Feedback Fault Tolerance of Real-Time Embedded Systems – Issues and Possible Solutions

Xue Liu, Hui Ding, Kihwal Lee, Lui Sha, Marco Caccamo
{xueliu, huiding, klee7, lrs, mcaccamo}@cs.uiuc.edu
Department of Computer Science, University of Illinois at Urbana-Champaign

Abstract

Fault tolerance is an important aspect in real-time computing. In real-time systems, tasks could be faulty due to various causes. Faulty tasks may compromise the safety and performance of the whole system and even cause disastrous consequences. In this paper, we study the possibilities of applying feedback control of software execution to real-time systems for fault tolerance purposes. A new fault tolerance architecture called ORTGA (On-demand Real-Time GuArd) is proposed. We argue the advantages and benefits of using ORTGA for fault tolerance in real-time systems. We also list research problems faced by ORTGA and point out directions for possible solutions. Throughout the paper, we use an example of real-time inverted pendulum control to illustrate ideas, problems and and possible solutions.

1 Introduction

Real-time and embedded systems are now a central part of our lives. Different from general computer systems, a real-time system is considered to function correctly only if it returns the correct result within the system-wide timing constraints [4]. Reliable functioning of real-time systems is of paramount concern to the millions of users that depend on these systems everyday. However, faults and failures can occur in real-time systems. Though failures can be caused by both hardware (e.g., electromechanical devices) and software, in this paper we focus on how to tolerate software faults in real-time systems.

Feedback is a universal mechanism which exists in many disciplines. Human uses feedback to correct faults and progress. Government uses feedback to avoid corruption and advance. Car cruise control uses feedback control to meet the targeted speed. Feedback is also commonly used in software industry: many Web sites (such as Amazon.com) use client feedbacks to improve their design; software vendors often employ user feedbacks to help select new features to be included in future releases; Microsoft uses application crash report (a kind of feedback) to improve the reliability of Windows operating system. In this paper, we discuss using feedback to achieve software fault tolerance. Specifically, we introduce ORTGA (On-demand Real-Time GuArd), a new fault tolerant architecture for real-time control systems.

Our objective is to identify some cutting-edge research problems and point out possible solutions on using feedback for fault tolerance in real-time systems.

The rest of the paper is organized as follows. The ORTGA architecture is presented in Section 2. To fix the ideas, a real-time control application (i.e. an inverted pendulum control system) is introduced as a motivating example throughput the paper. The research issues of feedback based software fault tolerance are elaborated in Section 3 within the context of ORTGA. Directions for possible solutions mainly addressing the timing issues of ORTGA are given in Section 4. We provide related work in Section 5 and conclude our paper in Section 6.

2 ORTGA Software Fault Tolerant Architecture

In this section, we discuss the ORTGA software fault tolerant architecture. One of the most important aspects of ORTGA is it uses *feedback control of software execution* to achieve fault tolerance. In order to understand this notion, let's first look at what are essential elements related to feedback control to make a general system (such as a social system – a government) fault tolerant?

Faults abound in any complex system such as a human government. Some kind of faults can not be easily eliminated, such as human operation errors or corruptions. In stead, these faults/errors are facts to be coped with. The idea of fault tolerance is to respond gracefully to these faults and not make them affect the healthy operation of the whole system. The first step in achieving a fault tolerant system is to detect the faults. Common ways to detect faults in a government include auditing or collecting employee feedbacks.

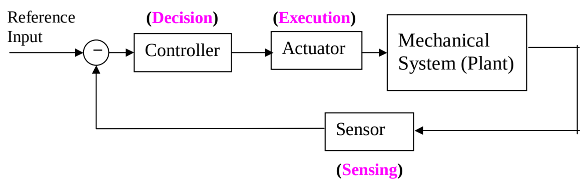

Figure 1: A typical feedback control loop

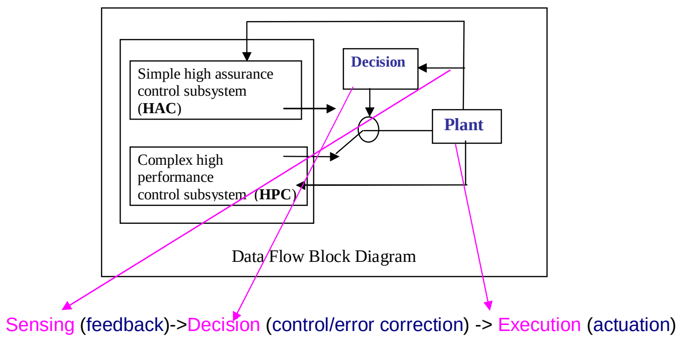
Simple high assurance control subsystem (HAC)
Complex high performance control subsystem (HPC)
Decision
Plant
Data Flow Block Diagram
Sensing (feedback)->Decision (control/error correction) -> Execution (actuation)

