INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

REAL-TIME PERFORMANCE GUARANTEES

IN

MANUFACTURING SYSTEMS

by

Lei Zhou

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1999

Doctoral Committee:

Professor Kang G. Shin, Co-Chair
Assistant Professor Elke A. Rundensteiner, Co-Chair
Professor Yoram Koren
Assistant Professor Nandit Soparkar
Professor Toby J. Teorey

To

my wife Huishan Joy Wang,

my mom Meiyu Sun,

and my dad Rongzhan Zhou,

whose love and support I can always count on

ACKNOWLEDGEMENTS

My advisors, Professors Kang G. Shin and Elke A. Rundensteiner, have been very helpful with their valuable advice and suggestions. Special thanks to members of the Real-Time Computing Laboratory (RTCL) and University of Michigan Database Group (UMDG), in particular, Tarek F. Abdelzaher, Nauman A. Chaudhry, Viviane M. Crestana, Scott Dawson, Wu-Chang Feng, Seungjae Han, Stacie L. Hibino, Parikh Hiren, Yun-Wu Huang, Ning Jing, Matthew C. Jones, Jin-Ho Kim, Harumi A. Kuno, Amy Lee, Ashish Mehra, Todd Mitton, Anisoara Nica, Young-Gook Ra, Anees A. Shaikh, Youngsoo Shin, Chi-To Shiu, Shi-Ge Wang, Michael J. Washburn, Xi Zhang, and Khawar M. Zuberi, for their help and feedback on my work and their company during my study.

This work was supported in part by the National Science Foundation under Grants DDM-9313222, IRI-9309076 and IRI-9504412, and the Engineering Research Center for Reconfigurable Machining Systems.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

LIST OF TABLES x

CHAPTER 1

INTRODUCTION 1

1.1 Motivation 1
1.2 Background 4
1.2.1 Real-time computing 4
1.2.2 Real-time performance guarantees 5
1.2.3 Real-time application development 7
1.3 Target Application Domain 8
1.4 A Map of the Dissertation 10

CHAPTER 2

OPERATING SYSTEM UNPREDICTABILITY 12

2.1 Introduction 12
2.2 Timer interval unpredictability 13
2.2.1 VxWorks timer 13
2.2.2 QNX timer 17
2.2.3 pSOSystem timer 18
2.3 Task Execution Time Unpredictability 19
2.4 Causes of Unpredictability 23
2.4.1 OS services 23
2.4.2 OS interrupts 24
2.4.3 Major classes of RTOS activities 26
2.4.4 Relative significance of RTOS activities 27
2.5 Summary 32

CHAPTER 3

HARD DEADLINE GUARANTEES IN THE PRESENCE OF TIMING UNPREDICTABILITY 34

3.1 Introduction 34
3.2 Scheduling Algorithm Performance in the Presence of Timing Unpredictability 37
3.2.1 Simulations 37
3.2.2 Experimental measurements 49
3.2.3 Discussion 52
3.3 RMTU: Rate-Monotonic in the presence of Timing Unpredictability 54
3.3.1 An empirical task schedulability model: RMTU 55
3.3.2 Derivation of model parameters 61
3.3.3 Model validation 65
3.3.4 Discussion 68
3.4 Related work 71
3.5 Summary 73

CHAPTER 4

PROBABILISTIC DEADLINE GUARANTEES 74

4.1 Introduction 74
4.2 Probabilistic Real-Time Constraint Model (PRTCM) 76
4.3 Performance Evaluation Criteria 80
4.3.1 Task-oriented objective function 81
4.3.2 Job-oriented objective function 82
4.3.3 Temporary overload 83
4.4 Heuristics for Probabilistic Guarantees 85
4.4.1 Completion-probability-cognizant heuristics 85
4.4.2 CPU-utilization-cognizant heuristics 87
4.5 Simulation Parameters 88
4.5.1 CPU utilization patterns 89
4.5.2 Task period distributions 90
4.6 Evaluation Results 90
4.6.1 Variable CPU utilizations 91
4.6.2 Fixed task execution times 95
4.6.3 Fixed task CPU utilizations 97
4.6.4 Summary 98
4.7 Comparison between Probabilistic and Deterministic Guarantees 100
4.8 Related Work 107
4.9 Summary 110

CHAPTER 5

MEASUREMENT-BASED SIMULATION TECHNIQUE 112

5.1 Introduction 112
5.2 MBST: Measurement-Based Simulation Technique for Probabilistic Deadline Guarantees 113
5.2.1 Task model 115
5.2.2 Run-time model 118
5.2.3 Simulation model 119
5.2.4 Probabilistic deadline guarantees 120
5.3 Application of MBST to Open-Architecture Machine Tool Controllers 121
5.3.1 UMOAC Testbed 122
5.3.2 Controller Tasks 124
5.3.3 Tasks with constant nominal execution times 125
5.3.4 Machine tool controller tasks 136
5.4 Discussion and Related Work 153
5.5 Summary 154

CHAPTER 6

APPLICATION DEVELOPMENT 155

6.1 Introduction 155
6.2 Prototype Open-Architecture Milling Machine Controllers 156
6.3 Two-Axis Controller with Separate Task Structure 158
6.4 Testbed without Network 162
6.5 Two-Axis Controller with Combined Task Structure 163
6.6 Related Work 169
6.7 Summary 169

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS 171

7.1 Research Contributions 171
7.2 Future Directions 172

BIBLIOGRAPHY 174

LIST OF FIGURES

Figure

1. Basic control loop. 9
2. Histogram of VxWorks timer intervals (bin width: 100 μs)....14
3. Histogram of VxWorks event-generating function execution times (bin width: 0.1 μs). 15
4. Timer "memory" behavior....16
5. Histogram of QNX timer intervals (bin width: 100 μs)....18
6. Histogram of pSOSystem timer intervals (bin width: 100 μs). 19
7. Measured execution time of task 1 in the first set (stand-alone). 21
8. Measured execution time of task 1 in the second set (RM)....22
9. Measured execution time of task 1 in the third set (FIFO)....22
10. Impact of tick size on task interval. 30
11. Impact of tick size on task execution time....31
12. Distribution of task periods....40
13. Performance of scheduling algorithms (seed: -1). 45
14. Performance of scheduling algorithms (seed: -2222). 46
15. Performance of scheduling algorithms (seed: -77). 46
16. Task execution with an ideal software timer. 56
17. Task execution with timer variation. 56
18. Execution time vs. response time vs. service time. 63
19. Single-task achievable CPU utilization versus task period. 64
20. Residual-value-based categorization of real-time task deadlines. 77
21. Completion-probability- and residual-value-based categorization. 79
22. Spectrum of fixed-priority scheduling algorithms. 85
23. Bimodal and uniform task period distributions. 91
24. Performance of scheduling algorithms under variable CPU utilizations and uniform period distribution. 93
25. Performance of scheduling algorithms under variable CPU utilizations and bimodal period distribution. 94
26. Performance of scheduling algorithms under fixed task execution times and uniform period distribution. 95
27. Performance of scheduling algorithms under fixed task execution times and bimodal period distribution. 96
28. Performance of scheduling algorithms under fixed task CPU utilizations and uniform period distribution. 97
29. Performance of scheduling algorithms under fixed task CPU utilizations and bimodal period distribution. 98

30. Performance comparison between probabilistic and deterministic guarantees under variable CPU utilizations & uniform task period distribution. 102
31. Performance comparison between probabilistic and deterministic guarantees under variable CPU utilizations & bimodal task period distribution. 102
32. Performance comparison between probabilistic and deterministic guarantees under fixed task execution times & uniform task period distribution. 105
33. Performance comparison between probabilistic and deterministic guarantees under fixed task execution times & bimodal task period distribution. 105
34. Performance comparison between probabilistic and deterministic guarantees under fixed task CPU utilizations & uniform task period distribution. 106
35. Performance comparison between probabilistic and deterministic guarantees under fixed task CPU utilizations & bimodal task period distribution. 106
36. Measurement-based simulation technique (MBST). 115
37. Comparison of data-generating approaches (XYZ Servo task). 117
38. Open-architecture controller model. 122
39. University of Michigan Open-Architecture Controller testbed. 123
40. Milling machine control application. 124
41. Execution time of simplified Display task running in isolation. 128
42. Execution time of simplified X Servo task running in isolation 128
43. Execution time of simplified XY Servo task running in isolation 129
44. Execution time of simplified XYZ Servo task running in isolation. 129
45. Execution time of simplified Force Acquisition task running in isolation. 130
46. Execution time of simplified Force Supervisor task running in isolation. 130
47. Execution time of simplified X Servo task running with Display. 132
48. Execution time of simplified XY Servo task running with Display. 132
49. Execution time of simplified XYZ Servo task running with Display 133
50. Execution time of simplified Display task running with X Servo. 133
51. Execution time of simplified Display task running with XY Servo. 134
52. Execution time of simplified Display task running with XYZ Servo 134
53. Execution time of simplified Force Acquisition task running with XYZ Servo, Force Supervisor and Display. 135
54. Execution time of simplified XYZ Servo task running with Force Acquisition, Force Supervisor and Display. 136
55. Execution time of simplified Force Supervisor task running with Force Acquisition, XYZ Servo and Display 136
56. Execution time of simplified Display task running with Force Acquisition, XYZ Servo and Force Supervisor. 137
57. Typical components of an open-architecture controller task. 138
58. Simulated and measured task execution time of Force Acquisition. 139
59. Simulated and measured task execution time of X Servo. 140
60. Simulated and measured task execution time of XY Servo 140
61. Simulated and measured task execution time of XYZ Servo. 141
62. Simulated and measured task execution time of Force Supervisor. 141
63. Simulated and measured task execution time of Display. 142
64. Execution time of X Servo task running with Display. 143
65. Execution time of XY Servo task running with Display. 143

66. Execution time of XYZ Servo task running with Display. 144
67. Execution time of Display task running with X Servo. 145
68. Execution time of Display task running with XY Servo. 145
69. Execution time of Display task running with XYZ Servo. 146
70. Execution time of Force Acquisition task running with XYZ Servo, Force Supervisor and Display. 146
71. Execution time of XYZ Servo task running with Force Acquisition, Force Supervisor and Display. 147
72. Execution time of Force Supervisor task running with Force Acquisition, XYZ Servo and Display. 147
73. Execution time of Display task running with Force Acquisition, XYZ Servo and Force Supervisor. 148
74. Completion time of Force Acquisition task running with XYZ Servo, Force Supervisor and Display. 150
75. Completion time of XYZ Servo task running with Force Acquisition, Force Supervisor and Display. 150
76. Completion time of Force Supervisor task running with Force Acquisition, XYZ Servo and Display. 151
77. Completion time of Display task running with Force Acquisition, XYZ Servo and Force Supervisor. 151
78. Prototype milling machine controller. 157
79. Prototype two-axis modular controller with separate task structure. 158
80. Task execution sequence. 162
81. Performance with & without QNX network drivers. 164
82. Prototype two-axis modular controller with combined task structure. 166
83. Mean intervals before and after combining tasks. 167
84. Standard deviations before and after combining tasks. 167

LIST OF TABLES

Table

1. VxWorks POSIX timer measurement statistics. 14
2. Sample measured periods with VxWorks 16
3. QNX POSIX timer measurement statistics. 17
4. pSOSystem timer measurement statistics. 19
5. Task sets 20
6. Statistics of measured execution times of task 1. 23
7. Measured statistics of task with priority 25 under QNX. 29
8. Measured statistics of task with priority 27 under QNX. 29
9. Simulation parameters. 38
10. Impacts of task random start and interval variation on miss ratios. 41
11. Impacts of task random start and interval variation with timer resets. 42
12. Overhead of timer resets. 43
13. Performance of scheduling algorithms under different system load. 45
14. Deadline misses of individual tasks. 47
15. Interval statistics under RM without timer resets. 49
16. Interval statistics under EDF without timer resets. 50
17. Interval statistics under FIFO without timer resets. 51
18. Miss ratios of simulations and measurements. 51
19. Statistics of task periods. 52
20. Statistics of timers with different nominal periods. 60
21. Single task experiment results. 65
22. Validation with three-task experiments. 66
23. Validation with five-task experiments. 67
24. List of simulation parameters. 89
25. Performance of scheduling algorithms under variable CPU utilizations and uniform period distribution. 92
26. MBST simulation configurations 121
27. Prototype controllers and their respective tasks. 125
28. Statistics of execution times of simplified tasks. 127
29. Statistics of timer overhead 131
30. Logical components of prototype controller tasks 139
31. Task completion times of the 3-axis controller with force control. 152
32. Predicted task completion times of 5-axis controller with force control. 153
33. Statistics for controller with separate task structure. 160
34. Start times of controller tasks. 161

35. Statistics for controller with separate task structure and without QNX network drivers running. 164
36. Statistics for controller with combined task structure and with QNX network drivers running. 166

CHAPTER 1

INTRODUCTION

1.1 Motivation

Timing constraints are an integral part of the correctness and safety of a real-time system. The single most important characteristic of a real-time system is the ability to determine for a given set of tasks whether the system will be able to meet their timing requirements.

A real-time task is described by its arrival time, deadline, worst-case contention-free execution time, and criticality (i.e., *hard*, *firm* and *soft* [142]). Under this categorization, only hard real-time tasks can possibly obtain deadline guarantees, while firm and soft real-time tasks receive only best-effort services.

There has been extensive research in the areas of scheduling theory for hard real-time tasks [47, 48, 51, 55, 77, 78, 88, 93, 137, 138]. However, there exists a wide gap between such real-time scheduling theories and the reality of applying the theory to task sets implemented via real-time operating systems (RTOSs). To simplify the issues associated with providing hard deadline guarantees, these real-time scheduling theories, such as the rate-monotonic (RM) scheduling theory [93], have generally ignored the implementation costs and unpredictability involved in scheduling a task set in a RTOS. However, a RTOS may introduce significant system overhead and unpredictability, for example, run-time priority-changing overhead and timer interval variation. Large overhead related to the change of task priority at run-time may make dynamic-priority scheduling prohibitively expensive. Time interval variation could delay the release of

tasks and cause them to miss their deadlines. Our research fills the gap by taking the RTOS unpredictability into account in determining if all deadline can be guaranteed.

For periodic hard real-time tasks, we introduce an empirical task schedulability model, called *Rate-Monotonic in the presence of Timing Unpredictability* (RMTU), to augment the original RM scheduling algorithm to handle timing unpredictability. The model parameters are determined empirically and systematically by running and measuring a set of simple tasks on the target system. The model is empirical because its parameters are derived from measurement data. It is also systematic because it includes a set of systematic experiments, which can be applied to different target systems.

While there is also much research in scheduling for firm and soft real-time tasks [31, 52, 60, 61, 87, 94, 142, 145, 147, 148, 149, 150, 175, 178], this is often inadequate for specifying requirements and characterizing performance of many real-time applications, where tasks can tolerate deadline misses but only to a certain degree. For example, a machine tool controller may function satisfactorily if it can obtain 90% of sensor readings in time. Such tolerance of deadline misses indicates that these controller tasks are not hard real-time tasks and therefore do not require hard deadline guarantees. But, on the other hand, best efforts are insufficient for these tasks because there is no guarantee that, for example, 90% of sensor readings will be done on time.

To address this problem, we develop a practical framework for probabilistic deadline guarantees. The first component of this framework is the *Probabilistic Real-Time Constraint Model* (PRTCM), with which the tolerance of application task deadline misses can be quantified in terms of *completion probability*. The second component consists of two classes of new scheduling algorithms: *completion-probability-cognizant* and *CPU-utilization-cognizant* heuristics. We then evaluate their performance as well as scheduling algorithms RM, Earliest-Deadline-First (EDF) and First-In-First-Out (FIFO), in the context of probabilistic deadline guarantees. An especially interesting situation occurs when the system is temporarily overloaded, due to RTOS unpredictability (e.g., timer

jitters [185]), hardware failure, or optimistic scheduling. Our simulation results show that most scheduling algorithms work well if the system is not overloaded. However, when the system is temporarily overloaded, RM performs well in terms of *useful job ratio* while UM_CP is superior in terms of *task completion probability miss ratio*. Finally, we show that the introduction of completion probability can improve CPU utilization as well as *job* and *task guarantee ratios*, by comparing the performance of these scheduling algorithms between probabilistic and deterministic deadline guarantees.

Probabilistic deadline guarantees require the knowledge of task completion time distributions. There can be many approaches to obtaining task completion time distributions. One extreme is to use formal analyses or simulations that assume an idealized computing environment. However, RTOS unpredictability makes such an approach difficult and unreliable. The other extreme relies solely on actual measurement data. However, without an understanding of the system or the application, such a method will be of little value because it cannot predict the application performance in a different computing environment or a slightly different application in the same environment.

To achieve a balance between the two extremes and maximize the benefits of both, we propose a *Measurement-Based Simulation Technique* (MBST) for making probabilistic deadline guarantees. MBST uses individual application task execution times (measured in isolation) as inputs, models task interaction and system overhead, and generates task completion time distributions to determine whether probabilistic deadline guarantees can be made. Applying MBST to our prototype open-architecture milling machine controllers, MBST is shown to produce results that match very well the actual measurements. It can also be used to predict the performance of tasks that have not yet been fully implemented.

