its own memory and a global read from the memory of the processor from which the read is being done. This would correspond to low values of *WrRatio* and high values of *RdRatio*. With increasing *RdRatio*, PS-EXCL performance does not vary as the communication time of a bead $(r(B) + w(B))$ remains the same. However, CONCUR and OVERLAP improve their guarantee ratios due to decreasing bead communication time ($r(B) + 2 \times w(B)$ per bead) and thus decreasing system load.

4.8. *Effect of Replication Probability*

Figure 7h shows the effect of varying *ReplProb* in the system from 0.0 to 1.0. As *ReplProb* rises, all three algorithms show decreasing guarantee ratio due to increasing load (as the number of fault-tolerant beads and thus the number of bead copies being rises). Note that all three algorithms show the same performance at $ReplProb = 0.0$ as they reduce to the simple dynamic scheduling algorithm. As *ReplProb* rises, OVERLAP starts performing better than CONCUR as it has greater flexibility in scheduling and running beads.

4.9. *Conclusions from the Simulation Studies*

The order of performance of the three PB approaches for the object-based model is OVERLAP $>$ CONCUR $>$ PS-EXCL. This is because the object-based task model is similar to the conventional task model discussed earlier but for tasks (here beads) having precedence constraints due to method calls and individual bead orders in methods.

With increasing communication ($CCRatio$), PS-EXCL tends to perform better than CONCUR as the amount of output in CONCUR and OVERLAP is twice that in PS-EXCL. More faults occuring in the system ($FaultProb$) lead to a drop in the performance of PS-EXCL while having no effect on CONCUR or OVERLAP. This

is because PS-EXCL benefits mainly from the resource time reclaimed from the secondary copies of beads not being executed; this decreases with increasing faults in the system.

The dynamic scheduling algorithm is able to utilize the advantage of increasing ARPC parallelism by cloning (Figure 7f). Increasing dependence of software components on the environment ($EnvProb$) reduce their clonability and thus the schedulability in the system. Figure 7e shows that the dynamic scheduling algorithm performs better if there is a lesser contention for software components among methods (increasing $SofRatio$).

