The Integration of Scheduling and Fault Tolerance in Real-Time Systems *

Prof. John A. Stankovic Fuxing Wang

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

1 Introduction

Next generation, mission critical systems will require greater flexibility, predictability, and reliability than is commonly found in today's systems [17]. While many advances are required to achieve the cost-effective engineering of these systems, one important research topic is the integration of schedulability and fault tolerance. In the past this topic has been approached by engineering expensive and highly static solutions, dependent on special purpose interconnection networks and architectures and static scheduling. This approach is not suitable for most next generation systems. Two promising approaches for more dynamic situations are the use of *imprecise computations* and the use of *planning mode* schedulers [12]. What is especially attractive is that these two approaches are not competing but rather can be used in conjunction with each other. In Section 2 we first discuss some background material on the interaction of scheduling and fault tolerance. In Section 3, we present the imprecise computation model and discuss its advantages and disadvantages. In Section 4 we present the planning mode scheduling approach with its advantages and disadvantages and suggest how planning can be combined with imprecise computation. In Section 5 we summarize key points and present a list of outstanding issues.

2 Background

Many solutions used today for scheduling real-time tasks are centralized and static. They are centralized in that the solutions are for single processor systems, or are simply applied in distributed systems ignoring system-wide requirements. They are static in the sense that they assume complete and prior knowledge of *all* tasks and make a priori scheduling and allocation decisions for the tasks. These static solutions suffer from inflexibility and are not feasible for dynamic, decentralized, complex real-time systems. On the other extreme one usually finds solutions for dynamic real-time systems based on assigning priorities to tasks and then using preemptive priority-based scheduling. This approach suffers from a number of problems: First, a single value, namely a task's priority, has to reflect a number of characteristics of the task including its deadline and level of importance. This assignment is error-prone and causes several well known anomalies because deadline and importance are not always compatible. Second, using this approach it is only known that a task has missed its deadline at or after the deadline occurs. This does not allow time for any corrective actions and is not suitable for many systems. Third, priority scheduling (as commonly defined) only addresses the cpu resource. This is a mistake. What value is there to immediately scheduling a task with a close deadline if the first action that the task takes is to ask for a locked resource and therefore must wait? What is required is an integrated approach to cpu scheduling and general resource allocation (i.e., resources other than the cpu such as data structures) [21]. In other words, the scheduling problem is NOT just a cpu scheduling problem, it is a combination of cpu scheduling and resource allocation. It is this combined problem which must be integrated with fault tolerance. For example, as we shall see later, while imprecise computation has tremendous potential, some scheduling solutions associated with imprecise computation advocate using earliest deadline scheduling without considering resource requirements. While this is a valuable first step in developing the theory, it is highly unlikely that complex systems will be composed of all independent tasks. Fourth, priority scheduling by itself provides little *direct* support for quantitatively assessing the performance of the system with respect to timing requirements.

Many fault tolerant mechanisms can be applied to real-time applications [2,3,9]. Each has different advantages and disadvantages based on different fault assumptions. Some fault tolerant mechanisms are based on temporal redundancy. *Retry* repeatedly executes the same software module. This tolerates transient errors. If a different copy of the same software module is used in each retry, this is a form of *Recov-*

*This work is part of the Spring Project at the University of Massachusetts and is funded in part by the Office of Naval Research under contract N00014-92-J-1048 and by the National Science Foundation under grant CDA-8922572.

ery Block (RB). In general, RBs may use a different software version for each retry block. This idea has been extended to *Distributed Recovery Blocks* [6]. Other fault tolerant mechanisms are based on spatial redundancy. *N-Modular-Redundancy* is used to bypass hardware faults, and a similar idea is used in *N Version Programming* (NVP) to tolerate software design faults. In this case, the outputs of the N versions are voted upon. Note that none of these techniques necessarily deal with timing constraints (although that is also not precluded). Some fault tolerant mechanisms are based on the timing properties of software modules. For example, a *Deadline Mechanism* can be used to select the proper version of a software module which meets a known deadline. The *imprecise computation* approach is based on an iterative method so that system scheduling can trade off time and precision of a software module with a deadline constraint. This same idea is used in the *Mandatory-Optionals* work [14] which does not require iteration and can be considered a discrete form of imprecise computation. Many systems contain fault tolerant mechanisms which are based on a combination of the above. *Multi-Primary-Voting and Multi-Ghost-Backup* is used in [8]. *N Self-Checking Programming* combines both temporal and spatial redundancy [9]. *Multi-Language Versions* combines NVP and RB [11]. A more complicated and flexible fault tolerant scheme, called *Resourceful System*, is reported in [1].

