

# ORTEGA: An Efficient and Flexible Software Fault Tolerance Architecture for Real-Time Control Systems

Xue Liu <sup>\*</sup>, Hui Ding <sup>†</sup>, Kihwal Lee <sup>‡</sup>, Qixin Wang <sup>‡</sup>, Lui Sha <sup>‡</sup>

<sup>\*</sup> School of Computer Science, McGill University

Email: xueliu@cs.mcgill.ca

<sup>†</sup> Mortgage Core Modeling Group, UBS Investment Bank

Email: hui.ding@ubs.com

<sup>‡</sup>Department of Computer Science, University of Illinois at Urbana-Champaign

Email: {klee7, qwang4, lrs}@uiuc.edu

## Abstract

*Fault tolerance is an important aspect in real-time computing. In real-time control systems, tasks could be faulty due to various reasons. Faulty tasks may compromise the performance and safety of the whole system and even cause disastrous consequences. In this paper, we describe ORTEGA (On-demand Real-TimE GuArd), a new software fault tolerance architecture for real-time control systems. ORTEGA has high fault coverage and reliability. Compared with existing real-time fault tolerance architectures, such as Simplex, ORTEGA allows more efficient resource utilizations and enhances flexibility. These advantages are achieved through the on-demand detection and recovery of faulty tasks. ORTEGA is applicable to most industrial control applications where both efficient resource usage and high fault coverage are desired.*

## 1 Introduction

Real-time and embedded systems now play an important role in our lives, with products covering a large variety of markets, such as aerospace, communication systems, automobiles, healthcare, and personal electronics. Real-time and embedded systems research is regarded as one of the next IT (Information Technology) frontier [1].

A real-time system has well-defined timing constraints. Different from general purpose computer systems, a real-time system is considered to function correctly only if it returns the correct result within the system-wide timing constraints [31, 7].

Reliable functioning of real-time systems is of paramount concern to the millions of users that depend on these systems everyday. However, fault and

failures can occur in real-time systems. Failures can be caused by hardware malfunctions and/or faults (e.g., electro-mechanical device), software faults (e.g., the processes/threads running on a computer), or communication medium faults.

Hardware and communication medium faults are typically tolerated by hardware redundancy [8] and techniques such as message buffering [9]. In this paper, we focus on how to tolerate software faults in real-time control systems.

Real-time control system software faults can be categorized along three dimensions [28]:

1. Resource sharing faults: The corruption of other module's code and data;
2. Timing faults: failure to meet timing constraints;
3. Semantic faults: producing wrong values.

Simplex [28, 25] is a software architecture which facilitates the building of dependable real-time control systems. It provides dynamic toleration of software faults. In Simplex, resource sharing faults are handled by address space protections. Timing faults are handled through real-time scheduling methods such as Generalized Rate-Monotonic Scheduling (GRMS) [30]. Semantic faults are handled through *Analytical Redundancy* by running redundant high-assurance controller to guard the system. Simplex has been successfully used in applications such as automated aircraft maneuvering [24] and semiconductor wafer-making [25].

However, Simplex fault tolerance architecture has two major drawbacks:

**1. Lack of efficiency:** In Simplex, the analytical redundant high-assurance controller runs in parallel with high-performance controller even when there are no faults. This wastes CPU cycles. In well-tested industrial applications, failures are infrequent. A parallel high-assurance controller

nearly doubles the CPU execution time for a single controlled system. This makes Simplex a high-cost scheme, and hampers its application to many real-time embedded systems, which are resource-constrained.

**2. Lack of flexibility:** Simplex enforces the same execution period on the high-assurance controller and the high-performance controller. This simplifies the real-time scheduling analysis of systems running under Simplex. However, a control system's performance is affected by the sampling/control periods<sup>1</sup> used [26, 29]. As a result, in practice, different controllers may use different periods for different performance considerations. For example, when a fault occurs in the high-performance controller, ideally the system designer prefers to run the high-assurance controller at a faster rate to help recover from the fault and protect the system promptly. Unfortunately, this design is not possible under Simplex because it lacks the flexibility to allow the high-assurance controller and the high-performance controller to run at different periods.

In this paper, we present a new fault tolerance architecture called ORTEGA (On-demand Real-Time Guard). ORTEGA maintains high fault coverage and reliability as the original Simplex, meanwhile significantly improves the flexibility and resource utilization efficiency. Hence ORTEGA is applicable to a wider range of real-time control systems.

In ORTEGA, the high-assurance controller is running in an on-demand fashion. Only when a fault is detected, will the high-assurance controller be activated to replace the faulty high-performance controller. Since only one controller is active to control the system at any time instant, much resource is saved. Through careful design and schedulability analysis, ORTEGA also allows the high-assurance controller and high-performance controller to run at different rates, which greatly facilitates the flexibility in control and fault tolerance design. We implemented ORTEGA on a real-time inverted pendulum control system and carried out extensive evaluations. Results confirm the effectiveness and efficacy of ORTEGA.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the ORTEGA architecture. Section 4 discusses in detail several design challenges related to ORTEGA and presents our solutions. Section 5 describes the implementation and evaluations of ORTEGA. Finally, Section 6 summarizes the paper and points out future research directions.