While the above approaches for hard and probabilistic deadline guarantees address the real-time performance issues given an application implementation, we still need to examine how a real-time application should be implemented in order to meet its requirements in the presence of RTOS unpredictability. We investigate the issues related to

application software development and evaluate strategies to minimize the effects of RTOS unpredictability.

1.2 Background

1.2.1 Real-time computing

As computers are becoming an essential part of real-time systems, *real-time computing* has emerged as an important discipline in computer science and engineering.

There are three major components and their interplay that characterize real-time systems [142]. First, time is the most precious resource to manage in real-time systems. A computation is defined as real-time if its correctness depends not only on its logical correctness but also on the time at which it completes. A real-time application is usually comprised of a set of cooperating tasks. The tasks are often invoked at regular intervals and have deadlines by which they must complete their execution, hence referred to as periodic tasks. For example, a sensor-reading task in a control application may read several position and velocity sensors every 10 milliseconds and must finish reading before the end of each period. Other tasks in a real-time application may be invoked only when certain events occur and they are referred to as aperiodic tasks.

The second major component of a real-time system is reliability. This is crucial because a failure in a real-time system could have severe consequences. Traditionally, deadlines of real-time tasks are classified as hard, firm or soft. A deadline is said to be hard if the consequences of not meeting it can be catastrophic, such as the deadline of the emergency shutdown task in a machine tool controller. A deadline is firm if the results produced by the corresponding task cease to be useful as soon as the deadline expires, but the consequences of not meeting the deadline are not catastrophic, e.g., the deadline of weather forecast (except for severe weather conditions). A deadline which is neither hard nor firm is said to be soft. The utility of results produced by a task with a soft deadline

decreases over time after the deadline expires. An example of soft deadlines may be the transaction deadline of an automatic teller machine. The longer the customer waits, the unhappier he or she becomes.

Third, the environment under which a computer operates is an active component of any real-time system. For example, a manufacturing machining system may be comprised of a milling machine and its controller. It is meaningless to consider the controller software and hardware alone without the milling machine itself. Because many important characteristics of the controller are determined by the physical limits of the milling machine, such as the range limits on the machine's displacement and velocity.

1.2.2 Real-time performance guarantees

The notion of predictability is very important to real-time systems. However, the meaning of predictability may vary from one real-time application to another or even from one real-time task to another within the same application. For example, some critical applications or tasks may require *hard deadline guarantees*, i.e., their deadlines are satisfied at all times. Some applications or tasks with less stringent timing constraints may require *probabilistic deadline guarantees*. Depending on the application requirements, the term "probabilistic deadline guarantee" could mean that a certain fraction of tasks are guaranteed to meet their deadlines, or that a given task has a certain probability of meeting its deadline. We provide a definition that encompasses both semantics in Chapter 4.

In order to provide real-time performance guarantees, it is necessary to analyze the tasks that comprise the real-time application to determine if all task deadlines can be met. This problem is called *schedulability analysis* and has been extensively studied in the literature. In general, the schedulability analysis is computationally intractable if the tasks are not preemptive [41] or have data dependencies [104, 152]. However, if the tasks are independent and preemptive, it is possible to efficiently determine their schedulability.

For example, if the total CPU utilization of the tasks is no greater than a threshold computed by the RM scheduling theory [93], all tasks can be provided hard deadline guarantees. RM is optimal for assigning static (pre-assigned) task priorities.

Various dynamic-priority scheduling algorithms have also been studied [51, 55, 93]. EDF is proven to be optimal for assigning dynamic task priorities [93]. At any time instant, it assigns the highest priority to the task with the earliest deadline. Using this scheduling algorithm, it is possible (in theory) to achieve 100% CPU utilization by the tasks. The assumptions and limitations of the RM and EDF scheduling theories are discussed in Chapter 3.

Like all computer applications, real-time applications derive many of their capabilities from the characteristics of their underlying RTOSs. Therefore, to evaluate the application performance, the characteristics of RTOSs should be taken into account. As we will describe in Chapter 2, RTOSs exhibit significant unpredictability, for example, timer interval variation. Such a variation could delay the release of tasks and cause them to miss their deadlines. Therefore, we need to investigate the characteristics of RTOS unpredictability and identify sources of disturbance.

Since a real-time application is typically comprised of a set of periodic tasks, we will measure the performance of such tasks, in terms of variations in task interval and execution time, in three most widely used commercial RTOSs (VxWorks, QNX and pSOSystem). We will then analyze the causes of the performance unpredictability. Our study show that interval and execution time variations are observable and not negligible. Furthermore, the major sources (such as interrupts) of RTOS unpredictability cannot be avoided because they are an inherent part of computer systems.

Any non-negligible RTOS unpredictability must be included in the schedulability analysis for all practical real-time systems. However, little research on schedulability analysis in the presence of RTOS unpredictability has been reported in the literature. Jeffay and Stone [66] considered interrupts in their schedulability analysis for periodic

tasks. They developed conditions under which the feasibility and schedulability problems can be solved, and demonstrated that their solutions are computationally feasible. But they assume that interrupts occur strictly periodically, which is rarely the case in practice. Instead, we develop an empirical schedulability model that extends the RM scheduling theory for hard deadline guarantees to handle timing unpredictability. We systematically measure the target system and derive the model parameters from the measurement data. As a consequence, our model reflects the characteristics of the target system more closely and provides more realistic (and more conservative) hard deadline guarantees.

While the notion of probabilistic guarantees have appeared in the literature [23, 46, 69, 75, 76, 122, 142, 166, 170], few research results on probabilistic guarantees have been reported. Thus, we develop a practical framework for probabilistic deadline guarantees, consisting of PRTCM, completion-probability-cognizant and CPU-utilization-cognizant heuristics, a performance comparison of our heuristics and RM, EDF and FIFO, and MBST.

1.2.3 Real-time application development

Many modern real-time applications, including open-architecture machine tool controllers, rely on the services provided by the underlying RTOS, such as the software timer, scheduler and inter-process communication (IPC). While real-time performance guarantee addresses deadline guarantee issues given an application implementation, it is a different problem how a real-time application should be implemented in order to meet its requirements in the presence of RTOS unpredictability.

While there has been extensive work on RTOSs (e.g., [155, 157, 165]) and their modeling (e.g., [72]) or open-architecture controllers (e.g., [5, 6, 7, 20, 53, 117, 125, 158, 160, 174]), it is unknown how well real-time controllers with such a modular structure perform in practice in an open-architecture environment and what kind or magnitude of impact the underlying RTOS unpredictability has on the controllers. We address this need in Chapter 6.

1.3 Target Application Domain

The requirements and characteristics of different types of real-time systems can be quite different. For example, manufacturing control applications may impose deadlines on real-time data access operations in the order of hundreds of microseconds. In contrast, the timing requirements of an air traffic control application may not be as stringent. Up to 5 milliseconds of read/write response time is acceptable [132].

While most results from our research can be generalized to other real-time systems, we focus on open-architecture machine tool controllers, because our research is an integral part of the Open-Architecture Controller project within the Engineering Research Center for Reconfigurable Machining Systems. We will use open-architecture modular controllers as an example application domain to demonstrate our research results for practical real-time applications.

Machine tool controllers are getting more sophisticated in recent years due mainly to the use of rapidly-advancing computer technology. However, there are still problems of high life-cycle costs and lack of openness in commercially-available controllers. It is often very difficult, if not impossible, to incorporate third-party software (such as new control algorithms or interpolators) into existing controllers. There is considerable research interest in the subject of open control systems, in both academia and industry. Examples of this activity include the *Open System Architecture for Controls within Automation Systems* (OSACA) project [125] in Europe and the *Enhanced Machine Controller* (EMC) project [5] at the National Institute of Standards & Technology in the United States.

As defined by the IEEE 1003.0 Technical Committee of Open Systems, an *open system* provides capabilities that enable properly implemented applications to run on a variety of platforms from multiple vendors, interoperate with other systems applications, and present the user with a consistent style of interaction [62]. This suggests that open-architecture machine controllers should have a modular structure such that a controller can

be assembled with modules from different vendors and can easily be extended by adding or changing modules. Open-architecture controllers should also have well-defined module interfaces so that modules can be developed independently by different vendors. For example, a manufacturer should be able to easily substitute its PID motion control module with a fuzzy logic control module, because the fuzzy logic control module offers better precision of the controlled process.

Controller modules to be plugged into open-architecture controllers can be either hardware or software. Examples of hardware modules include a VME processor board or a dynamometer, while examples of software modules include a POSIX-compliant RTOS kernel or a force controller. Modules can be selected based on price, performance and/or other criteria, while meeting the requirements of the controller. The resulting controller would be more flexible and adaptable.

Figure 1 shows a basic control loop of a typical control application. The controller takes inputs (e.g., the desired positions and velocities) from the user and the feedback (e.g., the actual positions and velocities) from the controlled process, computes new control commands to minimize the discrepancy between the desired and actual values, and sends them to the actuators.

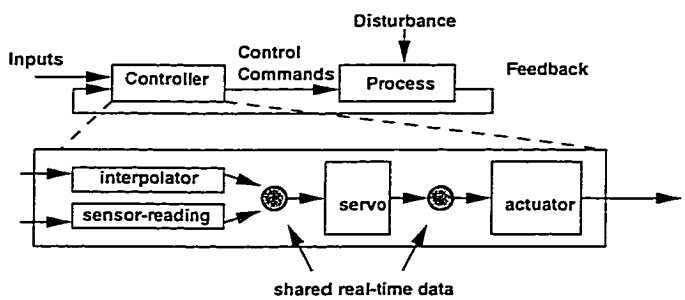


Figure 1: Basic control loop.

A controller consists of a set of periodic real-time tasks, such as sensor-reading tasks and servo motion control tasks, as illustrated in Figure 1, and a few aperiodic tasks, such as the emergency shutdown task.[1] Some control tasks have hard deadlines. For

example, the emergency shutdown task must bring the controlled mechanical machine to a complete stop within a short, pre-specified time limit. Failing to meet the deadline could cause severe property damages or human injuries. Such tasks require hard deadline guarantees. Other control tasks, e.g., the sensor-reading tasks, can tolerate deadline misses but only to a certain degree. Therefore, they require probabilistic deadline guarantees. The deadlines of these periodic tasks are at the end of their respective periods.

In a monolithic controller, the codes of these conceptual tasks are typically interwoven. In an open-architecture modular controller, however, the tasks are actual software modules that can be clearly identified and separated. Furthermore, the modules are constructed and organized in such a way that modules of same types (e.g., sensor modules) are interchangeable in the controller regardless of their vendors, as long as they meet the application requirements. The issues related to how to modularize the controllers are discussed in [5, 6, 7, 19, 125], but beyond the scope of this dissertation.

1.4 A Map of the Dissertation

The remainder of the dissertation is organized as follows:

Chapter 2 presents our measurement results of system unpredictability in three commercial RTOSs: VxWorks, QNX and pSOSystem. We identify the characteristics of system unpredictability and the sources of system disturbance.

Chapter 3 describes our approach for hard deadline guarantees in the presence of RTOS unpredictability. We first examine the effects of timing unpredictability on three real-time scheduling algorithms (RM, EDF and FIFO) in terms of task deadline miss ratios, using both simulations and experimental measurements. We then propose an empirical task schedulability model (Rate-Monotonic in the presence of Timing

1. The emergency shutdown task is not shown in Figure 1, because it is actually a higher-level task that overwrites the inputs to the controller. It relies on the lower-level control tasks (such as those shown in the figure) to bring the machine to a stable state within the time limit.

Unpredictability, or RMTU), as well as a systematic approach to determine the model parameters empirically.

Chapters 4 and 5 present our practical framework for probabilistic deadline guarantees. In Chapter 4, we define the Probabilistic Real-Time Constraint Model (PRTCM), introduce completion-probability-cognizant heuristics and CPU-utilization-cognizant heuristics, and compare their performance with RM, EDF and FIFO. In Chapter 5, we describe the Measurement-Based Simulation Technique (MBST) and demonstrate its application to our prototype open-architecture milling machine controllers.

Chapter 6 describes our experiments with real-time application development strategies to minimize the impact of RTOS unpredictability.

Chapter 7 presents a summary of our contributions and a discussion of future directions.

CHAPTER 2

OPERATING SYSTEM UNPREDICTABILITY

2.1 Introduction

Developing complex real-time systems is an evolutionary process fraught with technical difficulties. Historically, much of the real-time research has been based on idealized conditions like negligible implementation costs or nonexistence of RTOS unpredictability. For example, the RM scheduling theory [93] assumes an ideal system with neither implementation overhead nor timing unpredictability.

In reality, no RTOS can provide ideal services, such as POSIX timers and system scheduler, simply because there is always overhead associated with the implementation of such services. Furthermore, contemporary microprocessor architectures (such as Intel x86 and Pentium processors) rely on hardware interrupts to manage system resources (e.g., I/O devices). Interrupts can cause significant unpredictability in RTOS services, such as timer interval variation.

To help us research the issues related to providing deadline guarantees and developing real-time open-architecture controllers in the presence of RTOS unpredictability, we first attempt to identify the characteristics of such unpredictability, investigate the sources of disturbance, and study if they can be eliminated or alleviated.

2.2 Timer interval unpredictability

In a multi-tasking environment, periodic tasks rely on the RTOS timer to realize the periodicity. The timer service typically consists of one or more functions, such as the POSIX [62] functions *timer_create()*, *timer_delete()*, *timer_gettime()* and *timer_settime()*. A task uses a software timer to generate an event periodically. After each invocation of the task is completed, the task goes into sleep until the timer wakes it up with the next timer event.

Clearly, the characteristics of the RTOS timer service have a potentially significant impact on the performance of periodic tasks. We now present the experiments to measure the timer behaviors of three commercial RTOSs—VxWorks, QNX and pSOSystem.

2.2.1 VxWorks timer

Our first set of timer experiments is conducted on a Motorola MVME147 board. This is a VMEbus-based processor board with a Motorola 30 MHz 68030 CPU and 4 MB of RAM. It runs VxWorks (version 5.1.1) [171]. VxWorks is a development and execution environment for real-time embedded applications on a wide variety of target processors. It includes a high-performance RTOS which executes on a target processor. The system clock resolution of the VxWorks kernel is 1 *ms* (1000 ticks/second).

The VME StopWatch [49] is used for timing measurements. It is a board that is pluggable into the VME chassis and time-stamps bus events. These events are reads or writes to specific VME extended addresses. In our experiments, a simple inline function call is used to generate the events. The internal clock resolution of the VME StopWatch is 25 *ns*.

We use the POSIX timer to periodically trigger a simple task τ which consists of only two consecutive function calls that generate events E_1 and E_2, respectively. There is no other code between the function calls. Therefore, the elapsed time between E_1 and E_2 is

the execution time of one function call,[1] while the elapsed time between two consecutive E_1 (or E_2) events is the task interval (or timer interval, i.e., interval between two consecutive timer firings). Table 1 lists the statistics of the measurement of a task with a 10 *ms* period using the POSIX timer, while Figures 2 and 3 show the histograms of the measured task intervals and event-generating function execution times, respectively.

	Execution Time	Interval
Sample Size	963	962
Mean (μs)	1.90	9999.9
Standard Deviation (μs)	0.0383	202.7
Min (μs)	1.85	7716.3
Max (μs)	2.20	11964.9

Table 1: VxWorks POSIX timer measurement statistics.

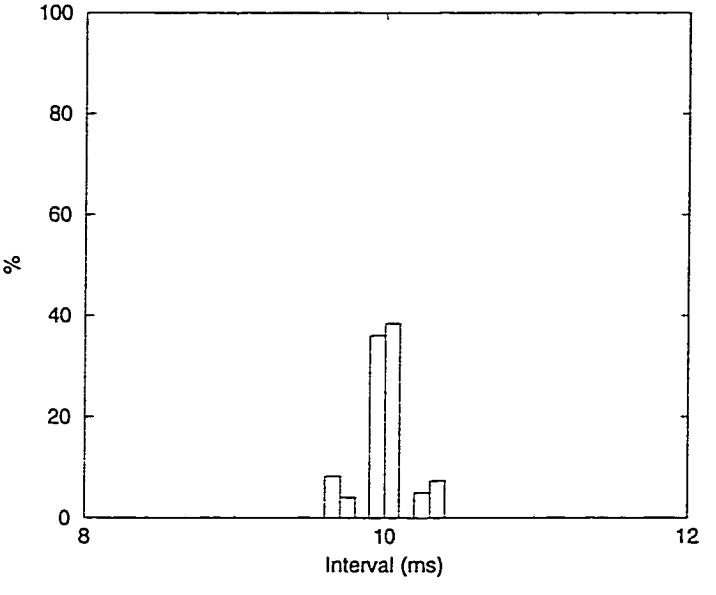


Figure 2: Histogram of VxWorks timer intervals (bin width: 100 μs).

1. Strictly speaking, this elapsed time consists of the execution time of the function call and system overhead associated with the call. Since we are only interested in the overall effect of the call, there is no need to separate the two components of the elapsed time.

