Supporting Fault-Tolerant Real-Time Applications using the RED-Linux General Scheduling Framework [⋆]

Kwei-Jay Lin and Yu-Chung Wang

Department of Electrical and Computer Engineering
University of California, Irvine, CA 92697-2625
{klin, wangy}@ece.uci.edu

Abstract. In this paper, we study the fault-tolerant support for real-time applications. In particular, we study the scheduling issues and kernel support for fault monitors and the primary-backup task model. Using the powerful scheduling framework in RED-Linux, we can support a jitter-less fault monitoring. We can also provide the task execution isolation so that an erroneous runaway task will not take away additional CPU budget from other concurrently running tasks. Finally, we provide a group mechanism to allow the primary and backup jobs of a fault-tolerant task to share both the CPU budget as well as other resources. All these mechanisms make the implementation of fault-tolerant real-time systems easier.

1 Introduction

As more computer-based systems are now used in our daily life, many applications must be designed to meet real-time or response-time requirements, or human safety may be jeopardized. Real-time applications must be fault-tolerant both to timing faults as well as logical faults. Timing faults occur when an application cannot produce a result before its expected deadline. Logical faults occur when an application produce a wrong result before or after the deadline. Both types of faults must be handled in a fault-tolerant real-time system. Supporting fault-tolerant mechanisms in real-time systems therefore is a complex issue. Finding a powerful real-time OS to support fault-tolerant applications is even more difficult.

We have been working on a real-time kernel project based on Linux. Our real-time kernel project is called RED-Linux (*Real-time and Embedded Linux*). For efficiency, we have implemented a mechanism that provides a short task dispatch time [18]. To enhance the flexibility, we provide a general scheduling framework (GSF) in RED-Linux [19]. In addition to the priority-driven scheduling, RED-Linux supports the *time-driven* [7–9] and the *share-driven* (such as the proportional sharing [14] and approximations [2, 17]) scheduling paradigms. In this paper, we investigate how GSF in RED-Linux may support fault-tolerant real-time systems. We review the primitives for many fault-tolerant real-time

[⋆] This research was supported in part by UC/MICRO 98-085, 99-073 and 99-074, Raytheon and GeoSpatial Technologies, and by NSF CCR-9901697.

system models and study how to support (or enforce) them in the framework. By adjusting scheduling attribute values and selection criteria in the scheduler, it is possible to implement many fault-tolerant scheduling algorithms in our framework efficiently.

In particular, we study the scheduling issues and kernel support for fault monitors and the primary-backup task model. Using the powerful scheduling framework in RED-Linux, we can support a jitterless fault monitoring. We can also easily specify the CPU budget for each computation so that an erroneous runaway task will not take away the CPU budget reserved for other concurrently running tasks. Finally, we provide a group mechanism to allow the primary and backup jobs of a fault-tolerant task to share both the CPU budget as well as other resources. All these mechanisms make the implementation of fault-tolerant real-time systems easier.

The rest of this paper is organized as follows. Section 2 reviews popular scheduling paradigms used in real-time systems and other real-time OS projects. Section 3 briefly introduces the RED-Linux general scheduling framework. We then study the fault monitoring issues for real-time system in Section 4. Section 5 presents the design of the task group mechanism in RED-Linux. The paper is concluded in Section 6.

2 Related Work on Fault-Tolerant and Real-Time Support

Several previous work has studied the fault-tolerant real-time scheduling issues. Liestman and Campbell [11] propose a scheduling algorithm for frame based, simply periodic uniprocessor systems. Each task has two versions: primary and backup. Task schedules are dynamically selected from a pre-defined schedule tree depending on the completion status of the primary tasks. Chetto and Chetto [5] present an optimal scheduling strategy based on a variant of the EDF algorithm, called EDL, to generate fault-tolerant schedules for tasks that are composed of primary and alternate jobs. Their method provides the ability to dynamically change the schedule and accounting for runtime situations such as successes or failures of primaries. Caccamo and Buttazzo [4] propose a fault-tolerant scheduling model using the primary and backup task model for a hybrid task set consisting of firm and hard periodic tasks on a uniprocessor system. The primary version of a hard task is always scheduled first if it is possible to finish it and the backup task before the deadline. If not, only the backup task is scheduled. Another interesting work related to real-time fault-tolerance is the Simplex architecture [15]. The Simplex architecture is designed for on-line upgrade of real-time software applications by using redundant software components. By allowing different versions of a software component to be executed in sequence or in parallel, real-time application software can be dynamically replaced with negligible down-time. The architecture can also be used for fault tolerance.