## 2 Related Work and Backgrounds

Fault tolerance is always an important issue in software systems [14]. There are well-developed fault tolerance tech-

---

<sup>1</sup>In this paper, we refer "sampling/control period" as "period" for brevity.

niques for general software systems. They can be classified into three general categories: fault masking (e.g., N-version programming [3]); backward fault recovery (e.g., recovery blocks [19] and/or checkpointing techniques in transaction-based systems [18]); and forward fault recovery (e.g., roll-forward checkpointing scheme in [17]).

However, none of the above-mentioned general fault tolerance schemes are readily applicable to real-time systems. For real-time systems, as discussed in Section 1, software faults can be categorized along three dimensions: 1) resource sharing faults; 2) timing faults; and 3) semantic faults. The general fault tolerance schemes above do not sufficiently take timing faults into consideration. What is more, how to *timely* recover the system from faults is an important research problem for real-time systems and has not been fully addressed in the general fault tolerance schemes.

Fault tolerance for real-time systems is an active research topic in the past decades. A comprehensive review of the recent progresses in this area can be found in [15, 10]. Here we only list the research which is most relevant to our work.

The researchers of the FORTS project [4, 5] from University of Pittsburgh address the CPU scheduling of recovery jobs in real-time systems. They assume that all faults can be detected at the end of each periodic execution of the job, and the recovery is done by re-executing the job before its deadline. This is effective for faulty controllers in which faults are non-recurrent. However, for control systems in which faults are recurrent or persistent, usually a recovery cannot be done by re-executing the same job within the same period. This is because most probably the re-execution will lead to the same fault again since it is still executed within the same faulty controller process. If we replace the faulty controller process with a higher-assurance controller process, the fault could be safely removed. A process replacement is different from a job re-execution and brings up many interesting research issues.

Lee et al. [11] develops a technique called *Process Resurrection* to recover a process from crash failure to meet the real-time requirements. Process Resurrection is tightly coupled with the crash detection mechanism of the underlying operating system, which offers signals as event notification mechanism. Common error notification signals include SIGSEGV (segmentation faults), SIGBUS (bus error), SIGFPE (arithmetic operation error such as divide by zero), and SIGILL (execution of an illegal instruction). In Process Resurrection, a special signal handler is assigned for every crash related signals in order to override the default signal handler and trigger the recovery. However the fault coverage of this technique is limited to process crashes. Faults such as deadline missing, infinite loop, or deadlock cannot be handled.

Simplex [28, 25] is a software architecture which facilitates the building of dependable real-time systems. It pro-

vides dynamic toleration of system faults. The plant under control 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 proven 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. In Simplex, the HAC and HPC subsystems run in parallel, but the software stays in separate processes. By using the redundant HAC to guard against possible faults in the HPC, Simplex achieves fault tolerance.

However, as we discussed in the Introduction section, Simplex is not resource-efficient and not flexible. In comparison, ORTEGA achieves the same fault coverage and functionalities as Simplex, but with significantly lower CPU resource usage and allows flexible controller implementations.

### 3 ORTEGA Architecture

In this section, we present the overall architecture and design considerations of ORTEGA.

#### 3.1 Components Organization and Fault Recovery Procedure of ORTEGA

The architecture of ORTEGA is shown in Figure 1. We call a plant for which ORTEGA provides fault tolerance protection an **FT-enabled plant**. In ORTEGA, there can be multiple FT-enabled plants. For each FT-enabled plant, there are three ORTEGA logical components: 1) a decision module; 2) a high-performance controller (HPC) module; and 3) a high-assurance controller (HAC) module. Figure 1 illustrates the case where there is one FT-enabled plant.

Similar to the Simplex architecture [25], the HAC subsystem is highly reliable but only provides basic performance. The HPC subsystem complements the conservative HAC core, but may contain faulty software components.

For both Simplex and ORTEGA, the decision module plays a key role in providing fault detection and recovery using its decision logic. In Simplex, at any time, both the

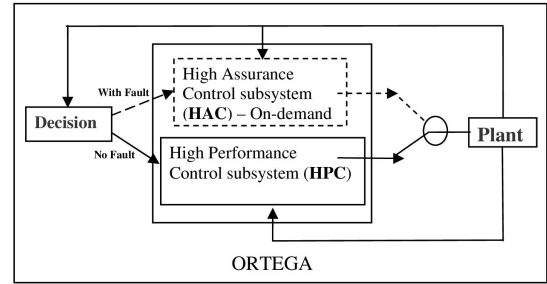


Figure 1. Illustration of the ORTEGA fault tolerance architecture.

HPC and the HAC are running; the decision module determines which control command among the two to be used for online control in each period. In contrast, ORTEGA runs the HAC only when it is necessary, i.e., only when the decision module detects an HPC fault/failure. As a result, *only one* instance of either the HAC or the HPC is running at any time. By eliminating the redundant execution of controllers, ORTEGA's run-time overhead is significantly less than that of Simplex. The saved CPU cycles can be used for other real-time or non-real-time tasks. It is worth noting that the on-demand execution can be implemented efficiently under ORTEGA to remove the potential overhead (in terms of delay) associated with controller thread/process start and stop.

In our implementation, the decision module uses a mutex semaphore to control which of the HAC or HPC is running. When the HPC is running well, the HAC thread simply blocks on the semaphore, i.e. the HAC is suspended. Only when a fault is detected in the HPC, the decision module will release the semaphore to let the HAC become active.

