A Survey of Various Algorithms to Achieve Fault Tolerance in Wireless Sensor Networks

1S.Puvaneswari, 2S.hemalatha, 3B.Sangeetha
1,2,3 Asst. Prof/CSE
Kings College of Engineering

Abstract: **We survey various algorithms for tolerating permanent and transient failures in Wireless Sensor Networks. These algorithms attempt to provide low-cost solutions to fault tolerance, graceful performance degradation, and load shedding in such systems by exploiting tradeoffs between space and/or time redundancy, timing accuracy, and quality of service. Here we describe various algorithms which are used to achieve fault-tolerant and increase the performance of a system. The algorithms are dynamic scheduling, off-line or static scheduling, and scheduling, a technique which is used the concepts of mathematical optimization to allocate tasks on the processors and derive fault tolerant and fault aware feasibility and diskless check pointing approach.**

Keywords: **Fault tolerance, Wireless Sensor Networks**

1. INTRODUCTION

The correctness of real-time safety-critical systems depends not only on the results of computations, but also on the time instants at which these results become available. Examples of such systems include fly- and drive-by-wire, industrial process control, nuclear reactor management, and medical electronics. Real-time tasks have to be mapped to processors such that deadlines, response times, and similar performance requirements are met, a process called *task scheduling*. Furthermore, many real-time systems function in a hostile, unpredictable environment and have to guarantee functional and timing correctness even in the presence of hardware and software faults.

Faults can be classified according to their duration: Permanent faults remain in existence indefinitely if no corrective action is taken. These faults can be caused by catastrophic system failures such as processor failures, communication medium cutoff, and so on. Intermittent faults appear, disappear, and reappear repeatedly. They are difficult to predict, but their effects are highly correlated. Most intermittent faults are due to marginal design or manufacturing. Transient faults appear and disappear quickly, and are not correlated with each other.

In real-time systems, fault tolerance is typically provided by physical and/or temporal redundancy. Physical redundancy in the form of replicated hardware and software components is used to tolerate both permanent and transient system failures. To reduce the overhead associated with replicated hardware, some approaches treat the set of processors as a pooled resource. When a processor fails, other members in the pool provide the functionality of the failed processor. Though this approach lowers the hardware overhead needed to tolerate failures, it typically causes some performance degradation and non-zero recovery latency. A common recovery technique is re-executing the failed task. Another is the primary/backup approach wherein if incorrect results are provided by the primary version of a task, the backup (alternate) is executed.

2. DYNAMIC & STATIC SCHEDULING

A mapping of tasks to processors such that all tasks meet their time constraints is called a *feasible* schedule. A schedule is *optimal* if it minimizes a cost function defined for the task set. If no cost function is defined and the only concern is to obtain a feasible schedule, then scheduling is optimal only if it fails to meet a task deadline when no other algorithms in its class can meet it.

A *dynamic* scheduler makes its scheduling decisions at run time based on requests for system services. After the occurrence of a significant event such as a service request, the algorithm determines which of the set of ready tasks should be executed next based on some task priority which is statically or dynamically assigned.

A *static* or *off-line* scheduling algorithm considers the resource, precedence, and synchronization requirements of all tasks in the system and attempts to generate a feasible schedule that is guaranteed to meet the timing constraints of all tasks. The schedule is calculated off-line and is fixed for the life of the system. Typically, a scheduling or dispatch table identifies the start and finish times of each task, and tasks are executed on the processor according to this

table. Static table-driven scheduling is applicable to periodic tasks or to aperiodic (sporadic) tasks that can be transformed into periodic ones.

3. FAULT TOLERANT DYNAMIC SCHDULING ALGORITHM

If an aperiodic task Tj's execution cannot be guaranteed by a processor in a distributed system, the task is transferred to a processor estimated to have sufficient resources and time to complete the task before its deadline. Tjs transfer can also be based on bids received from lightly loaded processors and sent to the processor deemed most likely to execute the task within the deadline.

