Dynamic Replication Decisions in Fault-Tolerant Multiprocessor-Based Real-Time Systems

Monika Andersson Wiklund and Jan Jonsson

Department of Computer Engineering
Chalmers University of Technology
S–412 96 Göteborg, Sweden
monikaw,janjo@ce.chalmers.se

Abstract

Fault-tolerance is an important property for many critical real-time applications such as telecommunication servers or space-borne signal-processing computers, both for static and dynamically arriving tasks. In this paper, we describe two method for supporting fault-tolerance for dynamically arriving tasks by replicating the tasks and scheduling the copies of each critical task. The methods are varieties of the Primary/Backup method for providing fault-tolerance and the idea is to dynamically decide whether the replication should be temporal or spatial given the execution time and deadline of the task and the load in the system. This offers a flexibility that improves the schedulability of the dynamically arriving tasks.

1 Introduction

Multiprocessor systems available today are used for new high-performance real-time applications, such as telecommunication servers or space-borne signal-processing computers. These systems work in hostile environments where it is not only important that data is processed correctly before a deadline, but also necessary to maximize the successful completion of tasks even in the presence of faults. One way of doing this is to replicate the tasks so that a copy of the task can be completed even if the original task fails. Depending on how the failure of a task is detected, there have to be a different number of copies of each original task.

The most commonly-used method to provide fault-tolerance is the Primary/Backup method where there are two copies of each task, one primary and one backup copy. In most cases, it is assumed that there is a function which detects any failures of a copy when the execution is completed. Since the function tells us when a copy fails and which copy it is that fails, it is possible to deallocate the resources used by the backup copy if the primary copy executes successfully.

However, in many systems it is not always possible to implement such a function, which means that the fault detection must be made in some other way. One method for detecting faults is then to execute multiple copies and compare the results generated by the copies; unfortunately, it is impossible to know which copy failed unless an odd number of copies are used. In some systems it is also sometimes possible to detect a fault before the task is completed because of special hardware mechanisms, for example signature checking. In this case it is possible to remove the copy and start a new execution immediately when the fault is detected.

In all of the above methods, replication is used to achieve fault-tolerance. The difference between the methods is only the number of replicas and how the faults are detected in each method. In both cases the replication can be temporal, spatial, or a combination of both. Which replication method is the best depends on the fault model, the purpose of the replication, and the properties of the system. If both transient, intermittent and permanent faults must be handled, the replication have to be spatial, or both temporal and spatial; but, if only transient faults are handled it is sufficient to use temporal replication. In this work we only consider transient or intermittent faults and thus we use both temporal and spatial replication.

To accept as many dynamically-arriving tasks as possible, it is necessary to execute as few copies as possible. This can be achieved by deallocating the backup copy if the original copy completes its execution successfully. The deallocation releases resources that will not be needed for the successful task and makes it possible to use these resources for any dynamically arriving task that enters the system. However, this deallocation and reuse of resources is only possible when the replication is either temporal or both temporal and spatial, which implicates that it is desirable to use temporal replication whenever it is possible.

In order to achieve fault tolerance for dynamically arriving tasks and to be able to guarantee as many new tasks as possible in the schedule we need to decide which is the best possible method for replicating each new task. One way to do this is to examine the deadline and execution time of the task and, if the deadline is long enough, use temporal replication, otherwise, spatial replication is used. However, the load in the system and the fault model must also be considered while making such decisions.

In this paper, we propose two methods for dynamic replication of dynamically arriving tasks on a multiprocessor system. In the first method a function is used for fault detection and all faults are detected at the end of the execution, while in the second method multiple copies are used for fault detection and faults can be detected at any time during the execution. Both methods assume that we have a known baseload containing periodic tasks with real-time guarantees, and the tasks that are handled by the method are aperiodic (e.g., telecommunication packets for GSM) and there is no information available for these tasks until they arrive in the system.