In ORTEGA, the HPC controls the plant during most of the time. In order to detect and recover HPC faults in a timely manner, the decision module should ensure that the HPC-controlled plant state stays within the HAC-established stability region. A method to determine stability regions for digital controllers will be presented in Section 4.1. When this condition is violated, the suspended HAC will be activated and takes over the plant to recover.

A second feature of ORTEGA is that the HAC and the HPC can run at different sampling rates. This allows better flexibility in designing the HAC and the HPC. For example, when the HPC is detected faulty, the carefully designed HAC can run at a faster rate to help recover the plant promptly.

In summary, the realization of ORTEGA is described as follows. On system start-up, all components are started but the HAC is blocked. As soon as a fault is detected in the HPC, the decision module deactivates the HPC and acti-

vates the HAC. Now the plant is under the control of HAC for recovery. If the type of fault is semantic (e.g. control command out of stability region), the HPC is allowed to be switched back later, after the HAC recovers the system from the previous fault and stabilizes the plant. If the failure is due to an execution error such as segmentation fault or entering infinite loop, the HPC will be restarted for later retry.

After the HAC has recovered and stabilized the plant, the active controller will switch back to the HPC for retry or performance considerations. Note unlike the recovery procedure, which must be done in a timely fashion in order for the plant state to stay within the HAC-established stability region<sup>2</sup>, the initiation time of the switch-back can be more flexible. In practice, the decision module can initiate the switch-back when the plant state is near the control set point and the CPU is in an idle interval[32]. This mechanism avoids the possible mode-change problem and simplifies schedulability analysis. We will discuss the details of schedulability analysis in Section 4.2.

### 3.2 Analysis of CPU Usage Savings of ORTEGA

ORTEGA eliminates the redundant execution of controllers hence reduce the resource usage significantly. In this section, we quantify the resource savings of ORTEGA in terms of CPU usage.

We use the commonly adopted periodic task model [12] to model the control tasks. Each periodic task is denoted by  $\tau_i$ . The *timing parameters* of a task  $\tau_i$  is represented by the tuple  $(C_i, T_i)$ , where  $C_i$  is the worst-case execution time, and  $T_i$  is the task period. For control systems, a task's deadline is usually the same as its period, i.e., we have  $D_i = T_i$ .

Compared with Simplex, ORTEGA greatly reduces CPU resource usage due to the on-demand execution of the HAC. Suppose the timing parameters of an FT-enabled plant under the control of its HPC is  $(C^p, T^p)$ , while the timing parameters of the same plant under the control of its HAC is  $(C^a, T^a)$ . We use  $P_r$  to denote the percentage of time used for recovery (i.e., when HAC is active) during a total run time of  $T$  (in the unit of milliseconds).

In the original Simplex, the total CPU resource usage (in the unit of milliseconds) is:

$$R_{Simplex} = (1 - P_r) \cdot \left( C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil + C^p \cdot \left\lceil \frac{T}{T^p} \right\rceil \right) + P_r \cdot C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil. \quad (1)$$

While in ORTEGA, the total CPU resource usage (in the unit of milliseconds) is:

$$R_{ORTEGA} = (1 - P_r) \cdot C^p \cdot \left\lceil \frac{T}{T^p} \right\rceil + P_r \cdot C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil. \quad (2)$$

<sup>2</sup>Schedulability analysis of the controller switching will be discussed in Section 4.2.

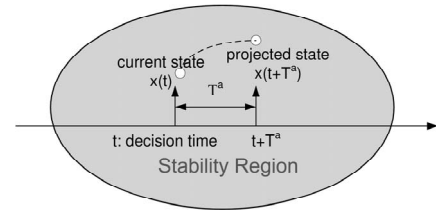
As we can see, ORTEGA saves  $(1 - P_r) \cdot C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil$  (msec) CPU usage during a total run time of  $T$  (msec). In practical industrial applications where faults are rare,  $P_r$  is very small, thus ORTEGA saves much of the CPU resource.

### 3.3 Extra One Period Delay

The design of the on-demand execution of HAC in ORTEGA saves much of the CPU resource. However, we note due to the on-demand recovery, ORTEGA may introduce up to one period delay in the recovery procedure. In this section, we detail the cause of the delay and provide a solution.