Since many real-time applications also require a high degree of fault tolerance, many systems must deal with both fault tolerance and timing issues. However, it is surprising that there are very few approaches that *explicitly* address real-time scheduling to meet timing and fault tolerance requirements. For example, much scheduling work attempts to produce a single safe schedule rather than validating all possible error recovery schedules.

3 Imprecise Computation Model

There are many different ways to address fault tolerance. Some techniques create redundancy in time. For example, task A and one (or more) backups(s) are scheduled so that if A fails, then there is still enough time to detect this failure and still execute the backup(s) before the deadline. It is important to note that the backup may or may not produce the full *accuracy* of computation, but that, in general, it is assumed that the answer is available only at the end of the computation. Other techniques create redundancy in space. For example, n copies are scheduled on different nodes and synchronized so that they can vote on outputs and complete by a deadline. Failures are masked. For each of these two techniques many variations have been used.

Both of these models require significant redundancy. Another fault tolerance model without redundancy is to schedule tasks so that as many high value tasks as possible make their deadlines. If, because of overload or processor failures, it is impossible for all tasks to make their deadlines, then the scheduler is responsible for omitting the least important work. While this approach is cheaper than those using redundancy, it suffers from the fact that some tasks may not execute at all (in the case of host failures this may be very important tasks and in the case of overload it should be the least important ones if the system is designed properly).

Recently, another model has emerged called the imprecise computation model [14,10,5]. This model prevents timing faults by reducing the accuracy of results in a graceful manner. In other words, an answer is available before task completion, but after a mandatory part completes execution (the mandatory part could be very small or zero), and its accuracy improves as more execution is consumed (considered the optional part). Note that this approach deals with a single task with the above features. Consequently, a failure such as the processor stops during the mandatory part, is not handled. Rather, the type of failure that is handled is *running out of time*. On the other hand, the approach is not precluded from being combined with other fault tolerant approaches (based on redundancy) in order to handle processor failures.

Let us now see how the imprecise computation model interacts with scheduling. In this model consider that each task is composed of a mandatory part (can be zero) and an optional part. If the entire task executes (mandatory plus the entire optional part) we consider the answer to be precise (error of zero). The scheduling algorithm must guarantee that all mandatory parts will meet their deadlines. This provides a minimum level of guaranteed performance even in overloads that would otherwise possibly cause an important computation to be late missed. The scheduling algorithm also attempts to execute as much of the optional parts as possible to minimize error. Many variations and extensions are possible. For example, different metrics can be used by the scheduler including minimizing the total error, minimizing the number of discarded optional tasks, and minimizing the maximum or average error. Extensions to combine this approach with another new approach called the planning mode approach[1] (see Section 4) are possible; extensions that combine imprecise computation with space redundancy are possible; extensions for dealing with the case where the set of tasks to be scheduled have different weights, where the tasks are periodic, and where tasks can be parallelized have been developed to some extent.

In all of the scheduling solutions developed so far in the imprecise computation model, tasks are pre-

[1] Actually, depending on how the imprecise computation model is used it can already be a planning mode scheduler.

emptable and independent. This enables the use of a rate monotonic or earliest deadline foundation for the basic guarantees. However, communicating tasks (in both multiprocessors and distributed systems) and those that share resources such as data will, in principle, void the basic assumptions behind these techniques. Future work must address these more complicated types of programs. Other practical problems are actually being able to implement tasks with mandatory and optional parts and knowing the error function associated with the optional part. In a dynamic system this can be quite complicated because the error function may be dependent on the current system state and, even worse, may be correlated to past executions of the task, e.g., the error may be cumulative over the past n execution of the task. This will require much more sophisticated scheduling algorithms whose execution time may grow too large. Also, in a complex system many types of tasks will be co-resident: those whose errors accumulate and those which do not, periodic and non-periodic, communicating and independent, sets of tasks with precedence (more than just precedence between the mandatory and optional parts of a single task), preemptive and non-preemptive, etc. How to develop scheduling heuristics for this set which provides the basic guarantees that the imprecise computation model provides is difficult and subject to future research. On the other hand, before we appear too negative, we believe that the imprecise computation model has great potential and all other methods also have similar problems.