To verify and test the methods we use extensive simulation studies. The results from a preliminary study are promising, and verifies the functionality of the proposed methods.

The remainder of this paper is organized as follows: In Section 2 we discuss related work in the area of fault-tolerant real-time systems and replication. Section 3 describes the fault models while Section 4 present the proposed replication methods. The implementation and simulation of the methods are discussed in Section 5. Finally, in Section 6 we summarize our results and discuss future work.

2 Related work

Hou and Shin [1] present a technique where a replication decision is made dynamically off-line by determining whether or not the task module should be replicated in a system with limited resources. Since the resources are limited, only those modules which have deadlines too tight for time redundancy should be replicated in space. The backup copy and the original copy of the task may not run on the same processor and the modules are allocated with an off-line Branch&Bound algorithm. The main objective of this method is to replicate only those modules that have to be replicated, not to find the best possible replication method.

González et al [2] offer a method that uses alternative lists for choosing which method of fault-tolerance should be used. Each arriving task has a list that contains up to three methods, TMR, Primary/Backup, and Primary/Exception. The methods are considered in the order they appear in the list and the scheduler tries to schedule the tasks with the different methods until a feasible schedule is found or the list is empty. With this approach the choice of fault-tolerance is made off-line since the lists must be prepared in advance. This means that the system load and load distribution have no influence over the replication decision.

Mossé, Melhem, and Ghosh [3, 4] present the Primary/Backup principle of replication for independent, dynamically arriving tasks. Backup overloading and resource deallocation is used to achieve high schedulability and the primary copy of a task is always scheduled as early as possible while the backup is scheduled as late as possible but with as much overloading[1] as possible. There are no dynamic decisions in this method since all backups are scheduled after the primary and on another processor than the primary.

The Distance Myopic algorithm [5, 6] also uses the Primary/Backup method. By computing the relative distance in position between the primary and backup copies in the dispatch queue, a flexible level of backup overloading gives a trade-off between number of faults that can be handled, and system performance. All dynamic tasks arrive centrally and are distributed to a processor via the dispatch queues and each processor then executes the tasks in its dispatch queue. When a task is executed successfully, the processor invokes a resource reclaiming algorithm which allows unused resources to be rescheduled and the backup copy to be deallocated (unless it has to execute as well).

Fohler [7] presents a method to achieve adaptive fault-tolerance for distributed systems by first providing a basic reliability level and then divide the schedule into disjoint execution intervals and define the spare capacities for each interval. An on-line scheduler is invoked at the end of each interval and takes care of aperiodic tasks that arrived during the interval by using the spare capacities. After this the spare capacities are updated and the scheduling decision is executed in the next interval using EDF. It is assumed that only one task at a time can be affected by faults and the number of replicas can be increased during runtime if there are enough spare capacities.

Ahn, Kim, and Hong [8] use a variety of the Primary/Backup method where the primary copy of a task may overlap the backup copy of another task. To prevent the domino effect, i.e., that the activation of a backup task causes a primary task to miss its deadline the scheduling is made by forming checkpoint cycles. There must be at least two processors in the cycle that can finish the task currently executing and the newly activated task within their deadlines if a domino effect ahould be avoided. Each task is scheduled on the processor that gives the longest possible checkpoint chain until a cycle is formed, then the processor that satisfies the condition above and who gives the longest checkpoint cycle is chosen.

Srinivasan and Shoja [9] present a method in which periodic tasks with two versions, primary and alternate, are scheduled dynamically to maximize the number of scheduled primaries. The tasks are first scheduled statically and during the execution of the tasks the dynamic scheduler is run to enhance the number of primaries that can be executed. If a primary executes without failure, the alternate for this task is redundant and a, so far, unscheduled primary might be scheduled instead. To find a schedulable primary as quickly as possible, all nodes needs to contain information about the unscheduled primaries in the other nodes. Each node thus has two lists, the schedule to follow

[1] Overloading means that several backup copies are scheduled in the same time slot. This can only be done if the primary copies of all the backups are scheduled on different processors.

and a shortest job list that contains all the unscheduled primaries across the system. When a message from the shortest job list is executed, a message is sent to all nodes and each shortest jobs list is updated.