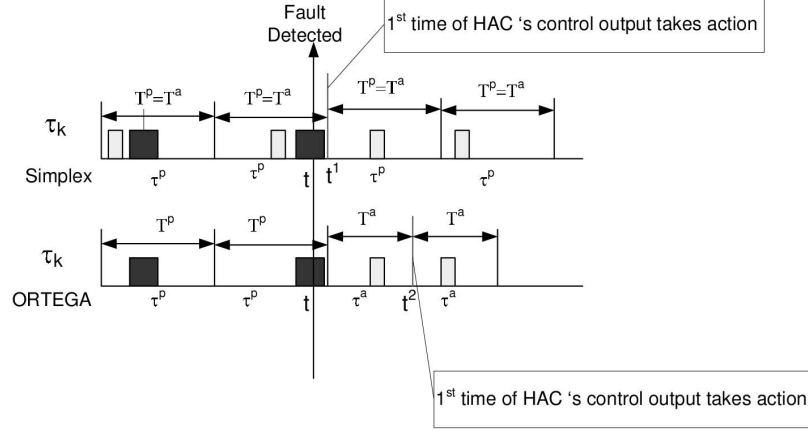
Figure 2 illustrates this extra delay. Suppose at time  $t$ , a fault is detected in the HPC. The upper half of Figure 2 shows the recovery timeline under the original Simplex, while the lower half of Figure 2 shows the recovery timeline under ORTEGA. For Simplex, since the HAC and the HPC are running in parallel, on detection of a fault, the HAC control command is immediately available and can take effect at the beginning of the next control period (i.e. at  $t=t^1$ ). For ORTEGA, the HAC is running on-demand. On detection of a fault, the HAC needs to carry out the control computation and its control command only becomes available during the first period when the HAC begins running. So the HAC control command takes effect one period ( $T^a$ ) later, i.e., at time  $t=t^2$ . As a result, compared with the recovery procedure using Simplex, the recovery procedure using ORTEGA will incur an extra delay up to  $t^2 - t^1 = T^a$ .

The extra delay incurred in ORTEGA can be *compensated* using state projection technique. The idea is illustrated in Figure 3. At any decision time  $t$ , the decision module projects the plant state one HAC period ahead, i.e., projects  $x(t + T^a)$ . If the projected state is still within the stability region associated with the HAC, the HPC can still run; otherwise, the HAC will be activated to take over the plant.



**Figure 3. Illustration of using state projection to handle the extra delay.**

It is worth noting that the flexibility of ORTEGA allows the period of the HAC ( $T^a$ ) to be smaller than that of the HPC ( $T^p$ ) for fast recovery. Hence the potential impact of the extra delay is small compared with the bene-



**Figure 2. Illustration of the extra delay in recovery using ORTEGA.**

fits gained in the increased stability region of the HAC in ORTEGA. The increased stability region allows for better fault-tolerance. This is discussed in Section 4.1. A simple and computationally-inexpensive state projection method is also presented in Section 4.1.

## 4 Design Challenges and Solutions

The design and implementation of ORTEGA raises several important research challenges. In this section, we discuss two most important challenges and present their solutions. The first challenge is how to determine the HAC stability region, which is used by the decision module for fault detection and recovery. The other is how to carry out schedulability analysis for ORTEGA.

### 4.1 Maximum Stability Region for Digital Controllers

ORTEGA's forward recovery scheme is based on the maximum stability region of the plant under the HAC controller. At any decision time, the decision module will check if the plant's (projected) state is still within the stability region associated with the HAC. If so, the HPC controls the plant; otherwise, the HAC will be activated to take over the control. In order to minimize the unduly restriction of the state space the HPC can use, we prefer a large stability region associated with the HAC. In this section, we propose a formal approach to determine the maximum stability region for a digitally-implemented control system.

Assume the plant to be controlled is governed by a continuous-time state space model as described below:

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t), \quad (3)$$

where  $x \in \mathcal{R}^n$  is the system state and  $u \in \mathcal{R}^m$  is the control input.  $A \in \mathcal{R}^{n \times n}$ ,  $B \in \mathcal{R}^{n \times m}$  are the corresponding system matrices. Controllers are typically designed in a state feedback form, i.e.  $u(t) = -Kx(t)$ , where  $K$  is the corresponding controller gain.

Modern control systems are typically implemented on digital computers. Due to the digitization of continuous controllers, the sampling and control of the continuous-time system (3) is enforced at discrete time points. As a result, for the purpose of design and analysis, we need to convert the continuous-time system to its discrete-time form according to the digital implementation method used. For example, the continuous-time system (3), when implemented using a zero-order hold with a sampling period  $h$ , can be represented as follow [2]:

$$x(k+1) = Fx(k) + Gu(k), \quad (4)$$

where  $x(k) \triangleq x(kh)$  is the state of the plant at the  $k$ th sample time, and

$$F = e^{Ah}, \quad G = \left( \int_0^h e^{As} ds \right) B. \quad (5)$$

Using Equations (4) and (5), we can get a simple and computationally-inexpensive state projection method to compensate the extra delay when using ORTEGA as discussed in Section 3.3. That is, at control interval  $k$ , since the decision module knows the current plant state  $x(k)$  and the current control output value  $u(k)$ , it can calculate the predicted plant state at interval  $k+1$  according to Equations (4) and (5).

Corresponding to the continuous-time state feedback controller  $u(t) = -Kx(t)$ , the digitally-implemented state

feedback controller is  $u(k) = -Kx(k)$ . By replacing the  $u(k)$  term with  $-Kx(k)$  in the discrete-time system equation (4), we get the closed-loop discretized control system as

$$x(k+1) = \bar{F}x(k), \quad (6)$$

where  $\bar{F} = F - GK$ .

To determine the stability of a closed-loop discretized control system as (6), we use the celebrated Lyapunov stability criteria which is summarized in the following theorem [2].