A simple fault-tolerant scheduling approach is to schedule the entire task set, that is, both primaries and backups. In the fault-tolerant scheduling approach proposed in, tasks are assigned "levels" based on their periods as follows. Let all tasks with period *p* be assigned level *i*. Then tasks in level *i* + 1 have period *m* ´ *p* for some positive integer *m* ³ 2. In Fig. 1(a), tasks with period 15 ms belong to level 1 and tasks with period 30 ms and 60 ms belong to levels 2 and 3, respectively.

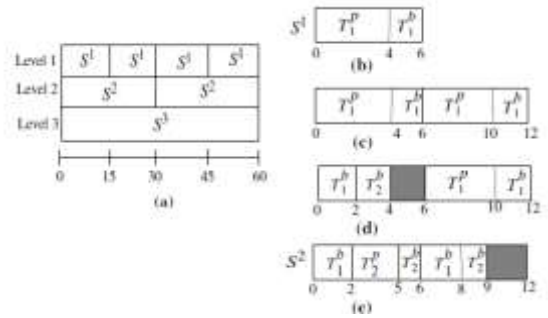

Fig1.(a) Level assigned based on their periods

First, backups of all level-1 tasks are scheduled. Then we schedule the maximum number of level-1 primaries that fit in the remaining time, thus ensuring that a backup is not scheduled earlier that its corresponding primary. The schedule for level-1 tasks is *S*1. Two *S*1 schedules are concatenated to get a provisional schedule *S*2, which is then modified by removing the minimum number of level-1 primaries such that all level-2 backups are scheduled. If *S*2 has enough idle time, level-2 primaries with least execution times are also scheduled. If any unscheduled level-2 primary has a lower execution time than any scheduled level-1 primary in *S*2, the level-1 primary with the largest execution time is dropped and replaced in *S*2 with the level-2 primary. Once *S*2 is constructed, two *S*2 schedules are concatenated to get *S*3, and so on. This algorithm schedules a primary and a backup or a backup for each periodic task in the system.

4. IMPRECISE ALGORITHM

Intermediate or partial results from task computations can be used instead of more precise final results when a real-time system suffers failures or transient overloads. Real-time application areas for imprecise computations include signal processing, machine vision, and linear control systems. In the *milestone* approach, partial results are obtained at different execution points in a computation and if a deadline is reached, the last recorded values form the task output. This method assumes that the precision of the results increases monotonically with time, that is, the longer a computation executes the more precise its results become. Milestones are specified using programming primitives, thereby allowing system designers to explicitly save partial results of selected program variables. The *sieve* approach is based on iterative functions where each iteration computes a closer approximation of the final answer. A well-known example is the Newton-Raphson method for finding the roots of a polynomial. If a deadline is close, such functions can skip one or more iterations to produce a result within the time limit, and the final result is not catastrophic to system correctness. The sieve approach implicitly specifies the imprecise results that can be obtained. A real-time task can integrate both the milestone and sieve approaches as follows. A task Tjs mandatory portion can use the milestone approach to produce acceptable results, while its optional portion can use the sieve approach to improve the results. The execution of depends on the availability of computing resources and time constraints.

Imprecise scheduling can handle transient overload in dynamic real time systems via a queuing theoretic approach. When the load is normal, the system computes precise results, that is, both the mandatory and optional portions of all tasks are executed. An unexpected external event can generate sporadic and aperiodic task requests, thereby increasing system load. In such cases, some tasks generate imprecise results by executing only their mandatory portions to ensure timing correctness of the system. For each task *Tj*, an on-line algorithm decides its computational level, that is, the algorithm directs *Tj* to produce precise or imprecise results depending on the current system load. After deciding *Tj*s computational level, the system is notified about the precision of the computation.