These methods differ from that of ours in the following ways. Several of the methods use information about tasks that have not yet entered the system, which is not always possible. Also, none of the methods decides dynamically how the replication of the dynamically arriving tasks should be done, which means that some task might be ruled unschedulable even though they might be schedulable if another replication method is used.

3 Fault Models

As in most techniques for fault-tolerant systems, the work in this paper assumes that the *single-fault assumption* is valid, that is, only one faulty result may be produced per replica set.

The faults may appear either in the hardware or in the software and they can be classified as being permanent, intermittent or transient.

Permanent faults in a hardware circuit are caused by, for example, electromigration[2], gate-oxide breakdown, and radiation, and can be detected in self-tests or by repeated execution. When a node can be suspected to contain a permanent fault it has to be stopped and tested, and, if it is faulty, exchanged or repaired. A malicious class of permanent faults that can not usually be detected or prevented by redundancy are design errors.

Transient faults are mostly caused by environmental effects such as radiation or cross-talk (electromagnetic interference) and these faults will not cause any remaining damage on the node. The faults can be detected either by hardware mechanisms, such as checksums or parity coding, or they can be detected by software mechanisms such as voting or integrity checks on the results from the computation.

Intermittent faults might be caused by electromagnetic interference that is caused by, for example radiation, a cellular phone, or a radar antenna which are periodically sending electromagnetic signals. Another possible cause for intermittent faults are undetected permanent faults that only become active from time to time. This means that intermittent faults really are special cases of either permanent or transient faults and can be detected in the same way.

In our methods, permanent faults can only be handled if the replication method operates in a way such that no replicas of the same task are scheduled on the same processing node, or that a cold spare takes over for the stopped node when a permanent fault is detected.

4 Our Dynamic Replication Methods

The replication methods used in our approach are based on the Primary/Backup method, that is, each dynamic task

[2]Movement of metal due to momentum exchange with electrons causing preferred direction of diffusion. [10]

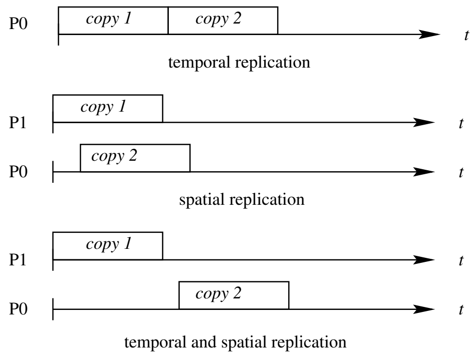

Figure 1: Replication varieties for two copies.

is replicated and have one primary copy which always executes and one or two backup copies that only execute if the primary copy was unable to finish its execution or if a fault is detected. In this work it is assumed that only transient or intermittent faults can occur; however, it is possible to adjust the method to allow for permanent failures as well.

Our first method for dynamic replication assumes that there exists a function which detects whether the task is successful or not when the execution is completed. Thus only one copy of each task is needed, the original and a backup. Our second method assumes that no such function exists, and we therefore use two copies of each task for voting; however, it is possible to find some faults before the execution is completed if the faults are detected by hardware mechanisms.

The way in which the scheduling is done means that sometimes the replication can be both temporal and spatial at the same time. That is, if the replication is temporal, but a backup copy cannot be scheduled on the same processor as the primary copy, then the replication will be both temporal and spatial. It is important to note that a backup copy always must execute after the primary copy if the replication is temporal, and that it may be partially or completely overlapped with the primary copy if the replication is spatial.

Also, by using a distributed scheduling algorithm [11] for scheduling, the replication strategy will depend on the load in the system. Unless we have to handle permanent failures, it is preferred that the primary and the backup copy should be scheduled on the same processor; however, if this is not possible due to high system load, we can allow the copies to be scheduled on different processors.