**Theorem 1** *A discrete-time linear time-invariant (LTI) system (6) is stable iff there exists a positive definite matrix  $P > 0$  such that*

$$\bar{F}^T P \bar{F} - P < 0. \quad (7)$$

For a real-life application, due to the limitation on physical plant and control actuators, there are constraints on the system states and/or control inputs. For system (4), these constraints can be represented as:

$$a_i^T x \leq 1, \quad i = 1 \cdots l, \quad (8)$$

$$b_j^T u \leq 1, \quad j = 1 \cdots r. \quad (9)$$

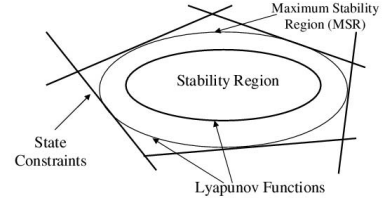
With digital controller  $u(k) = -Kx(k)$ , the constraints (8)(9) can be combined and represented as a polytope (a multi-dimensional figure whose faces are hyperplanes) in the system's state space:

$$\alpha_m^T x \leq 1, \quad m = 1, \cdots, q, \quad (10)$$

where  $\alpha_m^T = a_m^T$ , for  $m = 1, \cdots, l$ ;  $\alpha_m^T = b_j^T K$ , for  $m = l + 1, \cdots, q$ ,  $j = 1, \cdots, r$ , and  $q = l + r$ . The states inside the polytope are called *admissible states*, because they obey the operational constraints.

We formally define a stability region as a subset of the states within the polytope such that if the closed-loop discretized control system starts from a state within the stability region, the system states' future trajectory will always stay within the region and finally converge to the control set point.

From Lyapunov theory, we see a Lyapunov function  $x^T P x$  inside the state constraint polytope represents a stability region [25][23][22]. Geometrically, it defines an  $n$ -dimensional ellipsoid in the  $n$ -dimensional system state constraint polytope, as illustrated in Figure 4, where the state space is 2-dimensional. An important property of a Lyapunov function is: if the system state is within the ellipsoid associated with a controller, it will always stay within the ellipsoid and finally converge to the equilibrium position (set point) under this controller.



**Figure 4. Illustration of a stability region under state constraints.**

Mathematically, for a closed-loop discretized control system (6), a *stability region* under a specific Lyapunov matrix  $P$  with state constraints is defined as the following ellipsoid:

$$S_P \triangleq \{x | x^T P x < 1\}, \quad (11)$$

where  $P$  satisfies the Lyapunov stability criteria  $\bar{F}^T P \bar{F} - P < 0$  (i.e., Equation (7)) and  $x$  satisfies state constraints (i.e. Equation (10)).

However, a Lyapunov matrix  $P$  is not unique for a given stable closed-loop discretized control system. As we discussed, in order not to unduly restrict the state space within the operational constraints, we should find the maximum stability region (MSR). To get the MSR, we first give Lemma 1.

**Lemma 1** *Given a discretized LTI system (6) with state constraints (10), the stability region  $S_P$  defined in (11) satisfies the constraints in (10) iff  $\alpha_m^T P^{-1} \alpha_m \leq 1$ ,  $m = 1, \cdots, q$ .*

*Proof:* Please refer to [27] (Lemma 4.1). ■

Note that the area of the stability region defined in (11) is proportional to the determinant of matrix  $P^{-1}$ . Based on Lemma 1, the determination of the MSR of a closed-loop discretized control system (6) with state constraints (10) is then reduced to the following Linear Matrix Inequality (LMI) problem [6].

**Problem 1**      *Maximize*     $\log \det P^{-1}$

*s.t. :*                       $P > 0,$

$\bar{F}^T P \bar{F} - P < 0,$

$\alpha_m^T P^{-1} \alpha_m \leq 1, \quad m = 1, \cdots, q.$

Further, if  $\bar{F}^{-1}$  exists and let  $Q \triangleq P^{-1}$ , we convert the above LMI problem as the following new problem.

**Problem 2** Maximize  $\log \det Q$

$$\begin{aligned} s.t. : \quad & Q > 0, \\ & Q\bar{F}^T - \bar{F}^{-1}Q < 0, \\ & \alpha_m^T Q \alpha_m \leq 1, \quad m = 1, \dots, q. \end{aligned}$$

Problem 2 is a MAXDET problem [6]. It is readily-solvable using numerical software packages such as *sdp sol* [33] or *YALMIP* [13]. Note that the maximum stability region and its solution via LMI formulations (i.e. Problems 1 and 2) are different from those presented in [27]. Here we are dealing with discretized system under digital controllers, while in [27], the authors were dealing with continuous system under continuous-time controllers.

The resulting  $P = Q^{-1}$  from the MAXDET problem above defines the maximum stability region  $\{x | x^T P x < 1\}$  in the system state polytope (see Figure 4).

Using the method presented above, when the continuous-time system model, the underlying continuous-time controller and its control loop period are given, a system designer can calculate the maximum stability region of the corresponding closed-loop discretized control system offline.

It is obvious that the maximum stability region (MSR) is a function of the control loop period  $T$ . Hence we denote the MSR for a plant under a digital controller with respect to the control loop period  $T$  as  $MSR(T)$ . We further use  $A(T)$  to denote the area of  $MSR(T)$ . Since  $MSR(T)$  encloses the admissible states of the system to keep it stable with respect to the control loop period  $T$ , we call  $A(T)$  the **stability index** of the closed-loop discretized control system under control loop period  $T$ .