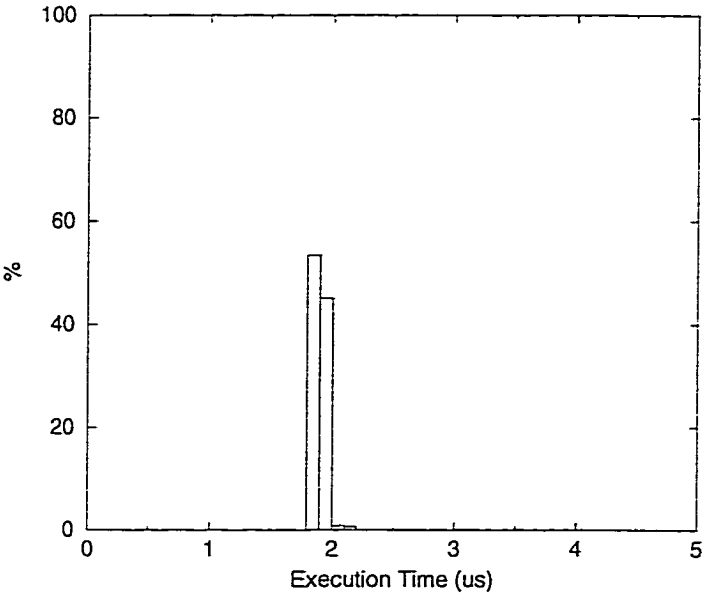


Figure 3: Histogram of VxWorks event-generating function execution times (bin width: 0.1 μs).

From the measurements, we made two important observations about timer characteristics. First, timer intervals vary around the nominal period (see Figure 2). This is because there are other OS activities besides the user task that consume CPU cycles. Many OS tasks run at higher priorities than user tasks. For example, the VxWorks process manager *tExcTask* runs at the highest priority. The hardware interrupts generated by devices (e.g., network devices) are also serviced at higher priorities by the OS kernel. All these inevitably cause the fluctuations in the timer firings as well as task execution times (see Figure 3).

Second, timers have exhibited "memory" behavior. From the actual measurement data (not shown here), we observe that, whenever an interval deviates from the nominal period by at least one or two percentage points, the next interval almost always swings in the opposite direction by roughly the same amount. Table 2 lists a subset of consecutively measured periods. Every number in the five-column table represents a measured interval. The first interval of each column was measured immediately after the last one in the preceding column, while other periods in each column were measured immediately after

the preceding ones in the same column. For example, the longest interval of 11,964.9 μs (first one in the third column) in Table 1 is followed immediately by the shortest interval of 7,716.3 μs, which is in turned followed by a slightly longer interval of 10,316.9 μs. This can be explained by the use of absolute time inside the OS kernel. One timer firing is late (hence the longer interval) because of other higher-priority system activities. The RTOS still tries to fire the timer at its next originally-scheduled time. Therefore, the interval immediately following the longer one is shorter than the nominal value. In this example, the second timer firing is slightly early, which further shortens the second interval. The third firing is on time. The average of these three intervals is 9,999.4 μs, which is very close to the nominal period of 10 *ms*. Figure 4 illustrates this phenomenon.

Measured Interval (nanosecond)				
1	2	3	4	5
9993000	10314375	11964875	9749575	9993625
10005875	9998875	7716275	10248675	10002650
9675825	9999875	10316875	10002150	9677625
10321250	9999950	10003375	9997750	10315375
9681200	10001800	10001000	10007550	10002125

Table 2: Sample measured periods with VxWorks.

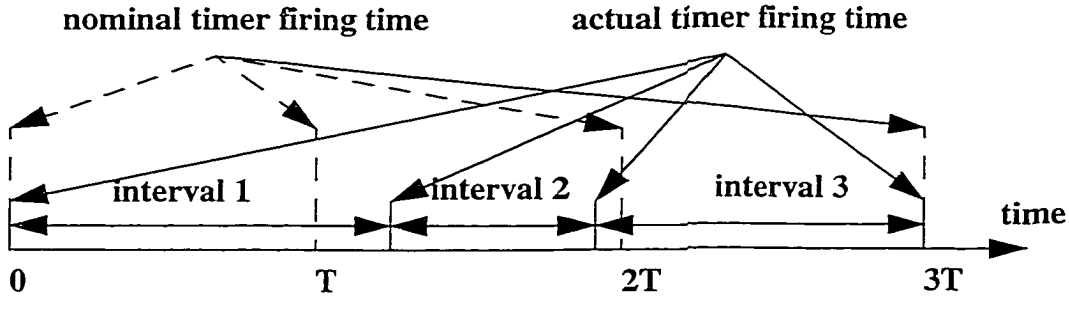


Figure 4: Timer "memory" behavior.

The nature of these two timer characteristics, variation and "memory" behavior, suggests that they are generic phenomena of RTOSs. To confirm this observation, we conduct similar timer experiments on two other RTOSs—QNX and pSOSystem.

2.2.2 QNX timer

QNX is a commercial, micro-kernel, POSIX-compliant RTOS [127]. It uses a priority-based, preemptive kernel scheduler. Our experiments are conducted on a XYCOM XVME-674/16 board running QNX version 4.22. This board is a VMEbus PC/AT processor module with an Intel 66 MHz 80486DX2, 32 MB dual-access DRAM, SVGA and IDE controllers. The QNX system clock resolution is set to 50 µs and the VME StopWatch is used for the timing measurement.

For the simple task with a 10 *ms* period (same as that in the VxWorks experiments) using POSIX timer functions, Table 3 gives measured task execution time and interval statistics. The QNX POSIX timer also exhibits interval variation. The timer "memory" behavior is seen as well, which can be illustrated by the following sequence of six consecutive timer intervals extracted from the measurement data: 9974.3, 9976.9, 10257.5, 9782.0, 9966.4, and 10005.0 µs. The first two intervals are close to the nominal value. The third interval is abnormally long, which is compensated by the next unusually-short interval. The fifth and sixth intervals are again close to the nominal value.

	Execution Time	**Interval**
Sample Size	1950	1949
Mean (µs)	4.27	9996.3
Standard Deviation (µs)	4.82	30.9
Min (µs)	1.45	9739.1
Max (µs)	16.00	10257.6

Table 3: QNX POSIX timer measurement statistics.

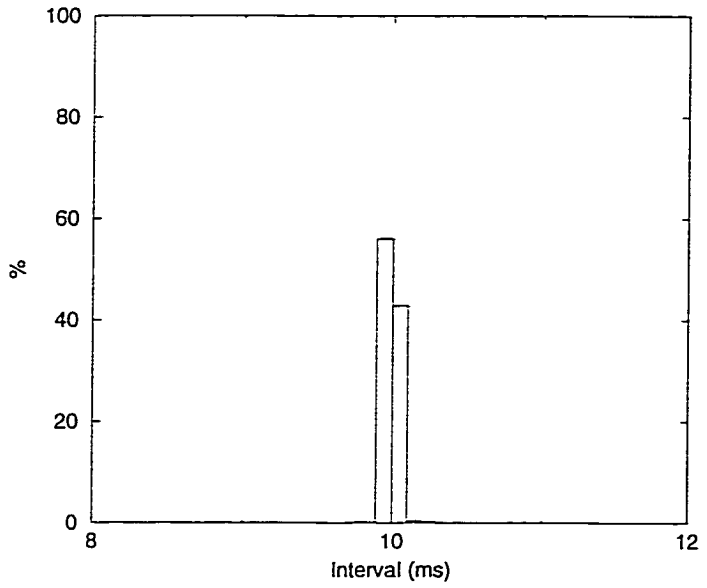


Figure 5: Histogram of QNX timer intervals (bin width: 100 μs).

2.2.3 pSOSystem timer

The pSOSystem is another commercial modular high-performance RTOS designed specifically for embedded microprocessors [63]. It includes a real-time multi-tasking kernel pSOS$^+$. Though the pSOSystem (version 1.1) is not POSIX-compliant, it has system calls similar to POSIX timer functions. We substitute the POSIX timer functions with pSOSystem-specific functions in our experiments.

We use a setup similar to that for VxWorks and QNX, where pSOS$^+$ runs on a VMEbus-based processor board—Ironics IV3207 (Motorola 25 MHz 68040 with 4 MB RAM). The system clock resolution of pSOS$^+$ is 10 *ms* (100 ticks/second). Again, timer interval variation (Table 4 and Figure 6) and "memory" behavior are observed in the measurement data.

	Execution Time	Interval
Sample Size	999	998
Mean (μs)	0.68	10000.2
Standard Deviation (μs)	0.011	82.5
Min (μs)	0.65	9764.2
Max (μs)	0.78	10244.8

Table 4: pSOSystem timer measurement statistics.

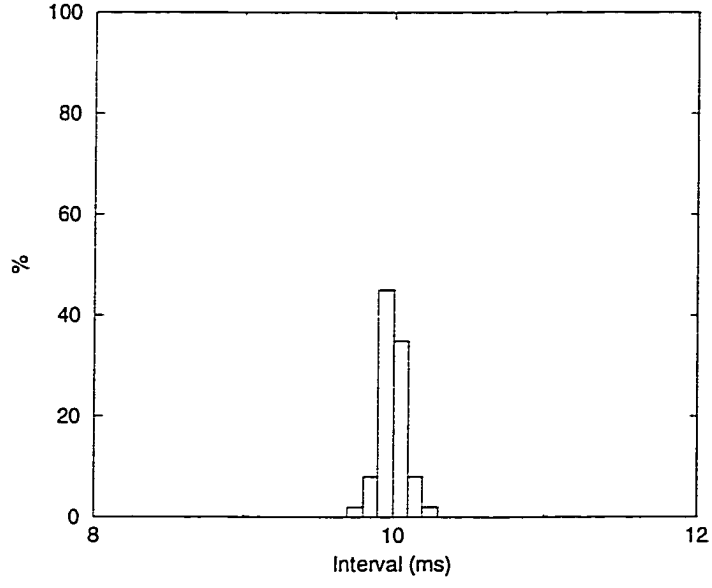


Figure 6: Histogram of pSOSystem timer intervals (bin width: 100 μs).

2.3 Task Execution Time Unpredictability

As we observed in the timer experiments, task execution times vary from one invocation to another even though they are supposed to be same. We now conduct more experiments to investigate the characteristics of task execution time variation.

We ran three sets of tasks, as shown in Table 5, in the VxWorks environment described earlier for the timer experiments. Each set was run at a time and there is no other user task in the system. Similar to the timer experiments described in Section 2.2, we

generate an event at the beginning and the end of a task. The elapsed time between these two events, called *measured execution time*, will include the execution time of a function call that generates the event, the execution time of the task, and the amount of time the task is preempted or blocked.

Task Set			task 1	task 2	task 3
1	N/A	period (*ms*)	10	N/A	N/A
		priority	20	N/A	N/A
2	RM	period (*ms*)	10	14	33
		priority	20	21	22
3	FIFO	period (*ms*)	10	14	33
		priority	20	20	20

Table 5: Task sets.

Individual tasks in these experiments have an identical contention-free execution time, because they use the same executable, but they can have different periods and priorities. The first set consists of a single task with a period of 10 *ms* and a priority of 20. The three tasks in the second set have periods of 10, 14 and 33 *ms*, respectively. They are scheduled using RM. In VxWorks, a smaller number represents a higher priority. The third task set is the same as the second, but using FIFO instead of RM.

In the three task sets, task 1 either runs alone or has higher or same priority than other tasks in the same set. Therefore, task 1 will not be preempted by either task 2 or task 3. In an idealized system with no system overhead or unpredictability, the measured execution times of task 1 in all three task sets should be the same.

Figures 7, 8 and 9 show the histograms of task 1 execution times for the three experiments, respectively. The horizontal axis is the measured execution time, while the vertical axis represents the percentage of task invocations with that amount of the measured execution time. For example, about 96% of measured task 1 execution times are between 1650 and 1700 µs when the task runs alone (Figure 7). However, they are

reduced to about 72% (Figure 8) and 60% (Figure 9) when the number of tasks increases from 1 to 3 and the tasks are scheduled by RM and FIFO, respectively. For the two experiments with three tasks, the distributions of task 1 execution times are also different when different scheduling algorithms are used, as depicted in Figures 8 and 9. Table 6 lists the measured statistics.

Clearly, the actual measured execution times of a task varies even though they should be the same in theory (which assumes no system overhead and unpredictability). Our experiments indicate that the actual measured execution times depend on not only the number of tasks running in the system but also how the tasks are scheduled (even when the concerned task has the highest priority). This makes theoretical analysis of the tasks extremely difficult. Again, the main reason for this phenomenon is that RTOS serves other system activities at higher priorities, which we will discuss in the next section.

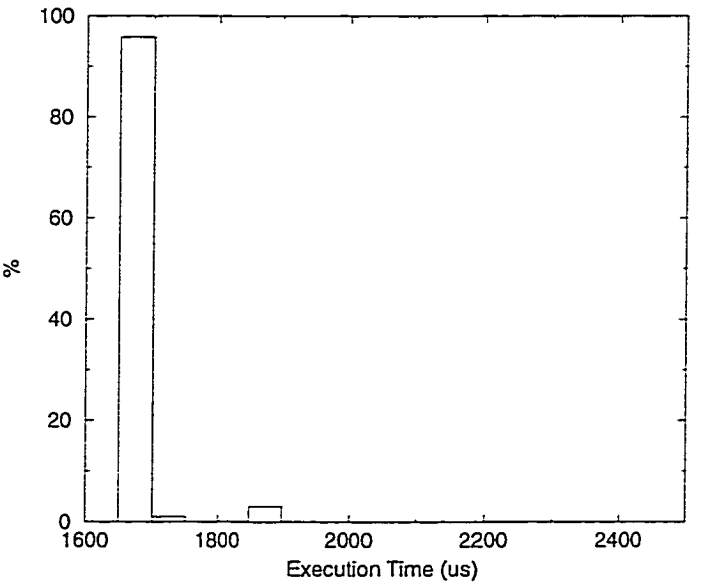


Figure 7: Measured execution time of task 1 in the first set (stand-alone).

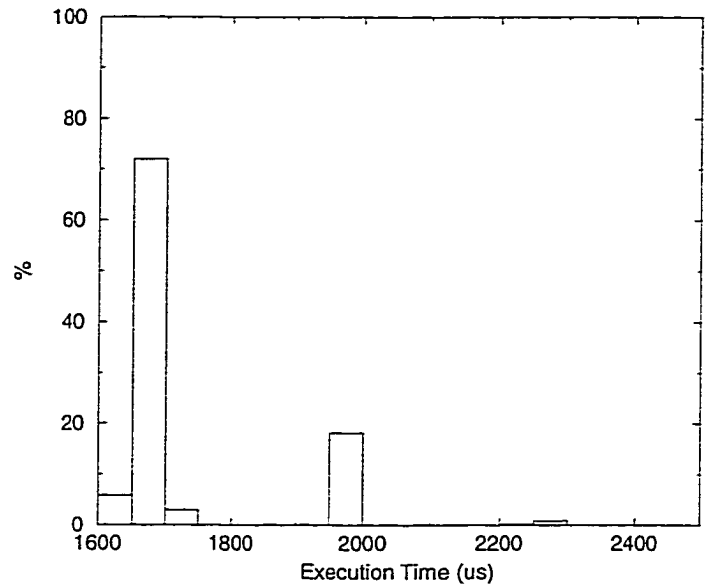


Figure 8: Measured execution time of task 1 in the second set (RM).

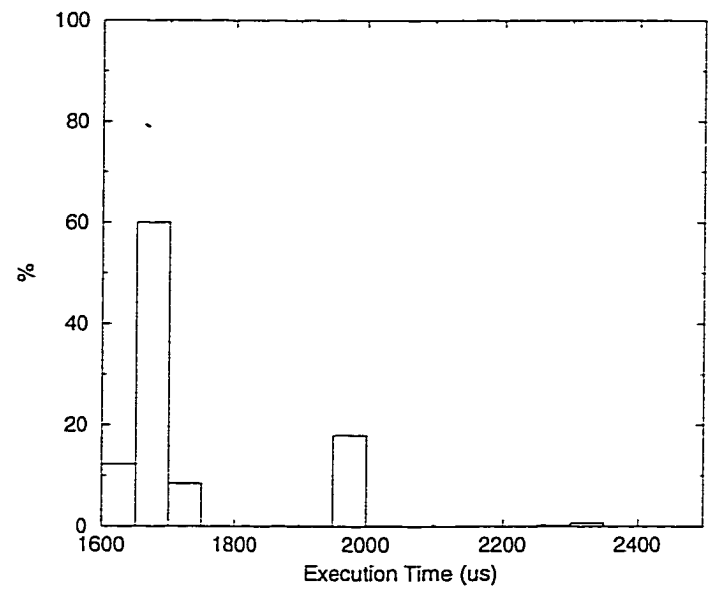


Figure 9: Measured execution time of task 1 in the third set (FIFO).

	task 1 execution time		
	first set	second set	third set
Sample Size	17503	10731	23123
Mean (μs)	1701.12	1750.21	1747.75
Standard Deviation (μs)	28.69	123.66	127.70
Min (μs)	1625.30	1625.00	1625.05
Max (μs)	2012.62	2299.62	2482.05

Table 6: Statistics of measured execution times of task 1.

2.4 Causes of Unpredictability

There is significant unpredictability in both timer intervals and task execution times, which is caused by OS activities. We now identify major classes of OS activities and examine their impacts on the variations of timer interval and task execution time.