Figure 2: Feedback Control of Software Execution

We call this step as *(fault) identification step*. After a fault is identified, we need to decide what is a good way to get rid of the fault, or at least confine the fault from propagating to other functional units. The result is a correction scheme. We call this second step as *decision step*. The last step is to make sure the correction scheme is executed in order to correct the fault occurred. We call the last step as *execution step*.

These three steps correspond to a typical feedback control loop for a mechanical system, as shown in Figure 1. The (fault) identification step is similar to sensing (where the sensor finds state or output errors and feed it back to the controller). The decision step is similar to control (where the controller calculates the control values to correct the error). The execution step is similar to the actuation (where the actuator puts the control values from the controller into action).

Following this analogy, we now introduce the architecture of ORTGA. We also discuss how ORTGA employs feedback to make a real-time control system fault tolerant. The architecture of ORTGA is shown in Figure 2. Similar to the Simplex architecture [7], in ORTGA the software component of the plant under protection is divided into a high-assurance-control (HAC) subsystem and a high-performance-control (HPC) subsystem. The HAC subsystem is a control software which was proved to be reliable. HAC's simple construction let the system designer leverage the power of formal methods and a rigorous development process. From the system level, high-assurance OS kernels such as certifiable runtimes are usually used in the HAC. From the application level, well-understood classical controllers designed to maximize the controlled plant's stability region is also used.

The HPC subsystem complements the conservative HAC core. From application level, an HPC can use more complex and advanced control technologies for higher control performance, including those difficult to verify, for example, neural network control. From system level, COTS real-time OS and middleware designed to simplify the application development can be used in HPC. However, these software components may not be certifiable and could contain faults.

Unlike Simplex, in ORTGA the HAC and HPC subsystems do not run in parallel. At any time, there is *only one* instance of either HAC or HPC is running. Normally, the HPC controls the plant. However, the decision logic in the decision module ensures that the plant state under the HPC stays within an HAC-established stability region (to be discussed in Section 4). If this is violated, the HAC will be kicked in and takes over the system. However, there are two major drawbacks of the original Simplex architecture. First, Simplex requires parallelly running the two controllers HAC and HPC for each plant. This "trade system resource for safety" approach makes the whole system inefficient, hence limits its application only in extremely safety-critical applications where cost and resource usage is not a major concern. Most of the industrial applications are cost-sensitive. In these applications, it will be ideal to have a fault tolerance approach which both minimizes the resource usage, at the same time achieve high fault coverage. Second, In Simplex, the design of maximum stability region of a system under a controller is based on continuous design. Since controllers are implemented digitally, its maximum stability region will be affected by its sensing-control loop period. How to determine the maximum stability region of a system under digital controller remains unanswered.

As we can see from Figure 2, ORTGA achieves fault tolerance by using *feedback control of software execution*. At every decision time, the decision module gets the state feedback from the plant and determines if the current state is still within the HAC-established stability region. If it is, the HPC still controls the plant; otherwise, the HAC is activated and it takes over the control of the plant. The decision module determines which output should be used for the plant. Then the plant will execute the control output values accordingly. These *Sensing(feedback)* $\rightarrow$ *Decision(control)* $\rightarrow$ *Execution(actuation)* steps constitute the feedback control of software execution (cf. Figure 1). By using the HAC to guard against possible faults in the HPC in real-time, ORTGA achieves fault tolerance.

3 Research Issues in Feedback Based Real-Time Fault Tolerance

In this section, we discuss the research issues in feedback based real-time fault tolerance, concentrating on the ORTGA architecture. To this end, we first show a real-time control system and use it as a motivating example to illustrate the following discussions.