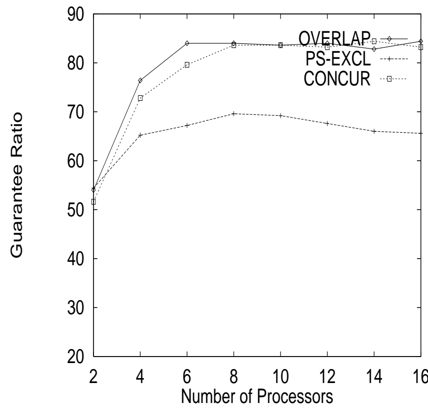

Figure 7a. Effect of NumProcs

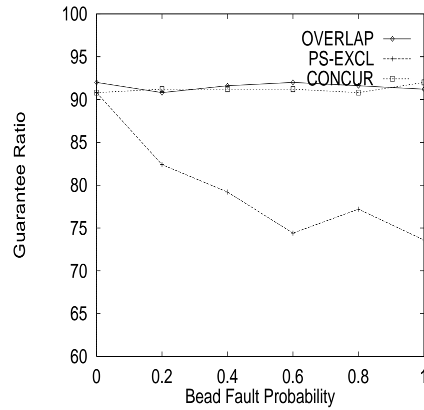

Figure 7b. Effect of FaultProb

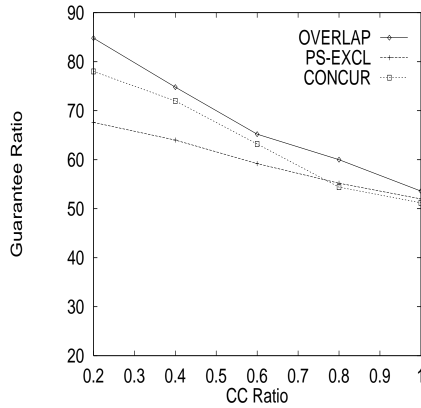

Figure 7c. Effect of CCRatio

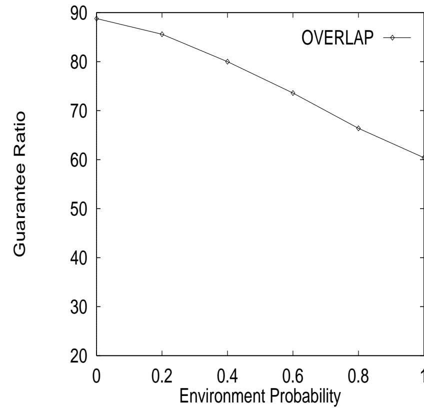

Figure 7d. Effect of EnvProb

Figure 7. Simulation results

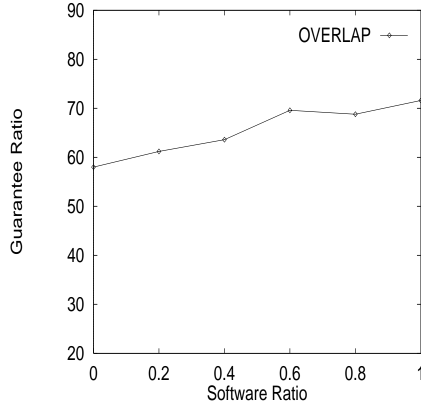

Figure 7e. Effect of SofRatio

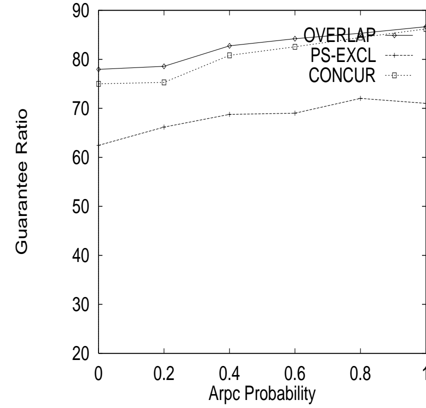

Figure 7f. Effect of ArpcProb

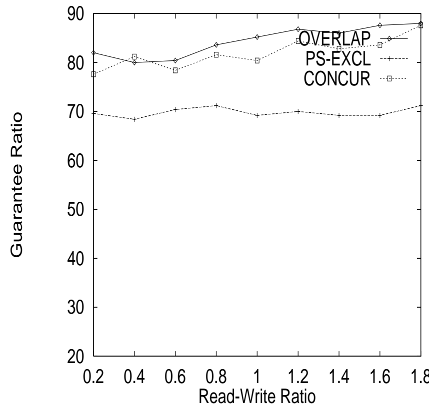

Figure 7g. Effect of RdRatio

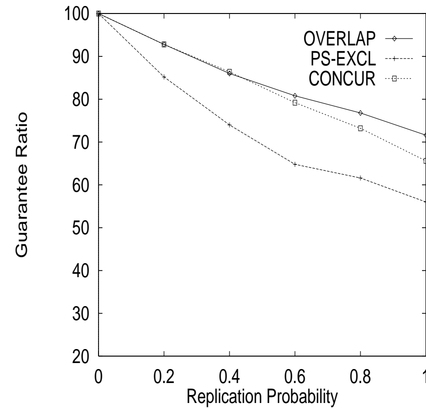

Figure 7h. Effect of ReplProb

5. Conclusions

In this paper, we have extended three different PB-based fault-tolerant approaches, namely, PS-EXCL, CONCUR, and OVERLAP, to object-based task model. We also proposed an algorithm for fault-tolerant dynamic scheduling of object-based real-time tasks. The proposed dynamic scheduling algorithm utilizes the parallelism due to cloning and ARPC. The implementation of CONCUR and OVERLAP is harder than PS-EXCL, as each set of output actions (of a bead) have to be scheduled twice in the former two approaches and only once in the latter. We also compared the performance of three PB-based fault-tolerant approaches for object based task models in the context of dynamic scheduling. From the experiments, the following observations are made:

- *In general*, the PS-EXCL approach works better than the CONCUR approach for precedence-free tasks, but the order slowly reverses as precedence constraints among the tasks (beads) increase. The OVERLAP approach scores over both PS-EXCL and CONCUR in both conventional and object-based task models.
- When the number of resource instances is low or resource constraints among beads is high, CONCUR's performance deteriorates.
- As the laxity of tasks (beads) increases, the performance difference between PS-EXCL and CONCUR widens.
- As the fault probability rises, the performance of PS-EXCL falls more steeply than that of OVERLAP. The performance of CONCUR remains the same irrespective of the fault probability.

Currently, we are working on integrating different fault-tolerant techniques (TMR, PB, and IC) with adaptive [3] selection of these in the object-based task model.

Notes

1. The IC and (m, k)-firm models were originally proposed for overload handling.
2. The general case of this is known as Recovery Blocks [17] where each task has many versions.