Our goal in this paper is not to propose a new fault-tolerant model but to study the OS support for those proposed earlier. Using RED-Linux's general scheduling framework, we hope to be able to support many existing fault-tolerant mechanisms effectively and efficiently. To support the fault-tolerant mechanisms mentioned above, at least two mechanisms are necessary. The first is a way to define the group relationship between related tasks (primary and backup, old and new versions etc.) to allow them to share the budget for CPU or other resources. The other is a predictable monitoring facility. These fault tolerance supports from RED-Linux will be discussed in this paper.

3 The RED-Linux General Scheduling Framework

The goal of the RED-Linux general scheduling framework (GSF) is to support most well-known scheduling paradigms, including the *priority-driven*, the *time-driven* [7–9] and the *share-driven* [14, 2, 17], so that any application can use RED-Linux for real-time support. Two features have been introduced: the general scheduling attributes used in the framework and the scheduler components used to make scheduling decisions.

In our model, the smallest schedulable unit is called a *job*. For systems with periodic activities, we call a job stream as a periodic *task*. Different scheduling paradigms use different attributes to make scheduling decisions. In order for all paradigms to be supported in GSF, it is important for all useful timing information to be included in the framework so that they can be used by the scheduler. We denote four scheduling attributes for each job in GSF: *priority*, *start_time*, *finish_time*, *budget*. Among the four, the start_time and the finish_time together define the *eligible interval* of a job execution. The priority specifies the relative order for job execution. The budget specifies the total execution time assigned to a job. These attributes can be used as constraints. However, these timing attributes can also be used as the selection factors when a scheduler needs to select a job to be executed next.

RED-Linux uses a two-component scheduling framework in . The framework separates the low level scheduler, or *dispatcher*, from the QOS parameter translator, or *allocator*. We also design a simple interface to exchange information between these two components. It is the allocator's responsibility to set up the four scheduling attributes associated with each real-time job according to the current scheduling policy. The dispatcher inspects each job's scheduling attribute values, chooses one job from the ready queue and dispatches it to execution. In addition to assigning attribute values, the allocator also determines the evaluation function of scheduling attributes, since each job has multiple scheduling attributes. This is done by producing an *effective priority* for each job. The allocator uses one or more attributes to produce the effective priority so that the dispatcher will follow a specific scheduling discipline.

More details on the GSF implementation and the performance measurement can be found in [19].

4 The Design of Fault Monitors

To provide fault tolerance, three facilities can be supported: fault detection, fault avoidance, and fault recovery. Fault-tolerant systems must be able to monitor the system and application status closely and predictably. The earlier a fault can be detected and identified, the easier it may be fixed.

Depending on the type and the likelihood of faults to be monitored, cyclic monitoring is often used in systems with safety properties that must always be maintained. For example, many system components send "heartbeat" messages to each other or to a central controller to let them know that the component is still alive and well. Another example is a temperature monitoring facility that constantly reads the temperature sensor and produce a warning if the temperature is too high. Cyclic monitors are scheduled independently from any user applications. Depending on their importance, they must be executed predictably and without jitters so that they do not miss a critical warning window for an important fault. However, the traditional priority driven scheduler may not provide the kind of predictability required by fault-tolerance monitors. There is no guarantee on the execution jitters since the temporal distance between two consecutive executions of a monitor task may be as long as twice the period length [7, 8].

One effective scheduling paradigm in RED-Linux for cyclic monitors is the time-driven (TD) (or clock-driven) paradigm. For embedded systems with steady and well-known input data streams, TD schedulers have been used to provide a very predictable processing time for each data stream [7–9]. Using this scheduling paradigm, the time instances when each task starts, preempts, resumes and finishes are pre-defined and enforced by the scheduler. User applications may specify the exact time and cycle when a monitor should be activated; the Dispatcher will activate the monitor accordingly. However, using the general scheduling framework in RED-Linux, other tasks may use their own schedulers independent of the TD scheduler for monitors. The integration of TD schedules with these application schedulers has many interesting issues. For example, if an application uses the fixed priority driven scheduling such as rate monotonic scheduling in the presence of TD schedulers, can we still guarantee that all periodic jobs will meet their deadlines using the schedulability condition for the RM model [12]?