4 Planning Mode Schedulers

One must recognize that there are different types of real-time systems with respect to timing constraints, granularities of time requirements, synchronization constraints, and of particular interest, fault tolerance requirements [18]. A real-time kernel must be reasonably generic so that it can be used in many different applications with differing requirements. Here we focus on how support for certain types of fault tolerance can be achieved by using on-line scheduling in a planning mode, and how on-line scheduling, together with some basic language support, can provide the flexibility required so that the same basic approach can be used in many types of applications. Some advantages of using scheduling in a planning mode include forecasting timing errors, carefully planning the activities of error recovery, planning fault masking for redundant application modules, planning system reconfigurations, managing the processing of exceptions, supporting graceful degradation in several ways including the use of imprecise computations, and allowing the dynamic adaptability of fault tolerance techniques. We also discuss how various fault tolerant models can be supported by three basic mechanisms plus an on-line scheduler.

The Spring system is currently being designed to be able to represent many fault tolerant mechanisms by three basic fault tolerant structures. These structures are then *dynamically* supported via the Spring *planning mode* scheduler. This approach allows the support of basic fault tolerance mechanisms as well as innovative combinations of planning and imprecise computation. The basic structures are the *Voting Scheme* (VS), *Primary-Backups* (PB), and *Alternative Set* (AS). While these three structures are straightforward, dynamic support for them is not common. The schemes work as follows.

VS is the typical voting technique where n copies of tasks vote. This is represented to the scheduling algorithm as the need to schedule n task copies and 1 or up to n voter tasks. There are location constraints on the n task copies and precedence constraints between the n copies and the voters. There is a single deadline on the voters. PB works by requesting the scheduling of one active task and $n-1$ backups with location constraints and one deadline which is the time by which exactly one of these tasks must complete. The scheduler plans the execution of the tasks so that these semantics are achieved. AS selects an execution time limit from a predetermined set of computation times among one or more software modules so that the selected module can be guaranteed to meet its deadline. This supports a discrete form of imprecise computation. However, if full computation time required for the selected task can be guaranteed, then it is guaranteed, else it receives an execution time that can be guaranteed for it. If this is less than a minimum amount of time required, then the task is not guaranteed. Note that the process of guaranteeing is the important part of scheduling and in Spring this includes addressing resource requirements. To use these three fault tolerance structures to represent a large class of fault models, the notions of task and task group are used. Each VS, PB, or AS mechanism is implemented as a task group which consists of a set of tasks with certain fault semantics. A task group may include other task groups. Using this type of recursive definition, many complex fault tolerant structures can be represented. For example, the AS mechanism can be used to implement imprecise computation to support *running out of time* errors, and embedding this AS structure in a PB mechanism can provide the redundancy necessary to support processor failures.

Tasks have different fault semantics. Because of the planning approach faults can be categorized into *pre-run time faults* and *run-time faults*. The pre-run time faults happen when the system load is relatively high and a scheduling algorithm can not find a feasible schedule. Thus, the algorithm predicts that some tasks will miss their deadlines. This type of fault can be handled by the imprecise computation model (AS task groups) such that the scheduler can avoid the faults by selecting an appropriate level of imprecise

computation. Run-time faults are mainly detected by the validation process embedded in the tasks or by the synchronization among redundant modules of a task, or after a task misses a deadline. The ultimate goal is to design a fault tolerant real-time system which can tolerate both pre-run time faults and run-time faults. This can be done by combining the structure of imprecise computation with other fault tolerant structures, e.g., VS and PB.

Our scheduler consists of two components: a task configuration module and a guarantee routine. The task configuration module dynamically configures the current task set by balancing the redundancy level of fault tolerance tasks and the load of the system, while the guarantee routine finds a feasible schedule for the task set determined by the task configuration module.