3.1 A Real-Time Control System

We consider a real-time inverted pendulum control system. Suppose there are N inverted pendulums running. Each inverted pendulum i is protected by an ORTGA instance O_i. Each ORTGA instance is responsible for fault tolerance in the corresponding pendulum (plant). Within each ORTGA instance, there is a high performance controller task (HPC_i) and a high assurance controller task (HAC_i). In this paper, we discuss a simple scenario where the two controller tasks for each pendulum have the same period and the same worst case execution time. Extensions to this model are left for future research. In ORTGA, at any time, for any pendulum i, either HPC_i or HAC_i is running but not both, so the controller tasks can be represented as a single real-time task τ_i. We use T_i to denote the sensing-control loop period and C_i to denote the worst case execution time of task τ_i. For control systems, the task deadline is usually the same as the task period, i.e. $D_i = T_i$. Figure 3 shows such a system with two pendulums ($N = 2$). Each ORTGA instance O_i, ($i = 1, 2$) is controlling one inverted pendulum. Each ORTGA instance corresponds to a real-time task τ_i, ($i = 1, 2$) running on the same CPU.

In real-time operating systems (RTOS), tasks are scheduled using some predetermined scheduling algorithms. There are two major types of priority-based scheduling algorithms, fixed priority scheduling algorithms and dynamic priority scheduling algorithms [4]. A typical example of fixed priority scheduling algorithms is Rate Monotonic (RM) scheduling; while a typical example of dynamical scheduling algorithms is Earliest Deadline First (EDF) scheduling.

Figure 4(a) shows the schedule of two real-time tasks under RM scheduling. Task τ_1's timing parameters (in milliseconds) are (C_1, T_1)=$(1, 2)$, and task τ_2's timing parameters are (C_2, T_2)=$(1, 8)$. Under RM, tasks with smaller periods (i.e. higher rates) have higher priorities. So in this example, τ_1 has higher priority than τ_2. From the figure, we can see that under RM scheduling, all jobs (instances of each control task) are schedulable, i.e., able to meet their deadlines.

HPCs may contain faults. Faulty controller thread will cause controller task to miss deadlines or even fail, lead to undesirable consequences such as instability, data losses, or performance degradation. In ORTGA, if a fault in the HPC is identified, the HPC needs to be killed and the new controller thread (HAC) will be created to substitute it to maintain system safety.

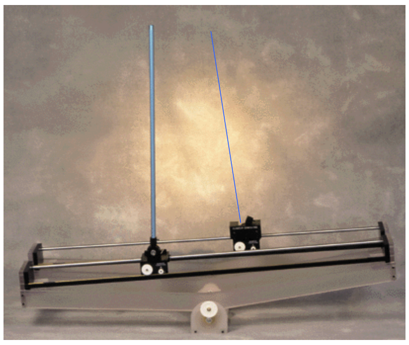

Figure 3: Double inverted pendulums running on a machine

3.2 Research Issues in ORTGA

The first design issue faced by ORTGA is what mechanism to use for online fault detection. Software faults in real-time systems can be categorized into two classes: logical domain faults, and execution domain faults. The former are usually caused by the logic of the underlying algorithm itself, which defines the computational logic. The latter are caused by various faults within the software other than algorithm logic, such as memory leakage, segmentation fault, divide by zero, spin in an infinite loop, deadlock, and live lock etc. The approaches to deal with logical domain faults are more of an algorithm design issue than a fault containment and tolerance issue. Therefore, in ORTGA we target our goal at the tolerance of execution domain software faults. Noticing that a common symptom of execution domain faults is that no system output is given within the task deadline, so we can use a *heartbeat* message mechanism to detect execution domain faults in real-time systems. In the heartbeat message mechanism, each controller thread sends out a brief message to a monitor thread to indicate its healthiness soon after it sends out its control value in each period. When no heartbeat message is received within a time period for a thread, it is possible that the thread has execution domain faults. On the other hand, if the heartbeat message arrives in a timely fashion, we know that there is no execution domain faults in the thread.

Besides the advantage of larger fault coverage, comparing to other approaches, Heartbeat message fault detection mechanism has the advantage of easy implementation. It is also non-intrusive, since no modification is needed in the OS kernel.

Given the heartbeat message fault detection mechanism, there are three important research issues to be further addressed in ORTGA.

Q1. How to treat false alarms when detecting faults of a controller thread?