4.1 Method 1: Two copies of each task

The dynamic replication strategy using two copies is very simple. When a dynamic task arrives in the system, the deadline and execution time of the task is examined. If the difference between the deadline (D_i) and the execution time (C_i) multiplied with a certain replication breakpoint, s, is larger than the task execution time of a potential backup task, then the task is replicated in time; otherwise it is replicated in space. Depending on the load in the system, the replication can also be a combination of both (see Figure 1).

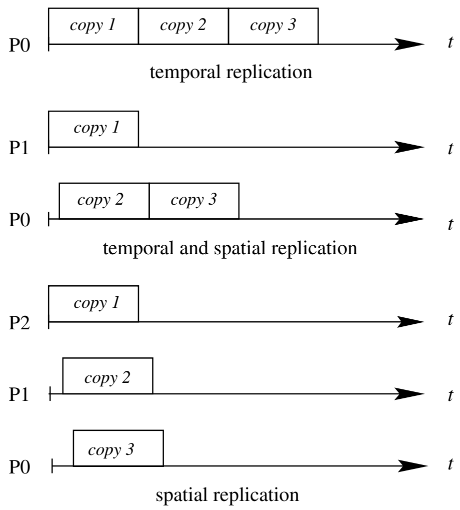

Figure 2: Replication varieties for three copies.

The reason for having the replication breakpoint parameter is that there might be extra cost involved in the execution, for example, a communication cost may be involved if the primary and backup copies are situated on different processors.

$$D_i - C_i \times s \geq C_i \rightarrow \textit{replicate in time}$$

When the task has been replicated, the two copies of the task are scheduled using a distributed scheduling technique such as the Random scheduling algorithm or the Flexible algorithm [11]. First, the primary copy is scheduled on the processor it arrived to, or if it is not possible to schedule it on this processor, it is sent to one of the other processors. Which other processor is chosen to send the copy to depends on which scheduling technique is used; a processor can either be chosen randomly or on the basis of the load in the system.

If the replication is temporal, the method attempts to schedule a backup copy on the same processor as the primary copy. If this is not possible, a processor is chosen accordingly to the scheduling technique and the method attempts to schedule the copy on this processor. For spatial replication, on the other hand, the backup copy may not be scheduled on the same processor as the primary, that is, another processor must be chosen before the scheduling attempt. Both the primary and the backup copies must be scheduled for the task to be accepted; if either copy cannot be scheduled the task is rejected.

The method can be altered to handle permanent failures by scheduling all copies on different processors whenever possible.

4.2 Method 2: Three copies of each task

In this method we need at least two, but sometimes three, copies of each task since we assume that there is no function for detecting faults, that is, there is no integrity check for the result of the computation. We assume that at most one copy of each task can be faulty, and thus three copies of each task is enough for fault detection. At first, two copies are executed and their results are compared; if the results match, the task is successful and the third copy is unnecessary. If the results do not match, the third copy must be executed and compared with the first results so that the correct result can be identified.

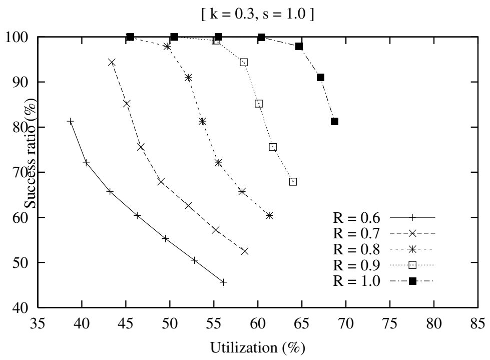

Figure 3: The influence of deadline ratio, R.