For the purpose of easy management, all user applications are divided into jobs. Each job has one or more versions of actual implementation, which is called as *alternatives*. Each alternative could be a general task group (non fault tolerant), a VS group, a PB group, a AS group, or a more complex task group. The different alternatives of jobs represent the different levels of fault tolerance requirements.

When a job arrives at the system, the new job and all unfinished jobs become candidates to be scheduled. The task configuration module determines an alternative for each candidate. All the selected alternatives constitute the tasks to be scheduled by the guarantee routine. If the guarantee routine fails to find a feasible schedule, the task configuration module must work out a new set of alternatives to permit the guarantee routine make another try. The process stops when a feasible schedule is found by the guarantee routine.

Figure 1 shows the effect of the task configuration module. The performance data are generated from the Spring Simulation Testbed. More simulation results can be found in [19], which compares the performance resulting from various combinations of task configuration and guarantee algorithms.

5 Summary

Real-time scheduling is very complicated, fraught with anomalies and misconceptions. Fault tolerance is also very difficult. Needing to simultaneously support both is even more difficult. What is required is a strong underlying theory, proper system level support (such as real-time languages [7], on-line scheduling algorithms, synchronization mechanisms, agreement protocols, recovery algorithms, mechanisms to support adaptive fault tolerance, architecture support [15,20]), and useful design and analysis tools – all developed by experts to insulate users from the complexity and to avoid mistakes. Each of these areas must focus on the integration of scheduling and fault tolerance and not treat them independently. While new approaches that arise from considering scheduling and fault tolerance together are to be highly encouraged (such as imprecise computation and planning mode scheduling), other more incremental solutions should also prove valuable. For example, beginning with some basic scheduling theory results and extending them to address the fault tolerance nature of tasks may prove very important. Extensions to the rate monotonic theory fall into this category. Similarly, beginning with basic fault tolerance results and explicitly trying to extend those results to consider the timing requirements in a flexible manner may also prove valuable. Results where replicas and voting tasks are dynamically scheduled fall into this category.

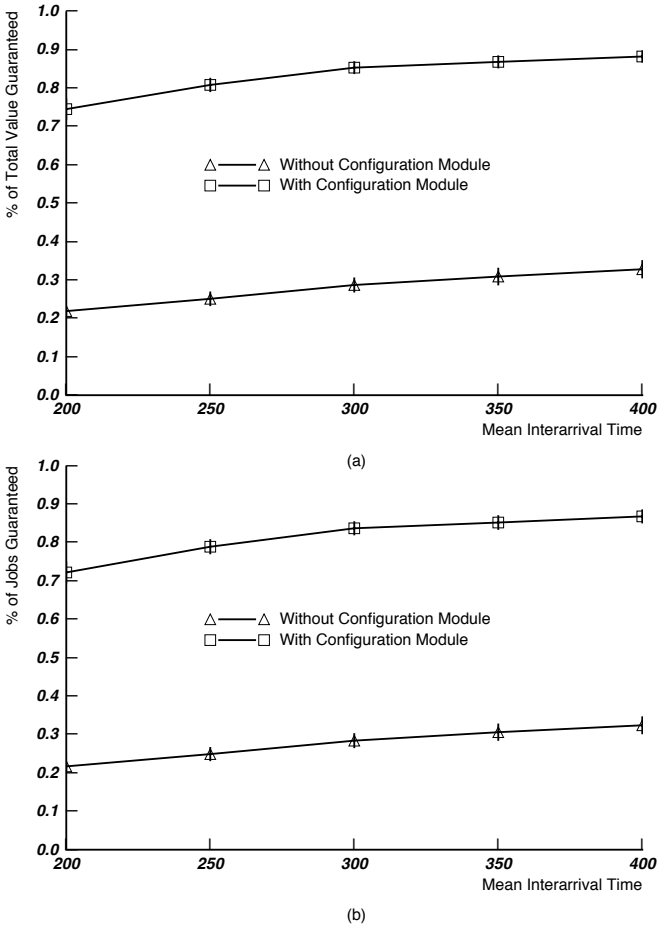

Figure 1: The performance with/without the configuration module.
#cpu=5, #resource=8, Use_P=0.1, Share_P=0.7, heuristic function : h = dl + w1 * est, w1 = 6.0, job_value in uniform distribution (3000, 4000), each job has 2 alternatives, the first alternative has, #process in uniform distribution (4, 6), the second alternative has one process, each process has 1 task, task_wcct in uniform distribution (100, 150), task_avrg in uniform distribution (70%, 90%) of task_wcct, latest finish time as task's deadline, job laxity in two uniform distributions (300, 500) of 60% and (700, 1000) of 40%.