5. MIXED CRITICALITY SCHEDULING ALGORITHM

A mixed criticality real-time system typically consists of a set of realtime tasks that vary in their 'importance' in ensuring the correctness of the

system, e.g., the successful execution of some tasks may be more important than the others. In general, we can classify the tasks as critical or non-critical, based on the consequences of deadline misses; a deadline miss on a critical task can cause catastrophic consequences, while a deadline miss on a non-critical task can cause only a minor degradation of the service provided by the system. Integrating mixed criticality tasks on the same platform can be beneficial in various ways, particularly in reducing cost and energy consumption.

The main advantages of our approach are:
1) Efficient handling of task criticalities using feasibility windows.
2) Improved processor utilization and hence cost reduction through optimization.
3) Fault tolerance strategies covering multiple fault types.
4) Graceful degradation of the system under faults.
5) Supports the development of certifiable fault-tolerant mixed criticality systems.

There are two types of feasibility windows:
1) Fault Tolerant (FT) feasibility windows for critical tasks
2) Fault Aware (FA) feasibility windows for non-critical tasks

A Fault Tolerant Feasibility Window (FTW) is a temporal window in which a critical task has to complete its execution, such that it can feasibly re-execute (i.e., before its original deadline) upon an error. A Fault Aware Feasibility Window (FAW) is a temporal window allocated to non-critical tasks,in order to control their interference with the critical ones, i.e., the execution of a non-critical task may not jeopardize the fault tolerant execution of any critical one.

Let us consider 2 tasks A and B, where A is a non-critical task and B is a critical task. Let the time period and execution time of A be 3 and 2 respectively and that of B be 6 and 2. Let us assume that the maximum number of re-executions required by B is 1.

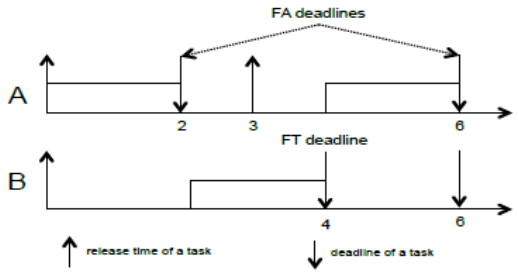

Fig. 2. FT and FA feasibility windows

a) Fault Tolerant Feasibility Window Derivation:
The latest time at which the alternate of task B should start executing to enable one feasible execution is given by its deadline minus the worst case execution time of the alternate. Since, according to our assumption, the WCET of the alternate is no greater than the WCET of the primary, the alternate of task B must start executing at time t = 6-2 = 4 to guarantee its successful execution. Hence the FT feasibility window of the primary of B is given by the interval, (0; 4] and that of its alternate is given by the interval (4; 6] as shown by the figure.
b) Fault Aware Feasibility Window Derivation:
To derive the FA feasibility window for task A, we first schedule the primary of b to execute as late as possible and schedule A in the remaining slack. The figure shows the FA deadlines of task A. Hence the FA-feasibility window of task A is (0; 2] for its first job and (3; 6] for its second job. In some cases, it might not be possible to derive FA feasibility windows for the non-critical tasks. In these cases, the non-critical tasks are assigned with their original feasibility windows, given by their original release times and deadlines, and while deriving task priorities, they are assigned a background priority so that they do not influence the critical task executions.

6. DISKLESS CHECKPOINTING APPROACH

Error recovery via checkpointing can be done in disk based or diskless manners. In the disk based approach, checkpoints are stored and read-back from a safe and a non volatile disk which incurs a great timing overhead to the system. But in diskless approach, the need of disks is eliminated and instead of such low speed storage devices, fast speed processors memories are used to store and read back checkpoints. In this method, check points and states of a processors is stored in the memory of other processors. When a failure occurs, the faulty processor can be recovered by restoring its checkpoints stored in the memory of a healthy processor.