Suppose a fault-tolerant system has monitor jobs and priority driven (PD) jobs. The monitor jobs are scheduled using TD at exact times. Therefore the priority-driven jobs are scheduled after the TD jobs are executed. If a PD job is running when a TD job is scheduled to start, the PD job will be preempted. In other words, all TD jobs are considered to have a higher effective priority than all PD jobs.

Using the RM scheduling, a system of n tasks are guaranteed to meet their deadlines if the total utilization satisfies the condition:

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1)$$

where a task τ_i must be executed for c_i time units per p_i time interval. However, when a RM system is scheduled after a time-driven scheduler, the execution of a periodic task may be delayed or interrupted by a TD job. To handle this problem, we can treat all TD jobs as "blocking" for PD jobs just like PD jobs are blocked on accessing critical sections. We can model all TD jobs as critical sections for PD jobs. As long as all TD jobs are short enough, the schedulability of all PD jobs can be guaranteed using this approach. In other words, all PD jobs can meet their deadlines as long as:

$$(\sum_{j<i} \frac{c_j}{p_j}) + \frac{c_i + b_i}{p_i} < i(2^{1/i} - 1)$$

where b_i is the blocking time by all TD jobs for task τ_i. To reduce b_i, we need to make sure that all monitoring jobs have short execution times and are not clustered together.

Another useful facility for fault-tolerant real-time systems is to monitor the timing events when executing user programs. Timing faults, i.e. results produced too late or an application uses more than its share of resources, may cause the system to produce erroneous responses. The OS should provide a powerful yet efficient mechanism to detect timing faults.

In the original GSF reported in [19], the Dispatcher reports only those events when jobs are terminated (voluntarily or involuntarily). Some fault monitoring algorithms need to know the exact time when a job is executed, suspended or terminated. To support these algorithms, we have extended the original GSF feedback mechanism so that the Dispatcher sends these information about job executions to the Allocator.

5 The Implementation of Task Group in RED-Linux

Many fault-tolerant functions are implemented using the primary-backup model or the N-version model so that a specific functionality can be provided by multiple jobs. For real-time scheduling, it is important for all these jobs to share a common CPU budget so that other tasks in the system will have enough CPU time for their execution. A *group* structure is thus introduced to distinguish a set of jobs from others. Every job in the set is assigned the same group number. For example, the primary-backup model [11, 5, 4] have two jobs. The two jobs may have different start time and finish time, as well as different priorities. However, the two jobs must share a total budget. Another example is the imprecise computation model [13] where a task may have many optional jobs that can be used to enhance the result quality. The optional jobs should share a common CPU budget for the group.

To support the task group, we have implemented a hierarchical task group mechanism in RED-Linux. Similar to the concept of the hierarchical file directory structure, a task group may have other task sub-group as a member. For scheduling purposes, each task group is assigned some specific resource budget to be shared by all tasks and task sub-groups in the group. Moreover, each task group may use a different scheduling policy to assign its resources. Therefore, the Dispatcher in GSF needs the capability to adopt different scheduling policies to select the next running job from a set of jobs.

In RED-Linux, each job is in a group and has a group number. Since each sub-group is like a job in a group, each sub-group has a group number as well. Another parameter, the *server* number, is used by each sub-group to identify itself as a server (for scheduling). Each server job is associated with a job queue which holds all jobs that will be scheduled using the server job's budget. Each server can use a different policy to schedule the jobs on its queue. For normal real-time jobs, they do not have a server number. Therefore the server number of a normal real-time job is set to be the same as the group number which it belongs. In other words, a job can be identified as a *server job* if its server number is different from its group number. A server job is a scheduling unit with an allocated budget but no application job. The Dispatcher will not execute the server job. Instead, it will select another job to be served by the server. The group algorithm implemented in the RED-Linux Dispatcher is shown as follows.