2.4.1 OS services

An OS provides the environment within which programs are executed. As described in [120], the most common classes of OS services to programs and the users are given as follows:

- *Program Execution*: the system must be able to load a program, as a process, into memory and run it.
- *Input/Output Operations*: a running program may require input and output, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as rewind a tape drive). Since a user program cannot execute I/O operations directly, the OS must provide some means to do so.
- *File System Manipulation*: OS provides a uniform logical view of information storage, the *file*, by abstracting from the physical properties of its storage

devices. Files store data and programs, and are mapped, by the OS, onto physical devices.

- *Error Detection*: OS constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a disk write error), or in the user program (such as an arithmetic overflow). For each type of error, the OS should take the appropriate action to ensure correct and consistent computation.

- *Resource Allocation*: when there are multiple users or jobs running at the same time, resources (such as CPU cycles and memory) must be allocated to each of them.

- *Accounting*: OS may need to keep track of which users use how much and what kinds of computer resources, for the purpose of charging for the system usage or accumulating usage statistics.

- *Protection*: when there are multiple users or jobs, they should not interfere with each other. In addition, their access to the information stored in the computer system and various resources must be controlled.

2.4.2 OS interrupts

Operating systems are event-driven programs [120]. If there are no jobs to execute, no I/O devices to service, and no users to respond to, an OS will sit and wait for something to happen. Events are almost always signaled by the occurrence of an interrupt. Several different types of interrupts may occur:

- *System call*: an interrupt is generated when a system call is made to terminate the currently running program (normally or abnormally), to ask the OS for information (such as time), to request resource (such as CPU or memory), or to request an I/O operation.

- *I/O device interrupt*: an I/O device will interrupt CPU when it finishes an I/O request. A timer interrupt signals the passage of a given period of time.

- *Program error*: certain types of program error (such as an illegal instruction) cause hardware traps. The trap transfers control through the interrupt vector to the OS just like an interrupt.

The term "interrupt" is used here in a loose sense for any event that causes CPU to make a temporary transfer of control from the currently running program to another that services the event.

The basic method of interrupting the CPU is to activate a control line that connects the interrupt source to the CPU. The interrupt signal is then stored in a CPU register which is tested periodically, usually at the end of every instruction cycle. Each interrupt source may require the execution of a different interrupt service routine (ISR). On detecting the presence of an interrupt request, the CPU executes the ISR associated with the interrupt.

The ISRs and processes get CPU cycles in the following order:

1. ISR associated with the highest priority interrupt.
2. ISR associated with the next highest priority interrupt.
3. ...
4. ISR associated with the lowest priority interrupt.
5. process with the highest priority.
6. process with the next highest priority.
7. ...

The reason that ISRs have higher priorities than all system and user processes is the hardware architecture. CPUs are designed to jump to ISRs automatically on detection of any interrupts. Because of this, a phenomenon called *priority inversion* could occur, where a higher-priority task could be preempted or blocked by a lower-priority task for an unbounded amount of time [137]. Suppose an application task process runs at a higher

priority than a keyboard driver process. If there are keyboard inputs during the execution of the task, interrupts generated by the keyboard will invoke the ISR attached to the keyboard interrupts. Since the ISR has higher priority, it will preempt the application task process. Therefore, the higher-priority task is effectively preempted by the lower-priority keyboard activities. Priority inversion is undesirable for real-time applications because it can cause, for example, unpredictable task completion times.

2.4.3 Major classes of RTOS activities

For real-time systems, we identify three major classes of activities — I/O interrupt handling, process scheduling and timer management — that have significant impacts on task execution and timer interval variations.

Interrupts are the primary means for I/O devices to obtain CPU services. They are used to request the CPU to initiate a new I/O operation, to signal the completion of an I/O operation, and to signal the detection of hardware or software errors. Interrupts improve computing performance by allowing I/O devices direct and rapid access to the CPU and by freeing the CPU from the need of continually testing the status of I/O devices, since I/O devices are typically much slower than the CPU.

Whenever a job is terminated or released, the OS is interrupted to schedule "jobs" on the CPU. The kernel scheduler is invoked whenever a process changes state, namely, when a process becomes unblocked, or the time-slice for a running process expires, or a running process is preempted. Unlike processes, the kernel itself is never scheduled for execution. It is entered only as the direct result of kernel system calls, either from a process or from a hardware interrupt. It decides which process to execute next. In most RTOSs, the kernel scheduler uses priority-based, preemptive policies.[2] This fact explains the phenomenon in Section 2.3 where the task execution time distribution is dependent on

2. Note that the kernel scheduler does not assign priorities to individual processes, which is the responsibility of the user. Instead, it merely executes the user-specified schedule.

the system load and scheduling policies. For example, during the execution of task 1 of the second task set in Table 5, tasks 2 and 3 may be released, thus calling the kernel scheduler. Even though the kernel scheduler will still choose task 1 to run because it has the highest priority, some CPU time is consumed to make the decision. Therefore, the completion of task 1 is delayed by the OS scheduling activities.

Since we are particularly interested in OS timer service behavior, we single out activities associated with timer interrupts. Note that periodic real-time tasks typically use the OS timer service to run periodically in a multitasking environment. The performance of OS timer management is reflected directly in the timer interval variation. The granularity of timers is determined by the system clock tick size, which is the rate at which timer interrupts (called *ticks*) are generated. Note that the system clock tick is not the same as the hardware clock tick, which is much more frequent. A timer interrupt signals the passage of some interval of time. At every system clock tick, the CPU is interrupted by a timer interrupt and runs an ISR for the timer interrupts, which, in turn, examines the existing software timers to see if any of them has expired. If a timer has expired, the OS sends a signal to the process waiting on the timer; otherwise, it does nothing. So, all time requests via software timers are rounded up to the timer granularity.

2.4.4 Relative significance of RTOS activities

Interrupts and process scheduling are handled by the OS kernel. In VxWorks and pSOSystem, timer management is also the responsibility of the kernel. In QNX (version 4.23), however, timers are maintained by the process manager (named *Proc32*), which runs as a real process. One can change the priority of the QNX process manager, thus allowing us to examine the relative significance of the kernel and process manager activities, in terms of their impact on the variations of task interval and execution time. If a user task has a higher priority than the process manager, it should be able to avoid any

negative impact by the process manager activities, except ISRs in the process manager, which always have higher priorities than processes.

For our experiment, we assigned the process manager (Proc32) a priority of 26. We then ran a user task with a period of 10 *ms* at a lower priority (which is chosen to be 25) than that of the process manager. All other system processes, including device drivers, have priorities of 24 or lower. In QNX, bigger numbers represent higher priorities. After measuring the performance of the user task, we then ran the same task at a higher priority (which is chosen to be 27 but can be any other higher priority) and measured its performance.

Furthermore, since the timer management activities occur at every system clock tick, this overhead should decrease as the tick size increases. To verify this conjecture, we ran our user task with tick sizes ranging from 50 μs to 5000 μs. Tables 7 and 8 list the measured performance with priorities of 25 and 27, respectively. For example, the first row of Table 7 shows that the mean execution time of the task with a priority of 25 is 560.4 μs, the standard deviation of its execution time is 8.0 μs, while the mean and standard deviation of the task interval are 9994.9 μs and 61.9 μs, respectively.

To better understand our measurement results, we plot the impacts of tick size on task interval and execution time in Figures 10 and 11, respectively. The "First set" of data shown in the figures is the same as those listed in Tables 7 and 8. We re-ran the same experiments and plotted their measurement data in Figures 10 and 11 as the "Second set." In Figure 10, the vertical axis of the top graph represents the mean of task interval, which should be as close to 10 *ms* as possible. The vertical axis of the bottom graph is the (logarithmic) standard deviation of task interval, which should be as small as possible. The horizontal axes for both graphs are the (logarithmic) system tick size. Similarly, the vertical axes of the top and bottom graphs of Figure 11 are the mean and standard deviation of task execution time, respectively, while their horizontal axes are the (logarithmic) system tick size.

Tick Size (μs)	User Task with a Priority of 25 and a Period of 10 *ms*					
	Execution Time			Interval		
	# sample	mean (μs)	std dev (μs)	# sample	mean (μs)	std dev (μs)
50	975	560.4	8.0	951	9994.9	61.9
100	974	477.4	9.1	949	9994.8	45.8
200	975	445.5	3.0	950	9994.3	93.8
400	974	434.2	4.8	949	9995.1	64.7
800	975	420.6	1.9	951	9995.2	400.4
1000	975	421.4	4.1	950	9993.0	48.3
1600	976	421.0	2.7	951	9994.4	686.8
2000	975	420.7	2.6	950	9994.1	55.0
3200	975	421.1	2.9	951	9989.9	1055.9
5000	975	421.3	5.6	950	9993.7	22.8

Table 7: Measured statistics of task with priority 25 under QNX.

Tick Size (μs)	User Task with a Priority of 27 and a Period of 10 *ms*					
	Execution Time			Interval		
	# sample	mean (μs)	std dev (μs)	# sample	mean (μs)	std dev (μs)
50	975	575.3	25.3	950	9994.8	23.8
100	975	481.6	13.6	951	9994.8	43.9
200	974	446.2	4.9	949	9994.1	68.0
400	974	433.5	3.6	949	9994.9	49.6
800	974	420.9	3.5	949	9995.2	400.2
1000	975	420.4	1.9	950	9994.2	41.2
1600	975	421.5	6.1	950	9995.7	689.8
2000	974	421.2	4.2	949	9993.8	26.3
3200	976	420.5	3.3	951	9994.3	1064.6
5000	976	420.9	3.5	951	9996.2	127.3

Table 8: Measured statistics of task with priority 27 under QNX.

There are a few interesting observations to make from these measurement. First, there is little performance difference between the task with a priority higher than the QNX

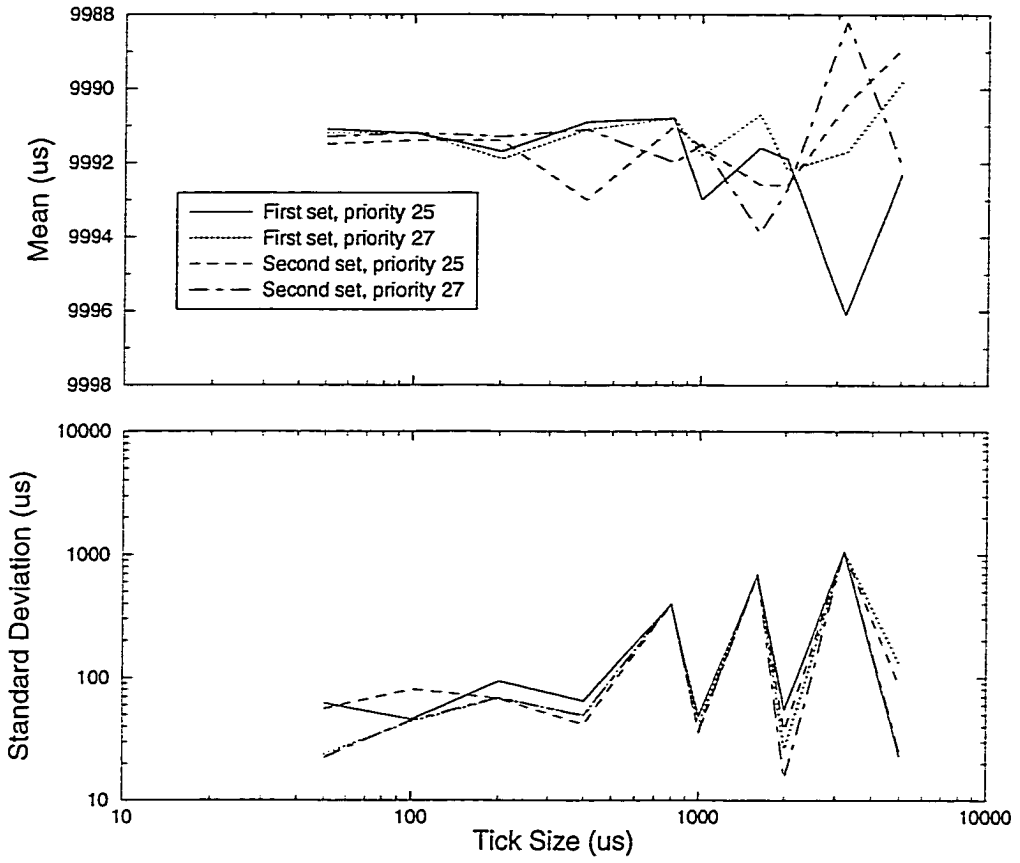


Figure 10: Impact of tick size on task interval.

process manager and that with a lower priority. This indicates that the impact of the activities of the process manager (except its ISRs) on the variations of task interval and execution time is minimal. The main sources of system unpredictability must then come from I/O interrupt handling, process scheduling and other ISRs, because they always preempt system and user processes, regardless the priority of the process manager (which runs as a regular user process).

Second, the mean values of the task interval are very close to the nominal period, as shown in Figure 10. They are within about 10 μs or 0.1% of the nominal period of 10 *ms*. As the tick size increases, the mean values tend to fluctuate a little more from one measurement to another, perhaps due to the less frequent examination of the timers.

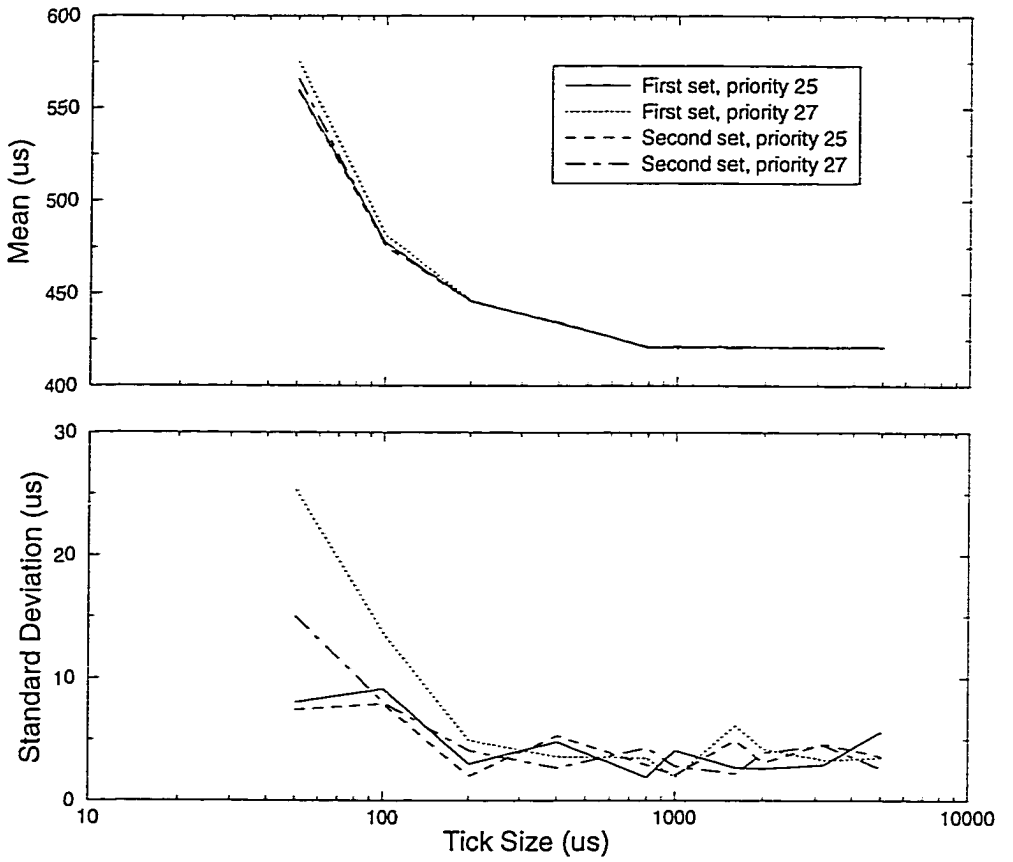


Figure 11: Impact of tick size on task execution time.

More importantly, Figure 10 shows significantly large standard deviations of the task interval when the tick sizes cannot divide the nominal period. For example, the interval standard deviation of the task with a priority of 25 and a nominal period of 10 *ms* is 400.4 μs, when the tick size is 800 μs. When the tick sizes are 400 μs and 1000 μs, the standard deviation values are dramatically reduced to 64.7 μs and 48.3 μs, respectively. Clearly, if the nominal period is not an integral multiple of the tick size, the timers will be fired either earlier or later than the desired time instants. In this case, the larger the tick size, the bigger the interval standard deviation, as shown in Figure 10.

Another interesting phenomenon is that the mean task execution time tends to converge to some constant (Figure 11), as the tick size increases. This indicates that the timer management activities do have a significant impact on task execution time. If a timer

interrupt occurs during the execution of the task, the task will be preempted by the ISR attached to the timer interrupt, which will consume some CPU cycles to examine the timers, thus delaying the completion of the task. When the tick size increases, such preemptions will occur less frequently and the measured task execution time will shorten. The measured task execution will converge to its contention-free execution time, which is a constant in our experiments. In Figure 11, the standard deviation of the task execution time is also larger when the tick size is small. It remains in a small range when the tick size is larger.

In summary, we identified three classes of OS activities I/O — interrupts handling, process scheduling and timer managing — as the main sources of variations in task interval and execution time. These OS activities take precedence regardless of the priority of the user task. Our experiments showed that it is important to set system tick size such that the nominal task periods are integral multiples of the tick size, in order to reduce the timer interval variation. Using relatively large tick size also helps reduce the measured task execution time.