For real-time control tasks, it is shown that usually several deadline misses can be tolerated without causing fatal problems such as instability [5]. What's more, even if a controller thread misses one heartbeat message, it may still be healthy, since other possibilities including temporary communication channel failure could exist. Considering these two factors, the monitor should not be designed too "aggressively" to identify a controller thread as faulty. One or more missing heartbeat messages may be allowed, otherwise false alarms could occur. False alarms should be avoided since they cause unnecessary recovery procedures, which may degrade the system performance, or even cause unschedulability. This is because unnecessary recoveries affect the schedules of other tasks running on the same CPU.

Q2. When a controller thread CT_i is identified as faulty and a recovery decision is made, when should the recovery procedure be started?

In generic software systems, when a fault is identified, the common solution is to recover the faulty component as soon as possible to minimize the performance loss. For example, the Recovery Oriented Computing (ROC) [6] fault tolerance approach aims to minimize Mean Time To Repair (MTTR). However, this wisdom may not be true in real-time systems. To understand this, we show the following example as illustrated in Figure 4.

In this example, the timing parameters of the two real-time tasks are (C_1, T_1)=$(1, 2)$, (C_2, T_2)=$(1, 8)$. These two tasks are schedulable under RM scheduling, as shown in Figure 4(a). Now suppose task τ_2 is identified as faulty by the monitor at time $t = 2.0$, as shown Figure 4(b). Suppose the overhead of killing the faulty thread (HPC) and replacing it with a new thread (HAC) takes 1.5 milliseconds. If the recovery procedure takes higher priority (i.e. to recover τ_2 as soon as possible), the recovery procedure is started immediately at $t = 2.0$. The recovery procedure will finish at $t = 3.5$. Now τ_1's second job begins execution at time $t = 3.5$, and will miss its deadline, which is 4.0. However, if we delay τ_2's recovery and let τ_1's second job begin execution at its release time $t = 2.0$ (as shown in Figure 4(c)), then the recovery of τ_2 begins at $t = 3.0$ and the recovery is finished at $t = 4.5$. As a result, no deadline will be missed for both tasks.

From this example, we clearly see that the determination of the **right time to recover** is crucial to guarantee schedulability in fault tolerant real-time systems. Sometimes, a "late" but timely recovery is more beneficial for all the tasks in the system to meet their timing requirements. Comparing to the objective of minimizing MTTR in ROC, one central problem of ORTGA is to determine RTTR (Right Time To Recover).

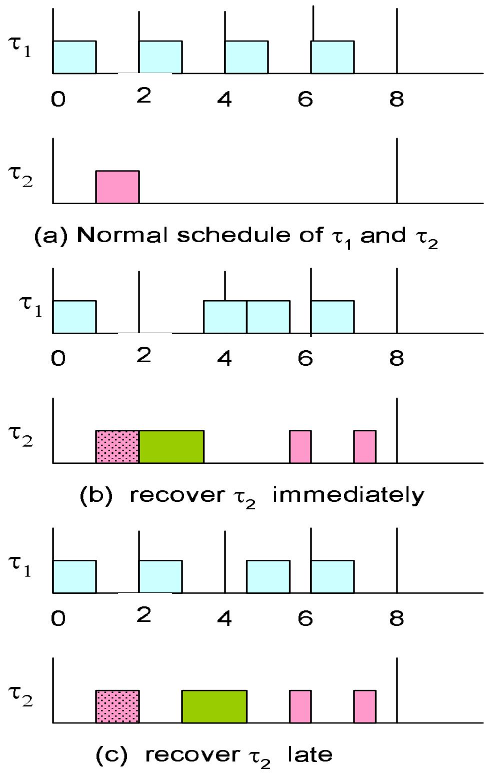

Figure 4: Illustration of a late recovery is more desirable.

Q3. What is the impact of the recovery procedure and the recovered controller thread to the schedulability and performance of the whole system?

From the example given in Q2, we know that when a controller thread needs to be recovered, the recovery procedure may affect the schedulability of whole task set.

In addition to that, in order to recover from the damage of the faulty controller thread, the recovered controller thread (i.e. the new controller thread who replaces the previous faulty controller thread) usually needs to carry out more complex control computations (such as Kalman filtering etc[1]). Also, the recovered controller thread usually runs at a faster rate (i.e. shorter sensing-control loop period). Let us denote the recovered controller task for faulty controller thread i as τ_{ir}, with execution time C_{ir} and period T_{ir}. Usually we have $C_{ir} \geq C_i$ and $T_{ir} < T_i$.