Now, let's look at a real-life control example. Consider a controlled inverted pendulum. The continuous-time system model of the inverted pendulum is shown below:

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t), \quad (12)$$

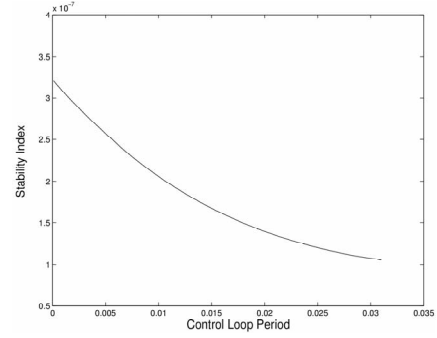
where the system matrices are:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -2.7528 & -10.9526 & 0.0043 \\ 0 & 28.5812 & 24.9179 & -0.0441 \end{pmatrix}, \quad (13)$$

$$B = (0, 0, 9432, -4.4385)^T. \quad (14)$$

The designed high-assurance controller for the inverted pendulum uses state feedback control [2] as shown below:

$$u(t) = -(-5.7807, -42.2087, -14.0953, -8.6016) x(t). \quad (15)$$



**Figure 5. Stability index v.s. control loop period for the inverted pendulum control system.**

Here the plant state is represented as  $x(t) = (x_1(t), x_2(t), x_3(t), x_4(t))^T$ .  $x_1(t)$  is the inverted pendulum's track position at time  $t$ ;  $x_2(t)$  is the inverted pendulum's angle position at time  $t$ ;  $x_3(t)$  is the inverted pendulum's track position velocity at time  $t$ ; and  $x_4(t)$  is the inverted pendulum's angle velocity at time  $t$ .

We implemented the method presented above to determine the maximum stability region when varying the control loop period from 0.001(sec) to 0.032 (sec). Figure 5 illustrates the corresponding stability index v.s. control loop period for the inverted pendulum system.

The physical meaning of the decreasing shape of the stability index shown in Figure 5 is clear: as the control loop period decreases (i.e. controller runs faster), the system becomes more stable, hence the stability index  $A(T)$  increases. On the other hand, as the control loop period increases (i.e. controller runs slower), the plant becomes less stable, hence the stability index  $A(T)$  decreases. So in order to have a larger maximum stability region, the HAC should run faster. This is supported in ORTEGA as the HAC can run at a rate faster than that of the HPC, as long as system schedulability is guaranteed. We provide the schedulability analysis of ORTEGA in next section.

## 4.2 Schedulability Analysis for ORTEGA Fault Tolerance Architecture

The new ORTEGA fault tolerance architecture saves CPU resource compared with Simplex. The saved CPU resource could be used for other real-time tasks. In this section, we discuss the schedulability analysis of ORTEGA together with these real-time tasks.

Consider an FT-enabled plant  $PL$  under the protection of ORTEGA. Let's denote the task when  $PL$  is being controlled by the HPC as  $\tau^p$  and denote the task when  $PL$  is being controlled by the HAC as  $\tau^a$ . Their timing parameters

are  $(C^p, T^p)$  and  $(C^a, T^a)$  respectively. When a fault is detected in the HPC, the decision module will issue a recovery request (RR) to initiate the recovery procedure. As a result,  $\tau^p$  will be aborted and the new task  $\tau^a$  will be activated for recovery.

Using similar notations, the decision module is modeled as real-time task  $\tau_d^p$  when  $PL$  is controlled under the HPC and modeled as task  $\tau_d^a$  when  $PL$  is controlled under the HAC. In ORTEGA, when controller switches, the decision module's period also switches accordingly. So tasks  $\tau^p$  (i.e. controller task) and  $\tau_d^p$  (i.e. decision module task) have the same phase and period. From a schedulability analysis perspective,  $\tau^p$  and  $\tau_d^p$  can be modeled as one single task  $\tilde{\tau}^p$ . Its execution time is  $\tilde{C}^p = C^p + C_d^p$ , its period is  $\tilde{T}^p = T^p = T_d^p$ . Similarly tasks  $\tau^a$  and  $\tau_d^a$  can also be modeled as one single task  $\tilde{\tau}^a$ , where  $\tilde{C}^a = C^a + C_d^a$ ,  $\tilde{T}^a = T^a = T_d^a$ .

As a result, when scheduled together with other real-time tasks, the decision modules and controllers for a single FT-enabled plant can be modeled as an abstract task  $\tilde{\tau}$ . It has two subtasks  $\tilde{\tau}^p$  and  $\tilde{\tau}^a$ , where  $\tilde{\tau}^p$  is running when the plant is under the control of the HPC and  $\tilde{\tau}^a$  is running when the plant is under the control of the HAC. Task  $\tilde{\tau}$  is called an **FT-enabled task**.

Assume there are a total of  $N$  real-time tasks,  $\tau_1, \dots, \tau_N$ , running on the CPU. They are ordered in the sequence of their priorities, with  $\tau_1$  having the highest priority and  $\tau_N$  having the lowest priority. Among them, tasks  $\tau_k$ , ( $k \in \mathcal{FT}$ ) are the FT-enabled tasks,  $\mathcal{FT} \subseteq \{1, \dots, N\}$  is the set of all FT-enabled tasks. Each  $\tau_k$ , ( $k \in \mathcal{FT}$ ) is composed of two subtasks:  $\tau_k^p$  represents the combined decision and control task when the plant is being controlled by its HPC;  $\tau_k^a$  represents the combined decision and control task when the plant is being controlled by its HAC.