2.5 Summary

In a multitasking environment, RTOS software timers are typically used to implement periodic real-time tasks. Because these tasks have deadlines, it is also essential to have an accurate estimate of the actual task execution times in order to assess the schedulability of the tasks. Therefore, the performance of a real-time task (in terms of its ability to meet its deadline) can be highly dependent on the predictability of RTOS timers and task execution time.

However, from our experimental measurement data, we observed that both RTOS timer intervals and task execution times exhibit significant unpredictability. Timer intervals can vary and have "memory" behavior, while task execution times can also

fluctuate. We identified three classes of OS activities, I/O interrupt handling, process scheduling, and timer managing, as the main sources of disturbance for the timer interval and task execution time variations.

The OS performance is also sensitive to the system clock resolution (tick size). When the resolution decreases (i.e., system clock tick size increases), the task execution time improves as reflected by smaller mean. The timer interval variation increases dramatically, if the nominal period is not an integral multiple of the tick size. Therefore, it is important to set system tick size such that the nominal task periods are integral multiples of the tick size, in order to reduce the timer interval variation. Using relatively large tick size also helps reduce the measured task execution time.

CHAPTER 3

HARD DEADLINE GUARANTEES IN THE PRESENCE OF TIMING UNPREDICTABILITY

3.1 Introduction

Since a real-time open-architecture controller typically consists of a set of cooperative periodic tasks, scheduling such tasks is of great importance. These periodic tasks must typically be completed by the end of their respective period.

According to our target application requirements, most control tasks have probabilistic deadlines while a few (e.g., an emergency shutdown task) have hard deadlines. In this chapter, we will investigate approaches of providing hard deadline guarantees, based on the assumption that all task deadlines are hard. In the next chapter, we will relax this restriction to include both hard and probabilistic deadlines and explore issues related to providing probabilistic deadline guarantees.

Many controller tasks have data dependencies among them. For example, the inputs of the control-law tasks depend on the outputs of the sensor-reading tasks. However, it has been shown that the scheduling problem for tasks with data dependencies is generally intractable [104, 152]. For tasks that have data dependencies with very short critical sections or use a nonblocking messaging mechanism to transfer shared data, they can be approximated as independent tasks. For example, a control-law calculation task with a 10 milliseconds period may need only a few microseconds of exclusive access to the shared data in order to get sensor readings and update control commands. The

probability that it will block either the sensor-reading task or the actuator task is very small. If the shared data have multiple versions and the controller can tolerate the use of slightly outdated data, there will be no blocking at all. This approximation of tasks with nonblocking data dependencies as independent tasks is necessary and crucial in that it reduces an intractable scheduling problem to one that is more manageable and still practically acceptable.

For a set of independent, preemptive, periodic tasks, whose relative hard deadlines are the same as their respective periods, Liu and Layland [93] proved that the *rate-monotonic* (RM) priority assignment is an optimal static-priority[1] scheduling algorithm. RM dictates that tasks with shorter periods will have higher priorities. It has an elegant sufficiency condition for task schedulability: if the overall CPU utilization is no greater than a certain threshold, which is a function of the number of tasks, all tasks are guaranteed to meet their deadlines. RM is optimal in the sense that no other fixed priority assignment can schedule a task set (guaranteeing all hard deadlines) which cannot be scheduled by RM.

They [93] also proved that *earliest-deadline-first* (EDF) is an optimal dynamic-priority scheduling algorithm for a set of independent and preemptive tasks with hard deadlines. At any given time, EDF assigns the highest priority to the task with the earliest deadline. EDF can achieve up to 100% of CPU utilization, which is a significant improvement over RM.

Both RM and EDF assume deterministic and precise task release times and negligible run-time scheduling and context switching overhead. However, the precise moments of RTOS timer firings can deviate from the nominal time instants, thus introducing variation in actual task intervals. This could adversely affect the performance

1. A scheduling algorithm is said to be *static* if the priorities cannot be changed once they are assigned to the tasks; otherwise, it is *dynamic*.

of scheduling algorithms in terms of deadline miss ratios. It is unclear how well RM and EDF perform under such a condition.

In Section 3.2, we will study the performance of most common scheduling algorithms—RM, EDF and FIFO (*first-in-first-out*)—in the presence of timing unpredictability. Our simulation and measurement results show that RM is as good as EDF and better than FIFO in meeting task deadlines. This is because that, in situations where not all tasks meet their deadlines, EDF may unnecessarily assign higher priorities to tasks that will surely miss their deadlines. FIFO has larger deadline miss ratios than RM and EDF under high system load, because it gives no preferential treatment to tasks with shorter periods. When system load increases, such tasks are more likely to miss their deadlines since they may have to wait for other less urgent tasks to complete. In terms of task interval uniformity, the performance of RM is also comparable to others. In addition, RM incurs little run-time scheduling overhead. Therefore, RM is chosen for our open-architecture controller research and development.

In the presence of RTOS timing unpredictability, however, the original RM theory is no longer valid and needs to be adjusted. In Section 3.3, we propose an empirical task schedulability model, called RMTU (*Rate-Monotonic in the presence of Timing Unpredictability*), to augment the original RM scheduling algorithm to handle timing unpredictability. The model parameters are determined empirically and systematically by running and measuring a set of simple tasks on the target system. The model is empirical because its parameters are derived from measurement data. It is also systematic because it includes a set of systematic experiments, which can be applied to different target systems.

Measurements of additional experiments with multiple tasks confirm the validity of our empirical model and the derived model parameters. With RMTU, RM can now be extended to provide hard deadline guarantees for tasks in practical computing environments, where timers are not perfect. Using our model, task schedulability can be easily checked and, if possible, empirically guaranteed.

3.2 Scheduling Algorithm Performance in the Presence of Timing Unpredictability

In this section, we study the performance of real-time scheduling algorithms in the presence of timing unpredictability and determine which one(s) should be used for our open-architecture controller research and development.

Because of the nature of timing unpredictability and the complexity of task scheduling, it is extremely difficult, if not impossible, to use analytic approaches. For this reason, we use both simulations and experimental measurements. We examine the performance of RM, EDF and FIFO, in terms of task deadline miss ratios, which is defined as the percentage of jobs (invocations of tasks) that miss their deadlines. The reason for choosing RM and EDF is that they are optimal under their respective assumptions and there are no known scheduling algorithms that are specifically designed to handle RTOS unpredictability. Since FIFO provides bounded task response time (assuming all task execution times are bounded), which is desirable for real-time applications, it is also chosen for our study.

3.2.1 Simulations

We first run simulations to study the performance of scheduling algorithms. The main advantage of simulation is that it allows us to isolate the impacts of individual factors that may affect the performance.

Table 9 lists all simulation parameters. Task intervals are assumed to follow approximately a normal distribution. Note that this interval variation is an inherent problem of RTOS timers but not part of application specifications or scheduling algorithm features.

The parameter we use to model timer interval variation is the standard deviation (σ) of the normal distribution. Intuitively, the variation should be within a few ticks of the system clock regardless of the length of the nominal period, because the software timer

Simulation Parameter	Notation
seed for random number generator	s
number of tasks	n
task execution time	C
task deadline	D
minimum period	T_{min}
maximum period	T_{max}
timer interval standard deviation	σ
random start	r

Table 9: Simulation parameters.

implementation is independent of the nominal period.[2] Therefore, we use the same standard deviation for all task periods and limit the maximum variation to 3σ. These choices of simulation parameters are justified by the experimental measurement results presented in Chapter 2.

Randomness of the task start is another parameter we use. All tasks start at the same time 0 by default; but they may also start at any time between 0 and the end of their respective period minus their execution time with an equal probability. The former condition represents the critical instant. The critical instant for a task is defined to be an instant at which a request for that task will have the largest response time [93], which occurs whenever the task is requested simultaneously with requests for all higher-priority tasks. The latter condition represents an environment where tasks may be equally likely to start at any given time.

All tasks in one simulation have the same execution time. Tasks are fully preemptive, though context switching overhead is assumed negligible.[3]

2. This conjecture is verified by the experimental measurements described in Section 3.3.1.

We first concentrate on the effect of aforementioned parameters on the miss ratio. We then examine the performance of the scheduling algorithms under different system loads.

3.2.1.1 Impacts on deadline miss ratios

In this section, we investigate the impacts of different parameters on the task deadline miss ratio. For this set of simulations, the parameters are set as follows:

- Seed: -1, used to initialize the random number generator.
- Number of tasks: 70.
- Execution time of each task: 5 μs.
- Min (max) period: 100 μs (1000 μs). Task periods are selected between min and max with equal probability.
- Deadline: the end of each period.

Figure 12 shows the period distribution of the set of tasks used in the simulations. The horizontal axis is task period, while the vertical axis is the number of tasks with any given period. This task set represents a high system load, where RM cannot guarantee all deadlines.[4] Here, we want to examine the effects of timer interval variation and random start under different scheduling algorithms. We will study the effect of different loads later.

---

3. In the context of RTDB, data access (i.e., read and write) operations to shared data objects are typically periodic in open-architecture controllers. These operations can also be modeled as periodic tasks. However, since such operations are atomic and very shorter (no more than a few microseconds), it is therefore more appropriate to treat them as non-preemptive operations. We conducted simulations for non-preemptive tasks as well. Our findings are similar to those for preemptive tasks.

4. Using EQ 1 in Section 3.3, the deadlines of 70 independent, preemptive, periodic tasks can only be guaranteed under RM if the CPU utilization is no greater than about 0.70.

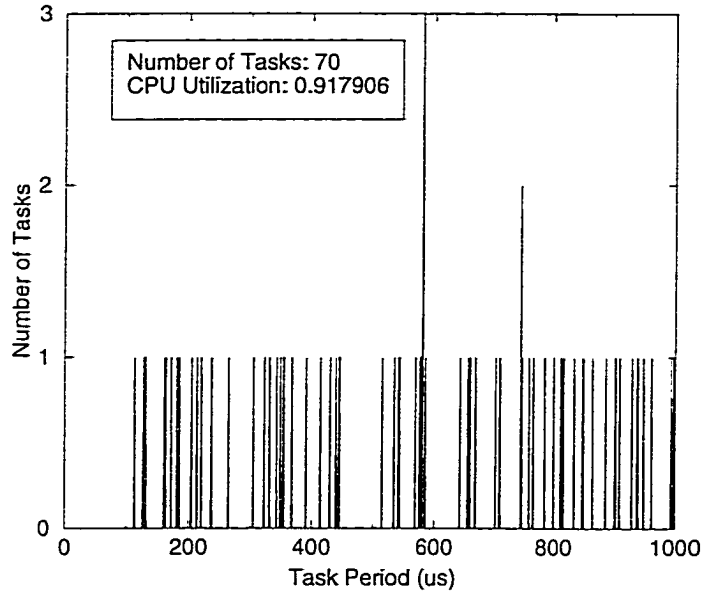


Figure 12: Distribution of task periods.

Table 10 illustrates the impacts of random task start and interval variation on task deadline miss ratios. These simulations clearly reveal the following phenomena:

1. While tasks may miss their deadlines, the miss ratios are small (less than 1%). This means that RM, EDF and FIFO perform quite well in the presence of timer interval variation, though they may no longer be able to guarantee all hard deadlines.

2. Random start of the tasks helps when there is no interval variation. This is no surprise, because random starts generally reduce the number of critical instants, thus the task is less likely to miss its deadline. When there is interval variation, random starts do not necessarily help task deadline miss ratios. Because interval variation without random starts could reduce the chance that tasks are released at the same time, the aggregate effect of interval variation and random starts could *increase* the chance of conflicts.

3. The variation of task periods has a significant impact. We observe that, when timer interval variation is introduced, either tasks start to miss their deadlines or there is a significant (more than 20 times) increase in miss ratios.[5] This is

because a job may be released late, thus leaving less time available before its deadline.

scheduler	random start	standard deviation (μs)	# jobs	miss ratio
RM	N	0	183104	0.009%
	Y	0	183076	0%
	N	50	183057	0.199%
	Y	50	183078	0.175%
EDF	N	0	183099	0%
	Y	0	183076	0%
	N	50	183084	0.199%
	Y	50	183077	0.175%
FIFO	N	0	183068	0.018%
	Y	0	183058	0%
	N	50	183074	0.461%
	Y	50	183053	0.417%

Table 10: Impacts of task random start and interval variation on miss ratios.

3.2.1.2 Effects of timer resets

RTOS timers exhibit "memory" behavior. When a longer interval is generated due to system unpredictability, it is typically succeeded by a shorter one. Intuitively, resetting the timer after each firing would eliminate the dependencies between task intervals, thus removing the source of the "memory" behavior. Because the tasks have also deadline constraints, we need to examine the effect of resetting timers on task deadline miss ratios as well. We use the same simulation parameters in Table 9, except that the timer is reset

5. While the absolute difference is small (less than 0.5%) between the miss ratios with and without interval variation, it is statistically significant. If we consider the task completion time as a discreet process with only two possible outcomes (meeting and missing deadline) and apply the Chi-square test, we can find that the task completion time distributions are statistically different with and without interval variation.

after each firing. We assume that the overhead of resetting timers is negligible. As we will show later in this section, this assumption does not affect our conclusions.

Table 11 tabulates the simulation results for the same set of tasks in the previous section. Comparing Tables 11 and 10, it is evident that resetting timers causes a significant increase in task deadline miss ratios in the presence of timer interval variation. For example, the last row of Table 11 shows that the miss ratio is 38.44%, while it is only 0.417% in Table 10 for the same task set under the same conditions except timer resets.

scheduler	random start	standard deviation (μs)	# jobs	miss ratio
RM	N	0	183104	0.009%
	Y	0	183076	0
	N	50	183182	42.567%
	Y	50	183178	47.973%
EDF	N	0	183099	0
	Y	0	183076	0
	N	50	183279	37.148%
	Y	50	182872	41.749%
FIFO	N	0	183068	0.018%
	Y	0	183058	0
	N	50	183173	48.181%
	Y	50	182843	38.435%

Table 11: Impacts of task random start and interval variation with timer resets.

Without timer resets, there is only one Gaussian random variable[6] that affects the job release times. That is, the n-th release time has a mean of $(n\text{-}1)T$ (where T is the nominal period) and a standard deviation of σ. However, with timer resets, each timer firing is independent of other firings except they all have the same nominal period. In other words, the time interval between any two consecutive timer firings is independently

6. Strictly speaking, it is not a Gaussian random variable because we limit the maximum variation to 3σ.

determined. Therefore, the release time of the n-th job of a task is the sum of n-1 independent Gaussian variables, each of which has a mean of T and a standard deviation of σ. The overall effect is that the n-th release time has a mean of $(n\text{-}1)T$ and a standard deviation of $\sigma\sqrt{n-1}$. The larger standard deviation is responsible for the significant increase of task miss ratios.

Clearly, resetting timers is not a viable solution because of the significant increases in task deadline miss ratios, even when the timer operation overhead is ignored. In a practical system, the timer reset overhead can be significant. Table 12 shows the measured results of a task with a nominal period of one millisecond. Each timer reset function call can take more than 100 μs. If this overhead is taken into account in our simulations, the task miss ratios will be even worse.

	Execution Time	interval
Sample Size	2043	2042
Nominal Value (μs)	0	1000
Mean (μs)	168.6	1150.9
Standard Deviation (μs)	22.0	46.4
Min (μs)	152.0	395.4
Max (μs)	362.6	1984.9

Table 12: Overhead of timer resets.

3.2.1.3 Performance comparison

From the above simulations, we see that the major impact on task deadline miss ratios comes from the variation of timer firings. As proven in [93], EDF is better than RM in that it can achieve up to 100% of CPU utilization while guaranteeing all deadlines. However, it is not clear how EDF and RM will perform, relative to each other, in the presence of interval variation, which may cause jobs to miss their deadlines. This set of simulations attempts to ascertain whether one of the two algorithms outperforms the other in the given

environment. We also include FIFO because it has the desirable characteristic of bounded response time (assuming all task execution times are bounded).

In the simulations, we vary the number of tasks, the scheduling algorithm, and the seed for the random number generator, while keeping the following parameters constant. These parameters are chosen to approximate our application development environment.

- Random start of tasks: No.
- Execution time of each task: 5 μs.
- Min period: 100 μs.
- Max period: 1000 μs.
- Standard deviation of periods: 50 μs.
- Deadline: the end of each period.

Table 13 lists the simulation results of the three scheduling algorithms using three different seeds to initialize the random number generator. These seeds are selected arbitrarily and they cause the random number generator to produce different sequences of random numbers conforming to the specified probability distribution.

Our first observation is that some tasks missed their deadlines even when the scheduling theory guarantees all deadlines. For example, in the first experiment (first row of Table 13), the CPU utilization of the 10 tasks is only 0.121. If there were no timing unpredictability, all tasks would have made their deadlines using any of the three scheduling algorithms (RM, EDF and FIFO). Second, all miss ratios are very small: less than 1% in our simulations.