However, the new task set $\{\tau_1, \ldots, \tau_{i-1}, \tau_{ir}, \tau_{i+1}, \ldots, \tau_N\}$ may not be schedulable any more, since the CPU bandwidth occupied by τ_{ir} is larger than that of τ_i. This requires the adjustment of the timing parameters of other (healthy) controller tasks. A common solution is to back off other controller tasks (i.e., decrease their sensing-control loop rates) to accommodate the newly released recovered controller task. How to adjust the sensing-control loop rate for each task in order for the new task set to meet schedulability constraint is one research issue.

Another impact of the recovered controller task to the other controller tasks is control performance loss. A controller's control performance depends on the sensing-control loop rate of the controller task [8]. Since the other controller tasks may have to back off to guarantee the schedulability of the whole task set, the system's overall control performance will be affected when such back off happens. When designing the recovered controller and determining its timing parameters, we also need to trade off between the overall system control performance loss and the task set schedulability affected.

4 Some Possible Solution Directions

As discussed in Section 3, the design of ORTGA raises some interesting research problems. In this section, we will briefly discuss some possible solutions to these problems. Due to space limit, the discussions here are rather incomplete and preliminary than thorough. Our motivation is trying to identify some possible directions for future research of feedback based fault tolerant real-time systems.

First, it is easy to see that Question 1 (minimize false alarms in faulty controller thread detection) and Question 2 (when to recover a faulty thread) are closely related. On one hand, in order to minimize false alarms, the monitor should not treat heartbeat message misses too aggressively. On the other hand, the decision of when to recover highly depends on fault detection. If a fault is detected too late, the recovery procedure may not have enough time to react, hence the recovery may fail. Moreover, as shown in Figure 4, even if a fault is detected early, sometimes a late but timely recovery may be more desirable since it helps to maintain system schedulability. So the core question here in terms of fault detection and recovery is what is the right time to treat a thread (who has missed heartbeat messages) as faulty and recover it. A too early decision may increase false alarms and affect system schedulability unnecessarily, while a too late decision will delay the recovery and lead to system failure.

Here, we propose a possible solution which treats the fault detection and recovery timing issues in one framework. The idea is to solve the two problems together in a time reversal fashion. Figure 5 shows a timeline of the proposed recovery framework.

Suppose the execution of recovery procedure takes a time overhead of t_s. t_s includes the time of killing the previous faulty HPC thread and the time of replacing it with the new HAC thread. Our minimum goal is to ensure that after the recovery, the system being controlled will still be stable. First, we determine the stability region of the controlled system. The stability region is defined as a set of plant states, outside which the system will not be stable anymore under the current controller. A stability region of a controlled system under a specific controller can be determined using Linear Matrix Inequality [2]. Due to space limit, we omit the details here. Interested readers are referred to [7]. The determined stability region of the controlled system under the HAC controller is shown by the shaded ellipsoid S in Figure 5. From this scenario, we see that for controller task τ (corresponding to the case when HPC is running), heartbeat messages (HB1, HB2) were received by the monitor at time t_1 and t_2, which indicates the HPC was healthy. Along with the heartbeat messages, updated plant states are also sent to the monitor. However, at time t_3, the 3rd heartbeat message was still not received. The monitor then needs to decide whether the HPC should be identified as faulty and if so, when the recovery procedure should be launched.

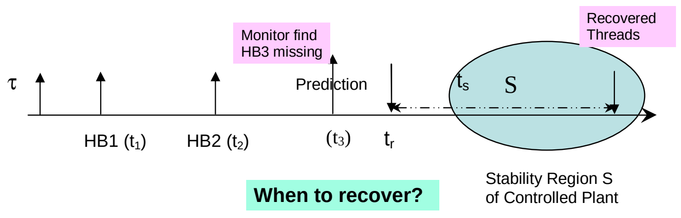

Figure 5: A detection and recovery decision framework.

After we have determined the HAC-established stability region S of the system, we can find the latest time point t_r satisfying the following criterion – the system state will still be within the HAC-established stability region after the recovery procedure is completed, only if the recovery procedure is scheduled by time t_r. We use notation $x(t)$ to represent the state of the plant at time t. t_r is determined such that $x(t_r + t_s) \in S$ under the current HPC control. Therefore t_r is the latest time when the recovery procedure should begin to maintain the plant stable. In this way, if until time t_r, no new heartbeat message is received by the monitor from this HPC thread, the recovery procedure has to be kicked in.