Since the third copy of each task only is needed when a fault has been detected, it is desirable that this copy is scheduled after the other copies so that it can be deallocated in case it will not be needed. There are three possible combinations for replicating the task: (i) all copies are temporal replicas, (ii) two copies are spatial replicas and one is temporal, or (iii) all copies are spatial replicas (see Figure 2). Which combination is used depends on the deadline of the task in the following way. If the deadline is long enough to allow all copies to execute after each other within the deadline, only temporal replication is used. If the deadline is too short for two copies to execute after each other, only spatial replication is used; otherwise, one spatial and one temporal replication is used.

To avoid that certain types of faults affect more than one copy in the same way, it is possible to place all copies on different processors and make sure that they do not start at exactly the same time. This means that we will mix the temporal and spatial properties of the replicated copies so that all copies will differ in both time and space.

5 Implementation and Simulation

We have implemented our methods in GAST [12] (Generic Allocation and Scheduling Tool), a software package developed at Chalmers University of Technology. The implementation of the proposed methods has been performed in steps. Initially, a simple version of the first method without resource deallocation was implemented and later resource deallocation was added. The next step was to implement a simple version of the second method; this implementation has just been finished.

When the simple method of the first version was implemented, we ran a small simulation study to verify that it behaved as expected. Later, we will add the results of newer simulation studies on the full implementation.

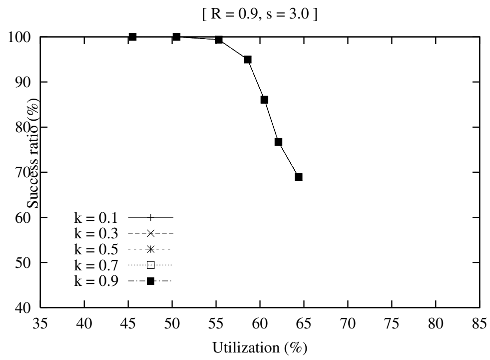

Figure 4: The influence fault intensity, k.

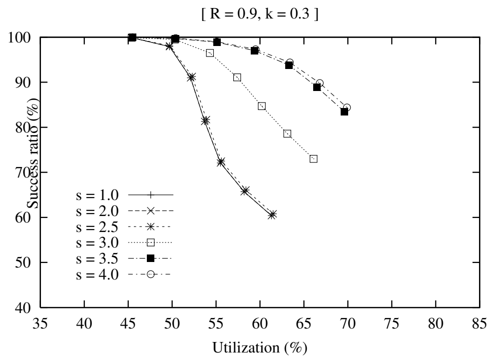

Figure 5: The influence of replication breakpoint, s.

For the small simulation study, we used a system with four nodes, a baseload of periodic tasks with periods of 100 time units and dynamically arriving tasks with a minimal inter-arrival time of 75 time units. Each task had an execution time that was randomly chosen from a normal distribution with a mean execution time of 15 time units and a variance of 1. The influence of three different parameters was examined by running 128 simulations for each setting. The parameters were; the deadline/period ratio (R), the fault intensity (k), and the replication breakpoint (s). The mean execution time was increased in steps of 5 from a start value of 5 to a final value of 45 for each combination of the parameters. We use a non-preemptive, time-driven execution model and we use success ratio as a measure of performance in our simulations. We define the success ratio as the fraction of tasks that can be scheduled with our method out of all dynamically arriving tasks.

The effect of the deadline/period ratio is shown in Figure 3. It is clear from the plots that if the deadline is shortened (that is, the ratio R is decreased) the success ratio is also decreased.

In Figure 4, the effect of the fault intensity is shown. As can be seen there is no effect on the success ratio when the fault intensity is increased. This is due the fact that we do not use resource deallocation in this simulation study. Because of this, backup copies that are not needed cannot be removed from the schedule and, thus, it does not matter to the success ratio how large the fault intensity is. If we use resource deallocation, however, there should be a difference in success ratio if the the fault intensity k is varied.