We now take a closer look at the relative performance of RM, EDF and FIFO. Figures 13, 14 and 15 depict their miss ratios for various system load in three sets of simulations, with a seed of -1, -2222 and -77, respectively.

seed	# tasks	CPU utilization	RM		EDF		FIFO	
			miss ratio	sample size	miss ratio	sample size	miss ratio	sample size
-1	10	0.121	0.230%	22629	0.203%	22629	0.221%	22629
	20	0.225	0.088%	42118	0.083%	42117	0.093%	42121
	30	0.345	0.057%	64665	0.080%	64660	0.068%	64657
	40	0.468	0.094%	93429	0.094%	93431	0.090%	93436
	50	0.616	0.068%	122874	0.096%	122858	0.092%	122881
	60	0.754	0.134%	150407	0.124%	150400	0.219%	150399
	70	0.918	0.199%	183057	0.199%	183084	0.461%	183074
-2222	10	0.089	0%	16634	0%	16634	0%	16635
	20	0.199	0%	38253	0%	38252	0%	38251
	30	0.365	0.132%	72640	0.110%	72636	0.138%	72639
	40	0.506	0.098%	100735	0.098%	100734	0.113%	100731
	50	0.589	0.089%	117277	0.079%	117282	0.082%	117275
	60	0.720	0.073%	143511	0.068%	143516	0.105%	143511
	70	0.852	0.210%	169694	0.191%	169710	0.334%	169690
	80	0.957	0.259%	190641	0.243%	190665	0.644%	190658
-77	10	0.126	0.020%	20250	0.020%	20250	0.015%	20249
	20	0.258	0.427%	46857	0.427%	46853	0.446%	46857
	30	0.344	0.344%	64323	0.345%	64320	0.364%	64323
	40	0.447	0.267%	83646	0.279%	83651	0.259%	83646
	50	0.584	0.202%	115506	0.224%	115517	0.231%	115509
	60	0.674	0.175%	134538	0.169%	134521	0.208%	134520
	70	0.765	0.166%	152833	0.162%	152836	0.228%	152822
	80	0.922	0.185%	184145	0.175%	184148	0.370%	184145

Table 13: Performance of scheduling algorithms under different system load.

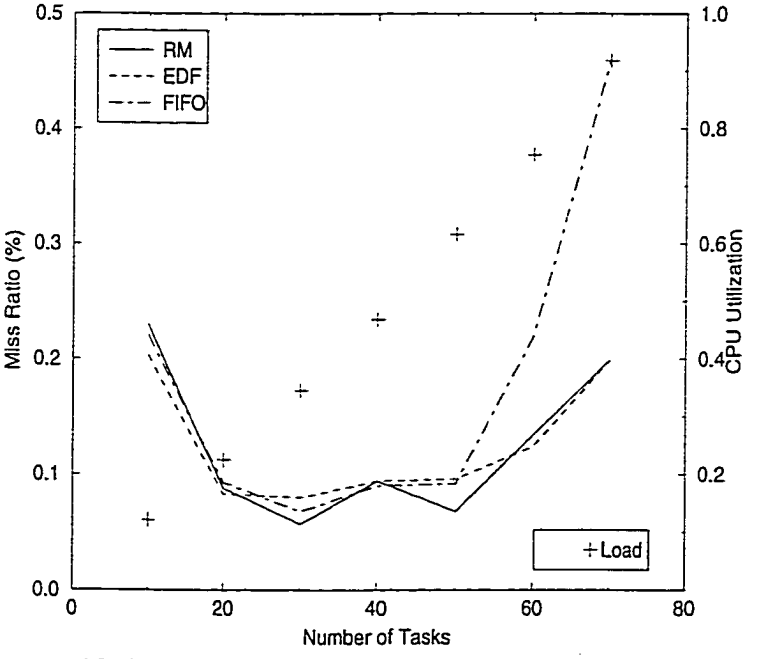


Figure 13: Performance of scheduling algorithms (seed: -1).

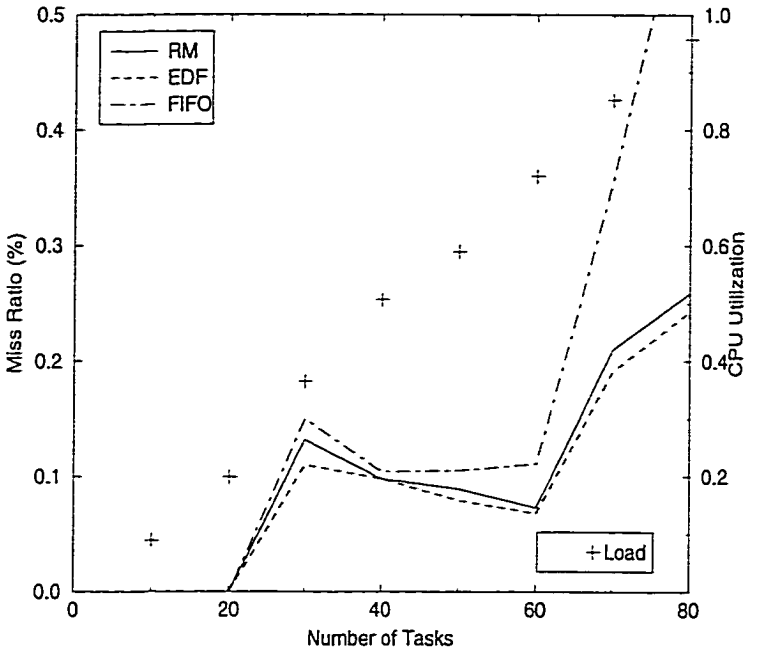


Figure 14: Performance of scheduling algorithms (seed: -2222).

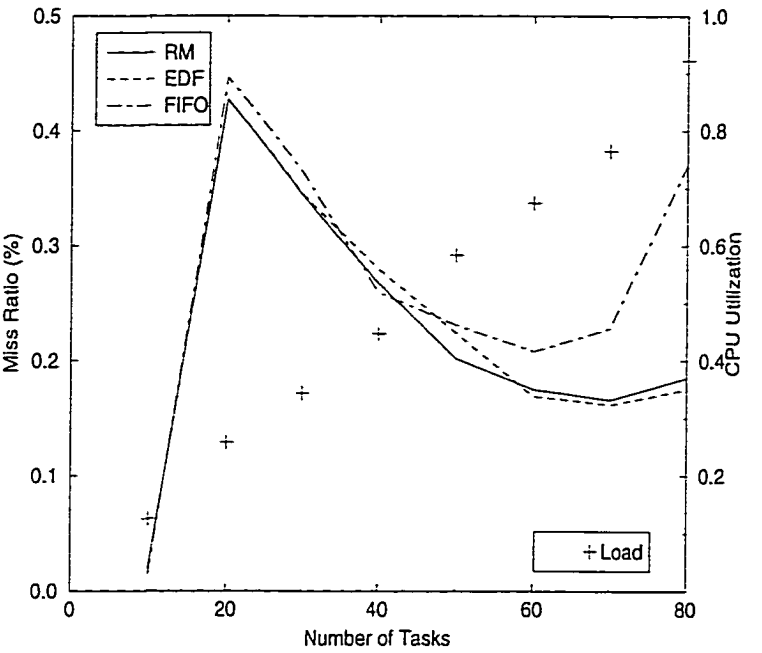


Figure 15: Performance of scheduling algorithms (seed: -77).

There are two interesting observations from these figures. First, the miss ratios of FIFO are significantly larger than that of RM and EDF under high system load, while all three scheduling algorithms have similar performance under low system load. This is because FIFO does not use any task deadline information. With FIFO, tasks with shorter

periods compete with other tasks for the CPU on an equal basis, but their deadlines tend to be in a more immediate future. While the CPU may be able to handle all tasks with little delay under light system load, the contention becomes more significant under heavy system load, thus causing more tasks with shorter periods to miss their deadlines. This is evident in Table 14, which lists the number of deadline misses for individual tasks. The table shows one low system load case and one high system load case. Only a few tasks with the shortest periods are listed in both cases. This observation indicates that RM and EDF are superior to FIFO.

seed	# tasks	CPU utilization	period (μs)	RM		EDF		FIFO	
				# misses	sample size	# misses	sample size	# misses	sample size
-1	10	0.121	130	52	7209	46	7209	50	7209
			332	0	2823	0	2823	0	2823
		total		52	22629	46	22629	50	22629
	70	0.918	113	153	8823	158	8824	303	8824
			127	64	7851	62	7852	142	7851
			129	38	7729	53	7730	130	7730
			130	47	7670	48	7671	107	7670
			131	44	7611	44	7612	112	7612
			163	2	6117	0	6118	15	6117
		total		364	183057	365	183084	844	183074

Table 14: Deadline misses of individual tasks.

Second, in the presence of timing unpredictability, RM and EDF have very close miss ratios regardless of system load. The theoretical performance advantage of EDF over RM is derived based on the assumption that all tasks meet their deadlines. However, the situation is quite different when there are tasks missing their deadlines. Tasks that are about to miss their deadlines tend to have earlier deadlines and, therefore, will be selected by EDF to run first. But this does not help the overall deadline miss ratios. For tasks that

are sure to miss their deadlines, it is better to drop them such that other tasks have a better chance to make their deadlines.[7] On the other hand, RM assigns tasks with shorter periods higher priorities, regardless of the urgency of their deadlines relative to others. Since such tasks are more sensitive to timer interval variation, RM effectively becomes a interval-variation-sensitivity-monotonic priority assignment, which intuitively helps reduce the effect of timing unpredictability. Therefore, in the situations where not all tasks meet their deadlines, the performance of EDF is comparable to that of RM.

We now examine the performance of these scheduling algorithms in terms of interval uniformity. Tables 15, 16 and 17 show the interval statistics of the simulations under RM, EDF and FIFO, respectively. We observe that the means of the differences between task periods and the nominal period are very close to zero in all simulations. The standard deviations of the periods are also relatively small.

RM and EDF perform similarly, since both allow higher-priority tasks to move ahead even when they are released later. EDF gives smaller minimums and maximums under high system load than RM. Because if a task is delayed longer, it tends to have an earlier deadline, thus being assigned a higher priority by EDF.

While the performance of FIFO is comparable to those of RM and EDF under light to medium system load (with 40 tasks or fewer), FIFO performs better than RM and EDF under high system load, in terms of means, standard deviations, minimums and maximums of task periods. This is because FIFO provides a tight upper bound in task response times, regardless task periods.

From the above simulation result, we have the following observations. First, under light and medium system load, the performances of FIFO, RM and EDF are comparable in terms of both deadline and interval constraints. Under high system load, FIFO offers better

7. In our study, all tasks are run to completion no matter if they meet their deadlines or not. This is the typical case in open-architecture machine tool controllers.

seed	# tasks	sample size	mean interval difference (μs)	standard deviation (μs)	min (μs)	max (μs)
-1	10	22628	-0.00047	68.8	-259	263
	20	42117	-0.00193	70.0	-281	300
	30	64664	-0.00045	70.3	-272	266
	40	93428	-0.00069	70.4	-278	285
	50	122872	-0.00141	71.1	-349	347
	60	150405	-0.00114	72.8	-480	502
	70	183055	-0.00015	83.2	-993	1692
-2222	10	16633	-0.00503	70.9	-267	297
	20	38252	-0.00020	70.9	-294	303
	30	72639	-0.00232	70.1	-272	278
	40	100734	-0.00184	70.6	-281	287
	50	117275	-0.00149	71.0	-379	389
	60	143509	-0.00182	72.3	-498	549
	70	169692	-0.00209	76.6	-992	1195
	80	190639	-0.00530	94.4	-992	3902
-77	10	20249	-0.00371	69.8	-261	257
	20	46856	0.00015	69.4	-276	282
	30	64322	-0.00092	69.7	-289	294
	40	83644	-0.00052	70.1	-288	305
	50	115504	-0.00091	70.6	-315	352
	60	134537	-0.00038	71.5	-495	469
	70	152831	-0.00038	73.5	-577	672
	80	184144	-0.00120	85.2	-994	2316

Table 15: Interval statistics under RM without timer resets.

performance than RM and EDF in terms of interval constraints, but it has worse performance in terms of deadline constraints. There is a trade-off here.

Second, RM and EDF perform comparably in terms of both deadline and interval constraints. However, we did not consider the run-time overhead of EDF in our simulations. From our measurements, the overhead of changing task priorities at run-time is more than 100 μs in the UMOAC testbed environment (see Chapter 4), which is prohibitively expensive for machine tool controllers. Therefore, RM is preferred because it has no such run-time scheduling overhead.

3.2.2 Experimental measurements

To verify the simulation results and to discover any factors/system behaviors that may not be accounted for in our simulations, we use the UMOAC testbed for actual measurements.

seed	# tasks	sample size	mean interval difference (μs)	standard deviation (μs)	min (μs)	max (μs)
-1	10	22628	-0.00047	68.9	-274	263
	20	42116	-0.00022	70.0	-277	260
	30	64659	-0.00183	70.4	-266	280
	40	93430	-0.00001	70.0	-279	295
	50	122856	-0.00066	70.1	-349	388
	60	150399	-0.00054	70.9	-470	524
	70	183082	-0.00044	75.2	-430	722
-2222	10	16633	-0.00503	70.7	-267	297
	20	38251	0.00007	70.5	-267	290
	30	72635	-0.00119	69.8	-272	285
	40	100732	-0.00318	69.9	-284	289
	50	117281	-0.00175	70.2	-379	389
	60	143514	-0.00173	70.7	-479	545
	70	169708	-0.00225	72.6	-435	673
	80	190664	-0.00287	76.8	-384	833
-77	10	20249	-0.00371	69.8	-261	253
	20	46851	-0.00127	69.3	-289	300
	30	64318	-0.00167	69.5	-288	272
	40	83650	-0.00027	69.6	-282	281
	50	115515	-0.00189	69.9	-310	327
	60	134520	-0.00056	70.3	-495	432
	70	152833	-0.00096	71.3	-501	595
	80	184147	-0.00071	75.5	-394	747

Table 16: Interval statistics under EDF without timer resets.

Since RM and FIFO can be more efficiently implemented than EDF, we will only consider RM and FIFO in our comparison between simulations and experimental measurements.

We first simulate the executions of a set of 5 tasks under RM and FIFO, with a interval standard deviation of 50 μs, no random starts, and an estimated execution time of 15 μs. The task periods are randomly chosen between 1 *ms* and 10 *ms*. The periods of these tasks are rounded to the nearest integral multiple of the system tick resolution (49447 *ns*) in order to obtain better timer accuracy. The performance of these tasks are then measured and compared with the simulation results.

Table 18 tabulates the deadline miss ratios from the simulations and measurements. While simulations predict that no tasks scheduled by RM will miss their deadlines, the experimental measurements show that a few invocations of task 0 missed their deadlines. This result is significant because the total CPU utilization for these 5 tasks is only about 1.82%, much less than the threshold of 74.3% given by RM (EQ 1 in

seed	# tasks	sample size	mean interval difference (μs)	standard deviation (μs)	min (μs)	max (μs)
-1	10	22628	0.00007	69.0	-292	276
	20	42120	0.00051	70.1	-282	266
	30	64656	0.00010	70.5	-269	264
	40	93435	-0.00026	70.5	-277	290
	50	122880	0.00036	70.4	-299	360
	60	150398	-0.00034	70.4	-376	414
	70	183073	-0.00016	71.1	-291	376
-2222	10	16634	0.00009	70.7	-295	299
	20	38250	-0.00219	70.6	-263	293
	30	72638	-0.00050	70.1	-285	285
	40	100730	-0.00006	70.4	-282	300
	50	117274	-0.00015	70.3	-294	305
	60	143510	0.00041	70.7	-328	401
	70	169689	0.00030	71.0	-362	352
	80	190657	0.00022	72.0	-367	491
-77	10	20248	0.00304	70.1	-283	258
	20	46856	0.00037	69.3	-270	274
	30	64322	-0.00055	69.8	-270	275
	40	83645	0.00017	70.0	-290	284
	50	115508	-0.00003	70.2	-311	298
	60	134519	-0.00055	70.3	-327	366
	70	152821	0.00054	70.8	-418	363
	80	184144	0.00003	71.8	-302	469

Table 17: Interval statistics under FIFO without timer resets.

Section 3.3). This means that actual RTOS unpredictability can be more complex and severe than the interval variation used in our simulations. Similar results are observed for the same tasks scheduled by FIFO.

task			0	1	2	3	4	total
simulation		period (μs)	2800	2277	9648	6018	6749	N/A
	RM	sample size	3443	4233	1000	1602	1429	11707
		miss ratio	0	0	0	0	0	0
	FIFO	sample size	3443	4233	1000	1602	1429	11707
		miss ratio	0	0	0	0	0	0
experiment		period (μs)	2818.5	2274.6	9642.2	6032.5	6724.8	N/A
	RM	sample size	641	751	170	259	222	2043
		miss ratio	0.624%	0	0	0	0	0.196%
	FIFO	sample size	641	751	170	260	222	2044
		miss ratio	0.624%	0.399%	0	0	0	0.342%

Table 18: Miss ratios of simulations and measurements.