Some of the main research questions are: What are good sets of integrated scheduling policies that span cpu scheduling, I/O scheduling, communication needs, resource allocation, and fault tolerance requirements; What new scheduling theory is required to support the above integration of issues; How can dynamic scheduling contribute to the tradeoff between time and space redundancy; Can a single sophisticated scheduling algorithm *cost effectively* handle complex task sets, or will tasks be partitioned into equivalence classes with algorithms tailored to each class; How would such a set of algorithms interact; What type of predictability, including fault tolerance guarantees, is possible for distributed real-time computation and can a comprehensive scheduling approach that supports predictable and analyzable distributed real-time systems be developed; How can task importance, computation time, tightness of deadline, and fault requirements be traded off to maximize value in the system; what are the roles of the scheduling algorithms in this analysis; What is the impact of off-line allocation policies and fault tolerance policies on dynamic on-line scheduling; How to deal with overloads and what should be the expectations with respect to performance and predictability in overload situations; How can we make scheduling and fault tolerance cost effective; How can we determine the impact when fault hypotheses and load hypotheses are wrong; What is the role of run time monitoring in supporting scheduling under fault assumptions;, and How can fault tolerance policies be adapted as systems react in the short and long term to changing environment and system conditions, objectives, and requirements.

References

[1] R. J. Abbott. Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys*, 22(1):35–68, 1990.

[2] T. Anderson and P. A. Lee. *Fault-Tolerance – Principles and Practice*. Prentice Hall, International, London, 1981.

[3] A. Avizienis and J. Laprie, eds, Dependable Computing for Critical Applications. Vol. 4, Springer-Verlag, 1991.

[4] J. Bannister and K. Trivedi. Task allocation in fault tolerant distributed systems. *ACTA Informatica*, 20:261-81, 1983.

[5] J. Chung, J. Liu, and K. Lin. Scheduling periodic tasks that allow imprecise results. *IEEE Trans. Computers*, Vol. 19, No. 9, Sept. 1990.

[6] K. H. Kim and H. O. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Trans. Computers*, 38(5):626–36, May 1989.

[7] E. Klingerman and A. D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, September 1986.

[8] C. M. Krishna and K. G. Shin. On scheduling tasks with a quick recovery from failures. *IEEE Trans. Computers*, 35(5):448–55, May 1986.

[9] J. C. Laprie et al. Hardware- and software-fault tolerance: Definition and analysis of architectural solutions. In *Digest of Papers FTCS-17*, pages 116–21, 1987.

[10] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao, Algorithms for Scheduling Imprecise Computation. *IEEE Computer*, May 1991.

[11] J. M. Purtilo and P. Jolote. A system for supporting multi-language versions for software fault tolerance. In *Digest of Papers FTCS-19*, pages 268–74, 1989.

[12] K. Ramamritham, J. Stankovic, and P. Shiah. Efficient scheduling algorithms for real–time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

[13] K. Ramamritham. Scheduling complex periodic tasks. *Intl. Conference on Distributed Computing Systems*, June 1990.

[14] W.-K. Shih, J. W. Liu, and J.-Y. Chung. Fast algorithms for scheduling imprecise computations. *Real-Time System Symposium*, pages 12–19, December 1989.

[15] K. Shin. HARTS: A distributed real-time architecture. *IEEE Computer*, Vol. 24, No. 5, May 1991.

[16] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, Vol. 1, pp. 27-60, 1989.

[17] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, October 1988.

[18] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A new paradigm for real-time operating systems. *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989, pp. 54-71.

[19] F. Wang. *Dynamic Scheduling in Real-Time Systems — Algorithms and Analysis* . PhD Thesis, University of Massachusetts, in preparation.

[20] Wensley, et. al., SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Procs. of the IEEE*, 66(11), October 1978, pp. 1240-1255.

[21] W. Zhao, K. Ramamritham and J. A. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Trans. on Software Engineering*, SE-12(5), May 1987.