The effect of the replication breakpoints is shown in Figure 5. Recall that, when the replication breakpoint is low, only spatial replication is used and when it is high only temporal replication is used. The figure shows that the success ratio decreases when the replication breakpoint is increased. The reason for this is that, for shorter deadlines, it is harder to find room for temporal copies than for spatial copies because the amount of free time between the baseload tasks is too small.

6 Summary and Future Work

We have presented two methods for dynamic replication of dynamically arriving tasks in a real-time multiprocessor system. The methods are varieties of the Primary/Backup method but have been extended to use the most suitable replication mode for each arriving task based on task characteristics and system load. The two methods are very similar except for how the fault detection is made.

The implementation of the first method is finished and verified by a small simulation study, and a simple implementation of the second method have just been finished. We will perform a more extensive simulation study to find which parameters are most influential on the success ratio for the methods. We will also investigate how much it helps to use resource deallocation. Finally, we will make a full implementation of the second method and perform further simulation studies to see if there are any differences between the full implementation and our simpler version.

References

[1] C.-J. Hou and K. G. Shin, "Replication and Allocation of Task Modules in Distributed Real-Time Systems," *Proc. of the IEEE 24th International Symposium on Fault- Tolerant Computing*, Austin, TX, June 15–17, 1994, pp. 26–35.

[2] O. Gonzlez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham, "Adaptive Fault Tolerance and Graceful Degradation Under Dynamic Hard Real-time Scheduling," *Proc. of the 18th IEEE Real-Time Systems Symposium*, San Francisco, CA, Dec. 2–5, 1997, pp. 79–89.

[3] D. Moss, R. Melhem, and S. Ghosh, "Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm," *Proc. of the IEEE 24th International Symposium on Fault- Tolerant Computing*, Austin, TX, June 15–17, 1994, pp. 16–25.

[4] S. Ghosh, R. Melhem, and D. Moss, "Fault-Tolerant scheduling on a Hard Real-Time Multiprocessor System," *Proc. of 8th International Parallel Processing Symposium*, Cancun, Mexico, Apr. 26–29, 1994, pp. 775–782.

[5] G. Manimaran and S. R. Murthy, "A Fault Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis," *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1137–1152, Nov. 1998.

[6] G. Manimaran and S. R. Murthy, "A new Study for Fault-Tolerant Real-Time Dynamic Scheduling Algorithms," *Journal of Systems Architecture*, vol. 45, no. 1, pp. 1–13, Oct. 1998.

[7] G. Fohler, "Adaptive Fault-Tolerance with Statically Scheduled Real-Time Systems," *Proc. of the Ninth Euromicro Workshop on Real Time Systems*, Toledo, Spain, June 11–13, 1997, pp. 161–167.

[8] K. Ahn, J. Kim, and S. Hong, "Fault-Tolerant Real-Time Scheduling using Passive Replicas," *Proc. on the Pacific Rim International Symposium on Fault-Tolerant Systems*, Taipei, Taiwan, Dec. 15–16, 1997, pp. 98–103.

[9] A. Srinivasan and G. C. Shoja, "A Fault-Tolerant Dynamic Scheduler for Distributed Hard-Real-Time Systems," *Proc. on the Pacific Rim Conference on Communications Computers and Signal Processing*, Victoria, BC, Canada, May 19–21, 1993, pp. 268–271.

[10] C. Hawkins, "Field Returns: Reliability Failure Mechanisms and Defect Electronic Test Properties," *Tutorial B, IEEE European Test Workshop*, Sitges, Spain, May 26, 1998, pp. XX–XX.

[11] K. Ramamritham and J. A. Stankovic, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Trans. on Computers*, vol. 38, no. 8, pp. 1110–1123, Aug. 1989.

[12] J. Jonsson, "GAST: A Flexible and Extensible Tool for Evaluating Multiprocessor Assignment and Scheduling Techniques," *Proc. on the Int'l Conf. on Parallel Processing*, Minneapolis, Minnseota, Aug. 10–14, 1998, pp. 441–450.