Table 19 shows the statistics on periods of both simulations and measurements. The standard deviations of the measured task periods are much larger than those of the simulations. This indicates that the timer interval standard deviation used in the simulations may not be large enough. The value of the standard deviation is chosen based on our timer experiments in Chapter 2. Here, we are dealing with a different number of tasks whose periods are also different. This increased system load may cause the timer variation to increase.

task			0	1	2	3	4
simulation		period (μs)	2800	2277	9648	6018	6749
	RM	sample size	3442	4232	999	1601	1428
		mean (μs)	-0.0125	-0.0130	-0.1081	-0.0862	0.0007
		std. dev. (μs)	70.3	69.9	71.5	71.5	68.6
		min (μs)	-263.5	-215.3	-210.8	-234.6	-239.0
		max (μs)	270.0	246.3	227.0	234.5	242.0
	FIFO	sample size	3442	4232	999	1601	1428
		mean (μs)	-0.0081	-0.0165	-0.0781	-0.0956	-0.0098
		std. dev. (μs)	70.2	69.9	71.3	71.9	68.0
		min (μs)	-263.5	-235.5	-210.8	-270.5	-239.0
		max (μs)	270.0	254.3	227.0	234.5	218.5
experiment		period (μs)	2818.5	2274.6	9642.2	6032.5	6724.8
	RM	sample size	640	750	169	258	221
		mean (μs)	-0.93	-0.74	-4.44	-1.88	-2.25
		std. dev. (μs)	385.0	185.5	172.2	180.2	102.7
		min (μs)	-2765.5	-2221.7	-802.1	-1222.5	-377.3
		max (μs)	4001.6	2168.8	1028.9	1760.8	386.4
	FIFO	sample size	640	750	169	259	221
		mean (μs)	-0.93	-0.71	-2.40	-2.01	-2.24
		std. dev. (μs)	387.4	312.9	232.3	175.9	90.7
		min (μs)	-2772.8	-2227.7	-1568.9	-1179.7	-400.2
		max (μs)	4760.5	3917.9	1435.6	1960.3	318.0

Table 19: Statistics of task periods.

3.2.3 Discussion

The results of our simulations and experimental measurements offer a number of important observations regarding the performance of real-time scheduling algorithms (RM, EDF and FIFO) in the presence of RTOS timing unpredictability.

First, tasks may miss their deadlines in the presence of RTOS unpredictability even when the scheduling theory guarantees all deadlines, because the theory assumes idealized

system conditions, such as deterministic and precise task release times and negligible run-time scheduling and context switching overhead.

Second, the task deadline miss ratios are small (less than 1% from both simulations and experimental measurements). This means that RM, EDF and FIFO are still good scheduling algorithms for practical real-time systems.

Third, FIFO leads to larger miss ratios than RM and EDF under high system load. This is because FIFO gives no preferential treatment to tasks with shorter periods. When system load increases, such tasks are more likely to miss their deadlines since they may have to wait for other less urgent tasks to complete. Therefore, RM and EDF are preferred over FIFO for providing deadline guarantees.

Fourth, RM and EDF have similar performance regardless of system load. In the situations where not all tasks meet their deadlines, EDF may unnecessarily assign higher priorities to tasks that will surely miss their deadlines. Since RM is easier to implement and incurs no run-time scheduling overhead, it is the preferred choice for our open-architecture controller research and development.

Last but not least, from the comparison between simulations and measurements, we learned that simulation by itself may be inadequate in determining the performance of scheduling algorithms because it is extremely difficult to model the RTOS unpredictability accurately in the simulation. For real-time systems, it is important to use scheduling algorithms that, while not necessarily optimal, will be predictable, guarantee acceptably high levels of resource utilization, and address practical issues such as RTOS timing unpredictability. Therefore, an empirical scheduling model that uses the measurements of the actual system is needed, which we will present in the next section.

3.3 RMTU: Rate-Monotonic in the presence of Timing Unpredictability

RM is the preferred scheduling algorithm for our open-architecture controller in the presence of RTOS unpredictability, in particular timer variation. However, the original RM scheduling theory is no longer valid under such a condition and simple simulation models may not be adequate in determining the actual scheduling algorithm performance. In this section, we propose an empirical task schedulability model, called RMTU (Rate-Monotonic in the presence of Timing Unpredictability), extending the original RM scheduling theory to handle timing unpredictability. By *empirical*, we mean the model parameters are derived from the measurement of the target system.

For completeness, we first briefly review the results of the RM scheduling theory [93]. Suppose there is a set of m independent, preemptive, periodic tasks, τ_1, τ_2, ..., τ_m, with periods T_1, T_2, ..., T_m, deadlines D_1, D_2, ..., D_m, and worst-case contention-free execution times (WCETs) C_1, C_2, ..., C_m, and initial release times I_1, I_2, ..., I_m, respectively. For each task τ_i, $i=1,2,...,m$, its deadline is assumed to be equal to its period. In other words, τ_i is released at time instants I_i, I_i+T_i, I_i+2T_i, ..., and must be completed by I_i+D_i, I_i+2D_i, I_i+3D_i, ..., respectively. Without loss of generality, the task periods are assumed to be sorted in a non-decreasing order, $T_1 \le T_2 \le \ldots \le T_i \le \ldots \le T_m$.

RM assigns higher priority to tasks with shorter periods. The priorities are fixed once they are assigned to the tasks. Based on the above assumptions and ignoring all overhead and system unpredictability, Liu and Layland [93] proved that RM is optimal in the sense that no other fixed-priority assignment rule can schedule a task set which cannot be scheduled by RM. Furthermore, all tasks can meet their deadlines under such a priority assignment if the CPU utilization U satisfies the following relationship:

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i} \le m\left(2^{\frac{1}{m}} - 1\right) \qquad \text{(EQ 1)}$$

EQ 1 can also be written in the following form, which can be compared more easily with the model we will propose.

$$\forall \tau_i, 1 \le i \le m, \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} \le i\left(2^{\frac{1}{i}} - 1\right) \qquad \text{(EQ 2)}$$

While the original RM scheduling theory is simple and elegant, its results are no longer valid in the presence of RTOS unpredictability. In order to use RM and provide hard deadline guarantees, the original RM scheduling theory needs to be extended. In Section 3.3.1, we propose an empirical model, RMTU, for determining task schedulability in the presence of timing unpredictability. In Section 3.3.2, we design a set of systematic experiments to derive the model parameters. We then conduct experiments designed to validate our empirical model in Section 3.3.3. Finally, we discuss the results in Section 3.3.4.

3.3.1 An empirical task schedulability model: RMTU

Because timing unpredictability is omnipresent in RTOSs and has an adverse impact on task deadline miss ratios, we must take it into account when checking the schedulability of the tasks. In this section, we first examine the effects of these timing characteristics on the schedulability of hard real-time tasks. Based on this analysis, we will then propose an empirical task schedulability model.

We will use the simplest scenario—a single user task in the system—to illustrate the difference between the ideal and the realistic conditions. Figure 16 shows the ideal case where there is no glitch in timer firings. Task τ has a period of T and a WCET of C. It is released initially at time I and subsequently at $I+T$, $I+2T$, ... Its deadline is the end of each period. We can see that all inter-timer-firing intervals in Figure 16 are equal to the nominal period T, as assumed by the RM scheduling theory.

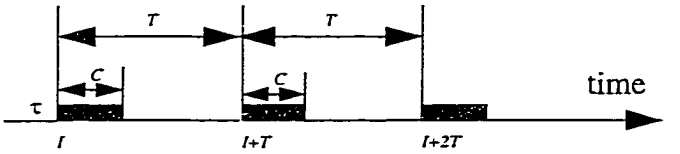


Figure 16: Task execution with an ideal software timer.

However, intervals between timer firings can vary. When a timer firing is delayed, the next firing can still be, and is typically, on time. This situation manifests itself as timer "memory" behavior, as illustrated in Figure 17. The initial timer firing is on time at the time instant I, the second is late by v time units at $I+T+v$, and the third is again on time at $I+2T$. The net effect is a longer interval of $T+v$, followed by a shorter interval of $T-v$. The parameter v is called the *timer deviation*. Note that v could change from one interval to another. If $v > T-C$, the job in that period will miss its deadline.

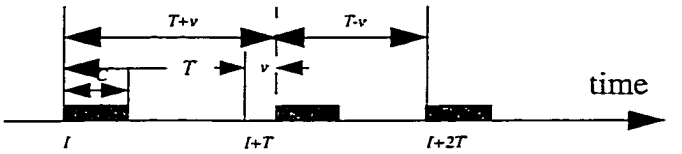


Figure 17: Task execution with timer variation.

From the above example, we can clearly see that the timer variation has an adverse effect on task schedulability, because it can leave less time for a task to complete by its deadline. We now want to quantify this effect. Since we are interested in meeting hard deadlines, we only need to examine the critical instant. A critical instant for a task is defined to be an instant at which a request for that task will have the largest response time [93]. For our single-task example in Figure 17, the critical instant is at time $I+T$, assuming that v represents the largest timer delay. Because of the delay, the task is released late at time $I+T+v$ instead of $I+T$. This leaves the task only $T-v$ time units to complete execution before its deadline of $I+2T$. For this task τ to be schedulable, assuming that the timer is the only unpredictable source in the system, the following inequality must hold:

$$\frac{C}{T}+\frac{v}{T}\leq 1 \qquad \text{(EQ 3)}$$

The effect of the timer delay v is similar to the situation where the task has a WCET of $C+v$. If the timer delay is treated as the additional execution time on top of the WCET, the following equation (actually, a set of inequalities) provides the *sufficiency* condition of task schedulability with RM for a set of m tasks:

$$\forall \tau_i,\ 1 \leq i \leq m,\ \frac{C_1+v_1}{T_1}+\ldots+\frac{C_i+v_i}{T_i}\leq i\left(2^{\frac{1}{i}}-1\right) \qquad \text{(EQ 4)}$$

where v_i denotes the worse-case timer deviation for the i-th task. The i-th inequality is a sufficiency condition for the schedulability of task τ_i. EQ 4 differs from the original RM EQ 2 only in that the C_i is replaced by C_i+v_i.

However, the timer deviation is different from the additional task execution time in that the processor can still do other useful work during the time a task is delayed by the timer. Therefore, in considering the schedulability of task τ_i, the timer deviations of higher-priority tasks should not be included in the computation. This gives rise to the following modified equation:

$$\forall \tau_i,\ 1 \leq i \leq m,\ \frac{C_1}{T_1}+\frac{C_2}{T_2}+\ldots+\frac{C_i}{T_i}+\frac{v_i}{T_i}\leq i\left(2^{\frac{1}{i}}-1\right) \qquad \text{(EQ 5)}$$

The timer deviation v_i in EQ 4 appears similar to the blocking factor B_i in [129]. Rajkumar *et al.* [129] studied the RM scheduling of periodic tasks with blocking for synchronization. They proved the following sufficiency condition for task schedulability:

$$\forall \tau_i, 1 \le i \le m, \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i\left(2^{\frac{1}{i}} - 1\right) \qquad \text{(EQ 6)}$$

where B_i is the *blocking factor* of task τ_i. This is the amount of time task τ_i can be blocked when it would otherwise be eligible to run. This blocking could result from waiting for some shared resource (e.g., a semaphore) held by a lower-priority task on the same processor or a task of any priority running on another processor (in the case of a multiprocessor system).

EQ 5 is similar to EQ 6, except that it has the timer deviation v_i instead of the blocking factor B_i. In other words, the effect of timer delay on task schedulability is similar to that of blocking, although the causes are different. The timer deviation originates from the non-ideal software timer characteristics—variation and "memory" behavior, while the blocking factor comes from the contention of shared system resources (other than the CPU, e.g., semaphores). In a multiprocessor system, the difference is more evident, where a task could be blocked multiple times within a period by a task of any priority running on another processor. This is clearly not the case for the timer delay. Therefore, the results of [129] can be readily applied to the effect of timer delay on task schedulability. That is, EQ 5 represents the sufficiency condition that all task deadlines will be met using the RM priority assignment.

Now, if we are given the WCET and the timer deviation for each task *and* assume that other characteristics of the underlying system are ideal, we can then determine the schedulability condition of the given set of tasks by applying EQ 5.

However, the underlying system is far from being ideal. Furthermore, in our target application domain, open-architecture machine tool controllers, the real-time software developers are typically mechanical engineers. They may often want to focus on issues in the areas of their expertise instead of worrying about RTOS problems. Thus, it is more

desirable to have the supporting tools to derive or estimate all necessary parameters, such as the WCETs and timer deviations.

To help the real-time application developers determine task schedulability with the RM priority assignment in the presence of timing unpredictability, we introduce an empirical task schedulability model—RMTU. RMTU has the following sufficiency condition for task schedulability:

$$\forall \tau_i,\ 1 \le i \le m,\ U_s + \frac{C_1}{T_1} + \ldots + \frac{C_i}{T_i} + \frac{v_i}{T_i} \le i\left(2^{\frac{1}{i}} - 1\right) \qquad \text{(EQ 7)}$$

where U_s is a constant representing the CPU utilization of OS activities, such as the OS kernel scheduling and interrupt handling; v_i is the worse-case timer deviation for task τ_i. EQ 7 differs from EQ 5 by the addition of U_s. Once we obtain the parameters U_s, C_i, and v_i, we can use EQ 7 to determine if all the hard deadlines can be met. This model is empirical in that the model parameters will be derived from measurement data of the target system.

Considering that the timer delay is caused by higher-priority OS activities, it should be independent of nominal timer periods. To verify this, we measure the variation of timers with different nominal periods. We used the VxWorks setup as described in Chapter 2 to run 8 periodic tasks, one at a time, and measured their intervals. The nominal periods of these tasks range from 5 *ms* to 200 *ms*. The typical range of control task periods is from 1 *ms* to 100 *ms*. Since the tick size is 1 *ms*, it is inappropriate to run tasks with a period that is very close to, or less than, 1 *ms*.

Table 20 summarizes the measurement results. The mean, minimum and maximum intervals in the table are normalized by their respective nominal task periods so that it is easier to compare the measurements of tasks with different periods. For example, for the 10 *ms* task (the second row in the table), the actual mean, minimum and maximum

intervals are 10000.38, 8391 and 11487 μs, respectively. We can see that the worst-case timer deviations are between one and two milliseconds and have no linear relationship with nominal timer periods.

Task Period (*ms*)	Sample Size	Mean interval (μs)	Standard Deviation (μs)	Min (μs)	Max (μs)
5	1213	1.26	132	-1669	1544
10	802	0.38	175	-1609	1487
20	607	-0.33	196	-1653	1385
30	539	-0.71	247	-1373	1393
50	394	0.42	254	-1060	1156
70	522	1.40	224	-1185	1264
100	417	3.54	150	-1425	1608
200	510	1.06	133	-1565	1540

Table 20: Statistics of timers with different nominal periods.

Given that the worst-case timer deviation is independent of nominal timer periods, RMTU (EQ 7) can be revised as follows:

$$\text{RMTU: } \forall \tau_i,\ 1 \le i \le m,\ U_s + \frac{C_1}{T_1} + \ldots + \frac{C_i}{T_i} + \frac{v}{T_i} \le i\left(2^{\frac{1}{i}} - 1\right) \qquad \text{(EQ 8)}$$

where the timer deviation v is a constant, regardless of task periods T_i. EQ 8 represents the sufficiency condition of task schedulability. It uses two model parameters U_s and v to capture the unpredictability of RTOS, CPU utilization of OS activities and variation in timer periods. These two parameters can be determined empirically. This model results in a tighter upper bound of CPU utilization for RM task scheduling in the presence of RTOS unpredictability.

3.3.2 Derivation of model parameters

Now that we have introduced the empirical task schedulability model RMTU, we will describe a systematic approach to deriving the model parameters.

3.3.2.1 Assumptions and derivation

To apply RMTU, we need to know all the parameters in EQ 8. T_1, T_2, ..., T_m are the task periods, which are realized using RTOS timers in applications. In an actual implementation of tasks, the periods could differ from the specified values. Since RTOS timers can deliver mean intervals very close to the nominal task period, we will use the task periods as given in our schedulability analysis.

Theoretically, the worst-case nominal task execution times C_1, C_2, ..., C_m may be obtained by analyzing the task code, given that the task execution times are bounded. In a real system, however, the same OS activities that cause timer glitches can affect task execution times as well. Therefore, the *measured* WCETs reflect more accurately the actual system performance than those obtained from code analysis and should be used in determining task schedulability. Finally, the worst-case timer deviation can only be obtained experimentally.

In essence, most model parameters need to be determined empirically. Now, we can design the experiments to derive these model parameters. Considering RMTU for the single-task case, EQ 8 becomes:

$$U_s + \frac{C_1}{T_1} + \frac{v}{T_1} \leq 1 \qquad \text{(EQ 9)}$$

We define the *available* CPU utilization (U_{avail}) as the percentage CPU cycles that can be used by user tasks. The *achievable* CPU utilization (U_{achiev}) is defined as the *maximal* CPU utilization by user tasks without missing any deadlines. It can be computed

as the summation of ratios of the WCET to period of individual tasks. U_{achiev} has the same value as the CPU utilization U in the original RM theory—the left-hand side of the less-than-or-equal-to operator in EQ 1, while the right-hand side represents U_{avail}.

EQ 1 indicates that both the available and achievable CPU utilizations are equal to 1 for all task periods in the single-task case. On the contrary, RMTU EQ 9 shows that the available CPU utilization (i.e., $1\text{-}U_s$) will be most likely less than 1. Furthermore, tasks with smaller periods will result in less achievable CPU utilizations.

There are a number of reasons for these results. First, the RM scheduling theory assumes that there are no other system activities. However, in any real system, there are OS tasks that run at priorities higher than all user tasks. Second, the RM theory ignores OS overhead, such as context switching and scheduling overhead. Again, this is not true in a real system. Third, the RM theory does not consider variations associated with the OS services, such as the POSIX timer, which have adverse impacts on task schedulability.