**Definition 1** Given the task set  $\{\tau_i\}$ ,  $i \in \{1, \dots, N\}$ , among which tasks  $\tau_k$ , ( $k \in \mathcal{FT}$ ) are FT-enabled tasks,  $\mathcal{FT} \subseteq \{1, \dots, N\}$ . If the task set  $\{\tau_i\}$  is schedulable under the ORTEGA task recovery and switch-back scheme with random failures, it is called **FT-schedulable**.

We will focus on the schedulability analysis when there is one FT-enabled task  $\tau_k$  in the task set, i.e.  $\mathcal{FT} = \{k\}$ ,  $1 \leq k \leq N$ . In ORTEGA, when a fault in the HPC is identified, the decision module will issue a recovery request (RR). At the same time, task  $\tau_k^p$  will be aborted, and  $\tau_k^a$  will be activated for recovery. This potentially raises a mode-change problem [16][20], since tasks  $\tau_k^p$  and  $\tau_k^a$  may have different timing parameters. We need to carry out schedulability analysis to guarantee that the tasks are schedulable during the recovery. Similarly, after the recovery,  $\tau_k^a$  can be switched back to  $\tau_k^p$  for retry or performance considera-

tions. We also need to guarantee the tasks are still schedulable with the switch-back.

We can easily apply the tools proposed in Real et al. [20] to analyze the aforementioned the mode-change schedulability problems. Due to page limit, we shall elaborate our analysis in our future publication.

## 5 Implementation and Evaluation of ORTEGA

We have implemented ORTEGA on an inverted pendulum testbed. The inverted pendulum and control hardware is manufactured by Quanser corporation (<http://www.quanser.com/>). Inverted pendulum is inherently unstable. If it tilts past a certain angle, even missing one control output from the controller would lead the pendulum to fall. As a result, the fault detection and recovery must be carried out in a timely and predictable manner [11]. In our test-bed, the control computer which runs ORTEGA uses a Pentium II 350MHz processor with 66MHz memory bus; it has 32KB of level 1 cache memory on chip, uses Quadrature optical encoder interface for sensing input, and uses a digital to analog converter for control output. ORTEGA is implemented in C, and runs on Linux kernel version 2.4.18-3 with RT scheduling and kernel preemption enhancements. ORTEGA uses a fixed priority scheduling scheme — Rate Monotonic Scheduling, as it is widely supported by current systems and standards including the POSIX real-time extension [21].

In our tested, we use a field-tested state feedback controller as the high-assurance controller (HAC). It was designed to be simple so it can be easily checked that there is no bug.

The high performance controller (HPC) can be any third party controller uploaded dynamically to our testbed.

The default HPC is non-faulty. We use it as a benchmark to evaluate the CPU resource savings under ORTEGA. Table 1 shows the execution time of the default HPC and the HAC. If the HPC and HAC both use the same period (the exact period value is irrelevant), then compared to Simplex, ORTEGA saves 29.29% of the CPU. If HPC and HAC have different periods, for example, HPC has a period of  $50msec$  and HAC has a period of  $20msec$ , then ORTEGA saves 50.87% of CPU compared to Simplex.

We also use other faulty HPCs to test the robustness of ORTEGA. The tested faults/bugs include (but are not limited to):

### 1. Infinite loop bug

The faulty HPC controller performs normal operation for a while, say 20 seconds (or after 400 control loops when the control rate is 20Hz), then it goes into an infinite idle loop.

### 2. Non performing bug

**Table 1. Execution statistics for the non-faulty HPC and the HAC**

Controller	Average Execution Time ( $\mu\text{s}$ )	Variance of Execution Time	Minimum Execution Time ( $\mu\text{s}$ )	Maximum Execution Time ( $\mu\text{s}$ )
HPC	2.6705	0.02181	2.3571	3.2857
HAC	1.1060	0.005812	0.9429	1.6371

The faulty HPC controller performs normal operation for a while, then it outputs control command of 0. In the inverted pendulum system, this corresponds to outputting a control voltage of 0 volt.

### 3. Maximum control output bug

The faulty HPC controller performs normal operation for a while, then it outputs the maximum control command allowed by the actuator. In the inverted pendulum system, this corresponds to outputting the control voltage of 5 volts.

### 4. Bang-bang type bug

The faulty HPC controller performs normal operation for a while, then it outputs the maximum and minimum allowable control values in a bang-bang manner. In the inverted pendulum system, this corresponds to outputting control voltages of +5 volt and -5 volts in every other control period.

### 5. Positive feedback control bug

The faulty HPC controller performs normal operation for a while, then it outputs the opposite of the correct feedback control values. The positive feedback will make the controlled system unstable.

### 6. Divided by zero bug

The faulty HPC controller performs normal operation for a while, then it calculates the control value with a faulty divided by zero operation.

According to our results, ORTEGA tolerates all the faults/bugs. When the HPC controlled systems fails the decision module's test, ORTEGA replaces the HPC with the HAC. Hence the inverted pendulum does not fall down.