Since disks are known as low speed devices using them for storing and reading back checkpoints would increase the total system runtime. On the other hand as diskless approaches uses fast memories, compared to disk based approach they would have a great impact on system performance. Some fault tolerant schedulers employs diskless checkpointing scheme to increase system performance. This technique eliminates the need of low speed hard disks for saving checkpoints which eventually leads to reduce the time takes to store checkpoints as well as reduces the interval of saving checkpoints. It is also implementable in multiprocessor systems which use processor's memory to store checkpoints.

There are many methods for diskless checkpointing technique. Neighbor based and coding based are two of the most famous methods. In neighbor based method, the checkpoints of the application processor is stored in the memory of the checkpointing processor. It cannot tolerate coincident failure of both procesors neither. Memory consumption is one of the most significant drawbacks in neighbor based approaches.

In the coding based m out of an existing processor are used as checkpointing processor to code and store checkpoints of the application processors. Therefore the check points of the faulty processors can be decoded and calculated again by using checkpointing and application processors. Coding based methods have two steps: each application checkpoints are coded and stored in the memory of the checkpointing processor which solely codes, stores and decodes checkpoints.

Parity technique requires only one dedicated checkpointing processor to store checkpoints of all application processors. The jth byte of the checkpointing processor is the result of taking XOR from jth byte of checkpoints of all application processors. When a processor fails, the checkpointing processor recovers the faulty processor by using its coded checkpoint and the checkpoint of other healthy processors. This technique reduces the size of checkpoint, but to recover faulty processor.

Sima and Nasser proposes a method is a combination of neighbor based and coding based approaches which stores checkpoints in the memory of other processors and therefore the need of hard disks for storing checkpoints is eliminated. It is supposed there are n processors in the system and each processor can have one or more tasks. The proposed method groups processors. The number of groups is equal to the number of simultaneous faults should be tolerate. At first, each processor calculates its checkpoint and stores it in its dedicated memory. To tolerate K simultaneous faults after grouping processors into K distinct groups, An XOR is taken from the first tasks of processors of a given group and the result is stored in the memory of the first and second processors of the next group. This process is repeated for all corresponding tasks in all processors of all groups.

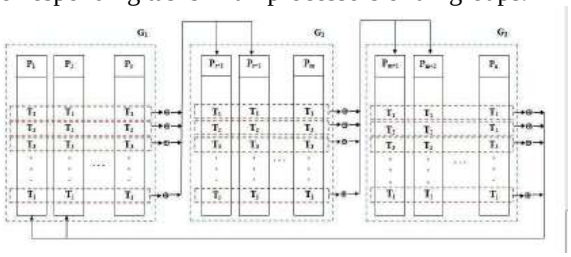

Fig.3 Coding and storing checkpoints to recover multiple failures.

In each group one faulty processor is recoverable. The recovery process is as follows: whenever a processor fails, a message is sent to the first checkpointing processor in the next group. If the checkpointing processor is healthy, the recovery process starts. Otherwise the message is sent to the second checkpointing processor. The checkpointing processor using its coded checkpoints and the received checkpoints decodes and recovers the checkpoints of the faulty processor and sends it back to the previous group to recover the faulty processor. The recovered processor recalculates and stores checkpoints of its tasks to be used for other processors failure in the future.

7. CONCLUSION

Error recovery is one of the most important parts of fault tolerance which leads to increase systems dependability. In this paper we describes five algorithms which will be used in various application depends on the situation. For future work it is suggested to consider the cost of performing scheduling and taking and saving checkpoints.

REFERENCES

[1] Nagarajan and John P.Hayes, " Task Scheduling algorithms for fault tolerance in real time embedded system"

[2] Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat ," Mixed Criticality Scheduling in Fault-Tolerant distributed real time systems", IEEE 2014.

[3] Sima and Nasser,"Diskless Checkpointing approach for failure recovery in multiprocessor safety critical embedded system", Iranian Conference on Electrical Engineering(ICEE 2015), Pg.No-688