With respect to Question 3 (schedulability and control performance impacts of the recovered controller thread), we can carry out offline real-time schedulability analysis [4] or apply online scheduling algorithms such as elastic scheduling [3] to guarantee all tasks meeting their deadlines during and after the recovery procedure is executed. To determine the impact of recovery procedure and recovered controller thread on overall system control performance, we may be able to use online load-driven scheduling algorithm to optimize the control performance [9] during and after the recovery.

When a recovered controller runs an extended period of time, it can be switched back to the original controller to improve overall system control performance. How to determine when to switch back is raised in Question 3. A possible solution is still using stability region as follows. Suppose the switch back time overhead is t_b, which includes the time of killing the current recovered controller thread and the time of switching back to the original controller thread. At any time t, when the monitor gets the heartbeat message of the recovered controller, along with its current plant state information $x(t)$, the monitor then calculates if $x(t + t_b)$ is already within stability region of the plant under the original controller. If so, the monitor can initiate the switching back procedure. This approach guarantees the stability of the system.

5 Related Work

Recovery-Oriented Computing (ROC) [6] is an approach for general fault tolerant systems. A major goal of ROC is to recover (by methods such as reboot or micro-reboot) the system as soon as possible when a fault has occurred, i.e. minimizing the MTTR (Mean Time To Repair) rather than maximizing MTTF (Mean Time To Failure). Hence ROC offers high availability[1]. ORTGA differs from ROC in two aspects. First, when the HPC is diagnosed as faulty, the recovered controller (HAC) is not a simple restart of the HPC. In stead, HAC is a predetermined reliable core controller which guarantees to make the plant stable[2]. Secondly, as we showed in Section 3.2, in real-time systems a *late* but *timely* recovery may be more desirable in some situations. Hence one fundamental research issue of ORTGA is to determine RTTR (Right Time To Recover) instead of minimizing MTTR.

Simplex [7] is a software architecture which facilitates the building of dependable real-time control systems. It provides dynamic toleration of software faults. In Simplex, analytical redundant high-assurance controller (HAC) runs in parallel with high-performance controller (HPC). This unnecessarily lowers the total CPU utilization available to other active tasks when no fault occurs. This drawback keeps the application of Simplex from those industrial applications where both an efficient resource utilization and a high fault coverage are desired. ORTGA solves this problem by running the HAC in an on-demand fashion. Hence it achieves much higher resource utilization.

6 Conclusions

In this paper, we propose ORTGA, a feedback based fault tolerance architecture for real-time systems. We discuss the advantages and benefits of using ORTGA for fault tolerance in real-time systems compared with other existing approaches. Most importantly, we list research issues faced by ORTGA, many of which are raised by the timing, performance and stability requirements of real-time systems. These requirements are not common in generic software systems, hence make the design of ORTGA quite different from traditional ROC. We also point out possible directions for finding the solutions to these problems.

We hope this paper will arouse the interests of researchers and foster the discussions in using feedback based fault tolerance for computing systems.

[1] Recall that availability is traditionally defined as the ratio of MTTF to MTTF + MTTR.

[2] Of course, in some application scenarios, we can use the restarted HPC as the HAC under ORTGA architecture.

Acknowledgment

This material is based upon work supported by the NSF CCR 02-09202, by ONR N00014-02-1-0102, and by MURI N00014-01-0576. The first author is also supported in part by the Mavis Memorial Fund Award. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] K. J. Astrom and B. Wittenmark. *Computer-Controlled Systems: Theory and Design, 3rd edition*. Addison-Wesley Pub Co., 1994.

[2] S. Boyd, L. E. Ghaoui, E. Feron, and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*. Society for Industrial and Applied Mathematics (SIAM), 1994.

[3] G. Buttazzo, C. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[4] J. Liu. *Real-Time Systems*. Prentice Hall PTR, 2000.

[5] P. Marti, R. Villa, J. Fuertes, and G. Fohler. On real-time control tasks schedulability. In *European Control Conference*, 2001.

[6] D. A. Patterson et al. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 2002.

[7] D. Seto, B. H. Krogh, L. Sha, and A. Chutinan. Dynamic control system upgrade using the simplex architecture. *IEEE Control System Magazine*, 1998.

[8] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control system. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 13–21, 1996.

[9] L. Sha, X. Liu, M. Caccamo, and G. Buttazzo. Online control optimization using load driven scheduling. In *Conference on Decision and Control*, Sydney, Australia, 2000.