References

1. L. Chen and A. Avizienis, "N-version programming: A fault tolerance approach to reliability of software operation," In *Proc. IEEE Fault-Tolerant Computing Symp.,* pp.3-9, 1978.
2. S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Trans. Parallel and Distributed Systems,* vol.8, no.3, pp.272-183, Mar. 1997.
3. O. Gonzalez, H. Shrikumar, J.A. Stankovic, and K. Ramamritham, "Adaptive fault-tolerance and graceful degradation under dynamic hard real-time scheduling," In *Proc. IEEE Real-Time System Symp.,* 1997.
4. K. Kim and J. Yoon, "Approaches to implementation of reparable distributed recovery block scheme," In Proc. *IEEE Fault-Tolerant Computing Symp.,* pp.50-55, 1988.
5. K.H. Kim and H. O.Welch, "Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults in real-time applications," *IEEE Trans. Computers,* vol.38, no.5, May 1989.
6. K.H. Kim and A. Damm, "Fault-tolerance approaches in two experimental real-time systems," In *Proc. Workshop on Real-Time Operating Systems and Software,* pp.94-98, May 1990.
7. K.H. Kim and C. Subbaraman, "Fault tolerant real-time objects," *Commun. of the ACM,* vol.40, no.1, pp.75-82, Jan. 1997.
8. J.W.S. Liu, W.K. Shih, K.J. Lin, R. Bettati, and J.Y. Chung, "Imprecise computations," *Proc. of IEEE,* vol.82, no.1, pp.83-94, Jan. 1994.
9. K. Mahesh, G. Manimaran, C. Siva Ram Murthy, and A.K. Somani, "Scheduling algorithms with fault detection and location capabilities for real-time multiprocessor systems," *J. Parallel and Distributed Computing,* vol.51, no.2, pp.136-150, June 1998.
10. G. Manimaran, C. Siva Ram Murthy, Machiraju Vijay, and K. Ramamritham, "New algorithms for resource reclaiming from precedence constrained tasks in multiprocessor real-time systems," *J. Parallel and Distributed Computing,* vol.44, no.2, pp.123-132, Aug. 1997.

11. G. Manimaran and C. Siva Ram Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems," *IEEE Trans. Parallel and Distributed Systems,* vol.9, no.3, pp.312-319, Mar. 1998.
12. G. Manimaran and C. Siva Ram Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Trans. Parallel and Distributed Real-time Systems,* vol.9, no.11, pp.1137-1152, Nov. 1998.
13. J.H. Purtilo and P. Jalote, "An environment for developing fault-tolerant software," *IEEE Trans. Software Engg.,* vol.17, no.2, pp.153-159, Feb. 1991.
14. K. Ramamritham, J.A. Stankovic, and P.-F. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Trans. Parallel and Distributed Systems,* vol.1, no.2, pp.184-194, Apr. 1990.
15. K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. of IEEE,* vol.82, no.1, pp.55-67, Jan. 1994.
16. P. Ramanathan, "Graceful degradation in real-time control applications using (m,k)-firm guarantee," In Proc. *IEEE Fault-Tolerant Computing Symp.,* pp.132-141, 1997.
17. B. Randell, "System structure for software fault-tolerance", *IEEE Trans. Software Engg.*, vol.1, no.2, pp.220-232, June 1975.
18. C. Shen, K. Ramamritham, and J.A. Stankovic, "Resource reclaiming in multiprocessor real-time systems," *IEEE Trans. Parallel and Distributed Systems,* vol.4, no.4, pp.382-397, Apr. 1993.
19. J.A. Stankovic and K. Ramamritham, "The Spring Kernel: A new paradigm for real-time operating systems," *ACM SIGOPS, Operating Systems Review,* vol.23, no.3, pp.54-71, July 1989.
20. T. Tsuchiya, Y. Kakuda, and T. Kikuno, "Fault-tolerant scheduling algorithm for distributed real-time systems," In Proc. *Workshop on Parallel and Distributed Real-time Systems,* 1995.
21. J.P.C. Verhoosel, D.K. Hammer, E.Y. Luit, L.R. Welch, and A.D. Stoyenko, "A model for scheduling object-based distributed systems", *J. Real-Time Systems*, vol. 8, no. 1. pp 5-34, January 1995.
22. I. Santoshkumar, G. Manimaran, and C. Siva Ram Murthy, "A pre-run-time scheduling algorithm for object-based distributed real-time systems", *Proc. 5th IEEE Joint Workshop on Parallel and Distributed Real-Time Systems, April 1-3 1997*, pp. 160-167.
23. A. D. Stoyenko, L. R. Welch, J. P. C. Verhoosel, D. K. Hammer, and E. Y. Luit, "A model for scheduling of object-based, distributed real-time systems," *J. Real-Time Systems*, vol. 8, pp. 5-34, August 1995.
24. G. Yu, *Identifying and exploiting concurrency in object-based real-time systems,* Ph.D. Thesis, New Jersey Institute of Technology, January 1996.
25. L. R. Welch, "Assignment of ADT modules to processors," In *Proc. IEEE Int. Parallel Processing Symp.*, pp. 72-75, March 1992,
26. J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, and E. J. Luit, "Incorporating temporal considerations during assignment and pre-run-time scheduling of objects and processes," *J. Parallel and Distributed Computing*, vol. 36, no. 1, pp. 13-31, July 1996.
27. M. Joseph, *Real Time Systems: Specification, Verification and Analysis.*, Prentice Hall International Series, 1996.