1. The Dispatcher selects a job K from all eligible jobs in group 0 according to the scheduling policy of group 0.
2. The server job list J is initialized to NULL.
3. If K is a real-time job, execute it, else if K is a server job with server number i, select a job L from all eligible jobs in group i.
   (a) Set a new timer to be the minimum of the following values: the current time plus the budget of the server job K, the current time plus the budget of L, the finish time of the server job K, the finish time of L.
   (b) Append the server job K to the job list J.
   (c) Set $K = L$ and repeat Step 3.
4. When K finishes or is interrupted, reduce the actual execution time used from the budget of all jobs in the server job list J.

The maintenance of the server list is necessary so that the execution of a job will consume the budget in all groups it belongs to under the whole group hierarchy.

6 Conclusions

In this paper, we present the support for fault-tolerant real-time applications using the general scheduling framework in RED-Linux. The scheduling framework is able to accommodate a variety of scheduling models used in fault-tolerant real-time applications. By using the group mechanism, fault-tolerant primary-backup tasks may share the CPU and other resource budget efficiently. By using the time-driven scheduling, fault monitors can be executed efficiently and without jitter. By using the budget mechanism, tasks are always given their guaranteed share regardless of the possible ill behavior of other tasks.

References

1. L. Abeni and G.C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, Dec 1998.
2. J.C.R. Bennett and H. Zhang. WF2Q: Worst-case fair weighted fair queueing. In *Proc. of IEEE INFOCOMM'96*, San Francisco, CA, pp. 120-128, March 1996.
3. S. Punnekkat and A. Burns. Analysis of checkpointing for schedulability of real-time systems. In Proceedings of IEEE Real-Time Systems Symposium, pages 198–205, San Fran- cisco, December 1997.
4. M. Caccamo and G. Buttazzo. Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems. Proceedings of IEEE Conference on Real-Time Computing Systems and Applications, Hiroshima, Japan, Oct 1998.
5. H. Chetto and M. Chetto. An adaptive scheduling algorithm for fault-tolerant real-time systems. Software Engineering Journal, May 1991.
6. A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Journal of Internetworking Research and Experience*, pp.3-26, October 1990.
7. Ching-Chih Han, Kwei-Jay Lin and Chao-Ju Hou. Distance-constrained scheduling and its applications to real-time systems. In *IEEE Trans. Computers*, Vol. 45, No. 7, pp. 814-826, December 1996.
8. Chih-wen Hsueh and Kwei-Jay Lin. An optimal Pinwheel Scheduler Using the Single-Number Reduction Techniques. In *Proc. of IEEE Real-Time Systems Symposium*, December 1996, pp.196-205.
9. Chih-wen Hsueh and Kwei-Jay Lin. On-line Schedulers for Pinwheel Tasks Using the Time-Driven Approach. In *Proc. of the 10th Euromicro Workshop on Real-Time Systems*, Berlin, Germany, June 1998, pp. 180-187.
10. K. Jeffay et al. Proportional Share Scheduling of Operating System Service for Real-Time Applications. In *Proc. IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 480-491, Dec 1998.
11. A. L. Liestman and R. H. Campbell. A fault-tolerant scheduling problem. IEEE Transactions on Software En- gineering, 12(11):1089–95, November 1986.
12. C.L. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46-61, 1973.
13. J.W.-S. Liu, K.J. Lin, W.-K. Shih, A.C. Yu, J.Y. Chung and W. Zhao. Imprecise Computation. In *Proc. of IEEE* 82:83-94, 1994
14. A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. In *IEEE/ACM Trans. Networking*, Vol. 1, No. 3, pp. 344-357, June 1993.
15. L. Sha. Dependable system upgrade. In *Proc. IEEE Real-Time Systems Symposium*, pp. 440-448, Dec 1998.
16. M. Spuri and G.C. Buttazzo. Efficient aperiodic service under the earliest deadline scheduling. In *Proc. IEEE Real-Time Systems Symposium*, Dec 1994.
17. I. Stoica, H. Zhang, and T.S.E. Ng. A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Services. In *Proc. of ACM SIGCOMM'97*, Cannes, France, 1997.
18. Y.C. Wang and K.J. Lin. Enhancing the real-time capability of the Linux kernel. In *Proc. of 5th RTCSA'98*, Hiroshima, Japan, Oct 1998.
19. Y.C. Wang and K.J. Lin. Implementing a general real-time framework in the RED-Linux real-time kernel. In *Proc. of RTSS'99*, Phoenix, Arizona, Dec 1999.