## 6 Conclusions

In this paper we presented a new fault tolerance architecture, ORTEGA, for real-time control systems. Similar to Simplex, ORTEGA is reliable and achieves high fault coverage. Compared with Simplex, ORTEGA has advantages including that it allows more efficient resource utilizations and enhances flexibility. This is achieved by running the high-assurance controller in an on-demand fashion instead of running in parallel. We implemented ORTEGA on an inverted pendulum control testbed and carried out extensive benchmark tests to evaluate the performance of ORTEGA.

Results demonstrate the efficiency and effectiveness of ORTEGA. We believe ORTEGA is a promising real-time fault tolerance architecture and is applicable to a wider range of industrial applications where both efficient resource usage and high fault coverage are desired.

In the future, we plan to make ORTEGA available to the industry and test its performance in more complex real-world deployments.

## References

- [1] National Research Council. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. National Academies Press, 2001.
- [2] K. J. Astrom and B. Wittenmark. *Computer-Controlled Systems: Theory and Design, 3rd edition*. Addison-Wesley Pub Co., 1994.
- [3] A. Avizienis. The methodology of n-version programming. In M. R. Lyu, editor, *Software Fault Tolerance*. John Wiley and Sons, New York, 1995.
- [4] H. Aydin, R. Melhem, and D. Mosse. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA00)*, 2000.
- [5] H. Aydin, R. Melhem, and D. Mosse. Tolerating faults while maximizing reward. In *Twelfth Euromicro Conference on Real-Time Systems (Euromicro'00)*, Stockholm, Sweden, 2000.
- [6] 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.
- [7] G. Buttazzo. *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*. Springer, 2nd edition, 2005.
- [8] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [9] S. Graham, G. Baliga, and P. R. Kumar. Issues in the convergence of control with communication and computing: proliferation, architecture, design, services, and middleware. In *Proceedings of the 43rd IEEE Conference on Decision and Control*, December 2004.
- [10] K. H. Kim. Slow advances in fault-tolerant real-time distributed computing. In *Proceedings of 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 106–108, Florianopolis, Brazil, October 2004.

- [11] K. Lee and L. Sha. Process resurrection: A fast recovery mechanism for real-time embedded systems. In *11th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, California, 2005.
- [12] J. Liu. *Real-Time Systems*. Prentice Hall PTR, 2000.
- [13] J. Lofberg. YALMIP : A toolbox for modeling and optimization in matlab. In *2004 IEEE International Symposium on Computer Aided Control Systems Design*, Taipei, Taiwan, 2004.
- [14] M. Lyu, editor. *Software Fault Tolerance (Trends in Software, No. 3)*. John Wiley and Sons, New York, 1995.
- [15] P. M. Melliar-Smith and L. E. Moser. Progress in real-time fault tolerance. In *Proceedings of 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 109–111, Florianopolis, Brazil, October 2004.
- [16] P. Pedro and A. Burns. Schedulability analysis for mode changes in real-time systems. In *10th Euromicro workshop on Real-Time systems*, Berlin, Germany, 1998.
- [17] D. Pradhan and N. Vaidya. Roll-forward checkpointing scheme: A novel fault-tolerant architecture. *IEEE Transactions on Computers*, 43(10), October 1994.
- [18] R. Ramakrishnan and J. Gehrke. *Database Management Systems (3rd Edition)*. McGraw-Hill, 2003.
- [19] B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*. John Wiley and Sons, 1995.
- [20] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.
- [21] Real-Time System Services Working Group. IEEE STD 1003.1, 1996 edition, 1998.
- [22] M. Roozbehani, A. Megretski, and E. Feron. Convex optimization proves software correctness. In *American Control Conference, 2005. Proceedings of the 2005*, pages 1395–1400 vol. 2, 2005.
- [23] M. Roozbehani, A. Megretski, and E. Feron. Safety verification of iterative algorithms over polynomial vector fields. In *Decision and Control, 2006. Proceedings of the 45th IEEE Conference on*, pages 6061–6067, 2006.
- [24] D. Seto, E. Ferreira, and T. Marz. Case study: Development of a baseline controller for automatic landing of an F-16 aircraft using Imis. Technical Report CMU/SEI-99-TR-020, Software Engineering Institute, Carnegie Mellon University, 2000.
- [25] D. Seto, B. H. Krogh, L. Sha, and A. Chutinan. Dynamic control system upgrade using the simplex architecture. *IEEE Control System Magazine*, 1998.
- [26] 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.
- [27] D. Seto and L. Sha. An engineering method for safety region development. Technical Report 18, CMU SEI, 1999.
- [28] L. Sha. Dependable system upgrade. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 440. IEEE Computer Society, 1998.
- [29] 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.
- [30] L. Sha, R. Rajkumar, and S. Sathaye. Generalized rate monotonic scheduling theory: A framework of developing real-time systems. *IEEE Proceedings*, 1994.
- [31] J. A. Stankovic. Real-time computing systems: The next generation. In J. A. Stankovic and K. Ramamritham, editors, *Tutorial: Hard Real-Time Systems*, pages 14–37. IEEE Computer Society, 1998.
- [32] K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.
- [33] S.-P. Wu and S. Boyd. Sdpsol: a parser/solver for sdp and maxdet problems with matrix structure. In L. E. Ghaoui and S.-I. Niculescu, editors, *Recent Advances in LMI Methods for Control*. SIAM, 1999.