EQ 9 can be rewritten as EQ 10. For simplicity, the task subscript and the less-than sign are dropped.[8] In the equation, task period T is given, while the WCET C can be measured. The available CPU utilization $1\text{-}U_s$ and the timer deviation v can be derived by using linear regression:

$$C = (1 - U_s)T - v \qquad \text{(EQ 10)}$$

3.3.2.2 Systematic approach

Based on EQ 10, we design a systematic approach to determining the model parameters empirically as follows:

8. This simplification is feasible because the derived model parameters will be plugged back into EQ 8.

1. Run a single task with a fixed period and adjustable execution time (e.g., by using a loop to perform some calculations and a user-specified parameter to set the number of iterations) at the highest priority allowed by the RTOS.
2. Measure the start and completion time of each task invocation to check if it misses its deadline. The number of measured task invocation should be as large as possible and no less than a few hundreds for a given task execution time.
3. If the task misses its deadline, decrease the task execution time and repeat Step 2; otherwise, increase it. Adjust the task execution time until that the task does not miss its deadline *and* that a small[9] increase in its execution time will cause it to miss its deadline.
4. Record the worst-case execution time C and the corresponding task period T.
5. Repeat Steps 1-4 for several different task periods in the range of target application task periods.
6. After obtaining (C, T) values from Steps 1-5, use linear regression to derive the available CPU utilization $(1-U_s)$ and the timer deviation v.

Note that the measured WCET also captures the effect of RTOS activities on the task execution time. It is different from *response time* or *service time*. Response time is the elapsed time between the release of the task to its completion, while service time is the task execution time net of RTOS activities or preemptions. Figure 18 illustrates the differences.

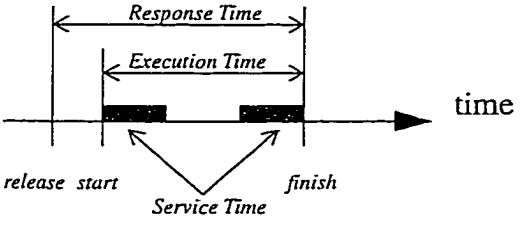


Figure 18: Execution time vs. response time vs. service time.

9. In our experiments, a small change is typically less than 1% of total task execution time.

3.3.2.3 Experimental results

We conducted the experiments described above on our computer system running VxWorks. Figure 19 shows the relationship between the *measured* achievable CPU utilization (WCET over task period) and the task period. While the RM scheduling theory (EQ 1) predicts that the achievable CPU utilization should be 1 regardless of different periods in the single-task case, Figure 19 clearly shows an upward trend in achievable CPU utilization as the task period increases (as indicated by EQ 9, which is derived from EQ 8), thus confirming the validity of RMTU (EQ 8).

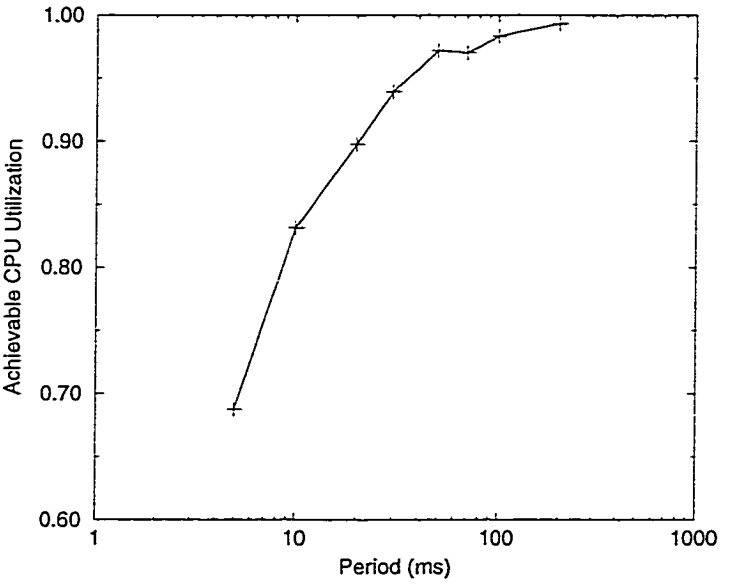


Figure 19: Single-task achievable CPU utilization versus task period.

Table 21 lists the measurement and computation results. The first 8 rows of the table shows the data for tasks whose periods range from 5 *ms* to 200 *ms*. The last 3 rows are the results of linear regression. For example, using the measured maximum execution times, the linear regression produces a timer deviation v of 1.802 *ms* and an available CPU utilization ($1-U_s$) of 1.0016 (fourth column in the table).[10] The correlation coefficient is

10. We will discuss why U_{avail} could be greater than 1 in Section 3.3.4.

very close to 1, indicating a strong positive correlation between the task period and measured WCET. We will discuss these results and their applicability in Section 3.3.3.

Task Period T (ms)	Sample Size	Achievable CPU Utilization	Maximum Execution Time C (ms)	Mean Execution Time (ms)	Minimum Execution Time (ms)
5	1214	0.688	3.438	2.771	2.695
10	803	0.831	8.314	7.726	7.650
20	608	0.898	17.954	17.672	17.588
30	540	0.939	28.184	27.700	27.583
50	395	0.972	48.597	48.002	47.962
70	523	0.970	67.920	67.395	67.259
100	418	0.983	98.347	97.694	97.651
200	511	0.993	198.606	197.698	197.586
timer deviation v (ms)			1.802	2.271	2.350
available CPU utilization $1-U_s$ (ms)			1.0016	0.9996	0.9995
regression correlation coefficient			+1.00000	+1.00000	+1.00000

Table 21: Single task experiment results.

3.3.3 Model validation

Since we have derived the timer deviation v and the available CPU utilization ($1-U_s$) experimentally, we can use RMTU (EQ 8) to determine the schedulability of any given set of independent, periodic tasks for that given hardware and software environment. RMTU can also be applied to admission control. Before a new task is admitted, its impact on the existing tasks in the system and the schedulability of the new task need to be checked using EQ 8.

To validate the empirical model, we have run several sets of three and five independent tasks in the same VxWorks setup where the model parameters were derived (Section 3.3.2). Tasks are scheduled using the RM priority assignment. In our validation experiments, all tasks in the same task set have the same amount of contention-free execution time, but differ in their periods. Only one set of tasks, either three or five tasks,

are run at a time. The execution times of the tasks in the same set are adjusted such that no tasks miss their deadlines *and* that a small increase in their execution times will cause at least one task to miss its deadline. The performance of the tasks are measured and compared with the WCETs and achievable CPU utilizations predicted by the original RM scheduling theory (EQ 1) and RMTU (EQ 8).

Tables 22 and 23 show the experimental results for the three- and five-task sets, respectively, as well as results by the RM theory and RMTU. For example, in Table 22, each of the 5 rightmost columns represents the data for a different task set. The first 3 rows are the periods of 3 tasks in the task set. The fourth row shows the composite periods of individual task sets. The composite period T_{comp} is computed using EQ 11. The remaining rows show the results of RM, RMTU and measurement.

	Set	1	2	3	4	5
period	task 1 (*ms*)	10	20	30	40	50
	task 2 (*ms*)	14	33	47	66	79
	task 3 (*ms*)	33	53	81	97	99
	composite period (*ms*)	4.957	10.084	14.935	19.817	23.387
RM	worst-case execution time (*ms*)	3.865	7.863	11.646	15.453	18.236
	achievable CPU utilization	0.780				
RMTU	timer deviation v (*ms*)	1.802				
	available CPU utilization $1-U_s$ (*ms*)	1.0016				
	worst-case execution time (*ms*)	3.603	7.536	11.338	15.116	17.848
	achievable CPU utilization	0.727	0.747	0.759	0.763	0.763
measure	sample size	400	264	243	234	218
	worst-case execution time (*ms*)	4.592	8.692	13.414	16.570	20.079
	achievable CPU utilization	0.926	0.862	0.898	0.836	0.859

Table 22: Validation with three-task experiments.

$$\frac{1}{T_{comp}} = \sum_i \frac{1}{T_i} \qquad \text{(EQ 11)}$$

	Set	1	2	3	4	5
period	task 1 (*ms*)	10	17	27	50	67
	task 2 (*ms*)	23	42	47	66	84
	task 3 (*ms*)	41	52	69	73	88
	task 4 (*ms*)	77	81	88	79	94
	task 5 (*ms*)	100	91	93	98	100
	composite period (*ms*)	5.240	7.987	10.535	13.945	16.998
RM	worst-case execution time (*ms*)	3.896	5.939	7.833	10.368	12.638
	achievable CPU utilization	0.743				
RMTU	timer deviation v (*ms*)	1.802				
	available CPU utilization 1-U_s (*ms*)	1.0016				
	worst-case execution time (*ms*)	3.810	5.793	7.645	10.134	12.358
	achievable CPU utilization	0.727	0.725	0.726	0.727	0.727
measure	sample size	103	142	145	183	148
	worst-case execution time (*ms*)	5.119	7.549	9.396	12.217	14.160
	achievable CPU utilization	0.977	0.945	0.892	0.876	0.833

Table 23: Validation with five-task experiments.

The periods of these tasks are chosen between 10 *ms* and 100 *ms*. To better verify RMTU, they are selected such that the composite periods of the task sets are distributed more evenly within the range.

The first set in Table 23 has five tasks, whose periods are 10, 23, 41, 77 and 100 *ms*, respectively. According to the original RM theory (EQ 1), the task set is schedulable if their achievable CPU utilization is no greater than 0.743. This corresponds to a worst-case execution time of 3.896 *ms*. Note that the execution times are the same for all five tasks. The timer deviation and available CPU utilization are 1.802 *ms* and 1.0016,[11] respectively, which were derived from the experiments in Section 3.3.2. Using

11. See next section for a discussion why the derived available CPU utilization may be greater than 1.

these two model parameters, RMTU gives an achievable CPU utilization of 0.727 and the corresponding WCET of 3.810 *ms*.

The last three rows of the table show the actual measurement data of the tasks. The sample size is the number of invocations for the task with the longest period. For tasks with shorter periods, there are more data points, proportional to the ratio of their task periods. For the first task set, there are 103 data points for the 100 *ms* task. For the 10 *ms* task, there would be roughly 10 times more data points. In the actual measurement, there are 1024, 445, 250, 133 and 103 data points for the five tasks in the first set of Table 23, respectively. The measured WCET for this task set is 5.119 *ms* (which corresponds to an achievable CPU utilization of 0.977), exceeding the 3.810 *ms* schedulability threshold set by RMTU. A tiny increase in the task execution time caused the task with the longest period (hence the lowest priority) to miss its deadline.

In these tables, no tasks missed their deadlines when the CPU utilization is less than, or equal to, the schedulability threshold given by RMTU, showing the validity of the model as a sufficiency condition for task schedulability with the RM priority assignment in the presence of timing unpredictability.

3.3.4 Discussion

There are a number of important observations to be made from our experimental measurement in Sections 3.3.2 and 3.3.3. First, RMTU gives a tighter upper bound of the achievable CPU utilization than the original RM scheduling theory. For example, the first set of tasks in Tables 22 have periods of 10, 14 and 33 *ms*, respectively. All tasks have the same execution time. While the RM theory indicates a CPU utilization upper bound of 0.780 and the WCET of 3.865 *ms*, our empirical model gives 0.727 and 3.603 *ms*, respectively. In other words, the deadlines guaranteed by the RM theory may actually be missed in our experimental environment, because the RM theory does not take the timing unpredictability into account.

Mathematically, RMTU gives a tighter upper bound of the achievable CPU utilization if and only if the following holds, by comparing the original RM theory (EQ 1) and RMTU (EQ 8):

$$U_s + \frac{v}{T_m} > 0 \qquad \text{(EQ 12)}$$

If we use the more conservative version of RMTU (EQ 14), EQ 12 will always hold. Even if EQ 8 is used instead of EQ 14, EQ 12 should also hold because of its physical implication—the overall effect of having RTOS activities and timer interval variation is the reduced CPU cycles for tasks, thus more difficult to schedule the task set.

Second, the tasks can typically reach better achievable CPU utilization than predicted by either RM or RMTU. For the above set of tasks, the measured WCET is 4.592 *ms* and hence the achievable CPU utilization is 0.926. However, this is not equivalent to an actual CPU utilization of 0.926, because most task executions take less time than the worst case. Therefore, the measured achievable CPU utilization should only be used as a sufficiency condition for checking the task schedulability.

Third, the available CPU utilization ($1\text{-}U_s$), one of our empirical model parameters, could be greater than one. At the first glance, it may be strange that the available CPU utilization can be greater than one. However, the available CPU utilization is again not the actual CPU utilization. When the WCETs are used in EQ 10 to derive model parameters, the available CPU utilization can be over-estimated (thus resulting in a value larger than 1). This is because only a very small number of task invocations actually take that much time to complete. If we want to be more conservative in determining the schedulability of tasks, we can re-define the available CPU utilization as:

$$U'_{avail} = \min(1, 1 - U_s) \qquad \text{(EQ 13)}$$

where (1-U_s) is the derived value from the experiments. RMTU EQ 8 can then be rewritten as:

$$\forall \tau_i, 1 \le i \le m, (1 - U'_{avail}) + \frac{C_1}{T_1} + \ldots + \frac{C_i}{T_i} + \frac{v}{T_i} \le i\left(2^{\frac{1}{i}} - 1\right) \qquad \text{(EQ 14)}$$

This revised equation represents a more stringent sufficiency condition for task schedulability. It could result in smaller achievable CPU utilizations.

The measured mean or minimum execution times can also be used in our empirical model. As shown in Table 21, this results in larger timer deviations and smaller available CPU utilizations. However, the measured WCETs are more appropriate for the following reasons. First, since we are interested in meeting hard real-time deadlines, a more conservative approach is in order. Second, the measured WCETs reflect the real computing environment better than either mean or minimum execution times. Note that the measured minimum task execution times correspond to the *contention-free* WCETs that would have been derived by code analysis. Because of higher-priority OS activities, the actual (measured) WCETs can be much longer than the theoretical (contention-free) ones. For example, the measured WCET of Task 1 in the first set of Table 21 is 4.592 *ms*, while its measured minimum execution time is only 3.885 *ms*. Furthermore, the underlying OS is very complex and cannot be fully modeled by the timer characteristics we described earlier. The precise schedulability analysis of a set of independent, periodic tasks in a real system is extremely difficult, if not impossible, and may not be available in the near future. By using measured WCETs, we have the additional benefit of factoring into our schedulability analysis system disturbances other than the timer variation.

3.4 Related work

Since the introduction of the RM priority assignment [93], there has been significant interest in applying and extending the theory. A periodic task set $\{\tau_1, \tau_2, ..., \tau_m\}$ with $D_i = T_i$, $1 \le i \le m$, is schedulable by RM if EQ 1 holds. This result has been extended for the cases with $D_i \le T_i$ [68, 77, 78, 88], with deadlines being integral multiple of periods [78, 89], and with $D_i > T_i$ [138].

Rajkumar *et al.* [129] and Lortz and Shin [98] studied scheduling issues for periodic tasks with blocking due to non-preemptive critical task sections or mutually exclusive access to shared resources. In the uniprocessor case, the sufficiency condition for schedulability under RM is as follows:

$$\forall \tau_i, 1 \le i \le m, \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i\left(2^{\frac{1}{i}} - 1\right) \qquad \text{(EQ 15)}$$

where B_i is the blocking factor of task τ_i.

All of the above RM extensions assume ideal timers, except that the task deadlines do not necessarily coincide with their respective periods. But the ideal-timer assumption no longer holds in real systems.

EQ 15 is the same as a primitive form of our empirical model EQ 5, except that it has the blocking factor B_i instead of the timer deviation v_i in RMTU. While we can use the results from [129], our model is addressing a phenomenon that has causes different from blocking. Furthermore, RMTU does not assume ideal RTOS timers while EQ 15 does. RMTU can be considered as an extension of EQ 15 in the sense that the blocking factor can be lumped together with the timer deviation. We also developed a set of systematic experiments to derive the model parameters empirically.

While most existing research related to the scheduling of periodic tasks deals with simplified or idealized system conditions, there is also work investigating the application of the theory to task sets implemented via real-time operating systems.

Kettler, Katcher and Stronsnider [72] proposed an engineering methodology that allows users to accurately model and evaluate RTOSs, thus providing a framework to account for implementation costs in real-time scheduling theory. Their approach requires a high degree of expertise to create a valid model for any given RTOS. Such expertise may be scarce among users. Furthermore, as the RTOSs become more sophisticated with added functionality (such as network and multimedia device modules), the modeling work becomes more difficult. Instead, we designed a set of systematic experiments to characterize the underlying RTOS and we provide a simple sufficiency condition for applying the RM scheduling theory.

Audsley *et al.* [11] proposed a non-preemptive fixed-priority scheduling approach for safety critical hard real-time systems. They describe the advantages of the fixed priority scheduling over the traditional cyclic executive scheduling approach and outline an approach for gathering the necessary evidence for presentation to certification authorities. They also point out some disadvantages in the fixed-priority scheduling. In particular, the *release jitter* of tasks may be worse than that using cyclic scheduling due to the dynamic nature of the run-time schedule. Release jitter refers to the same phenomenon as the timer variation. Their solution is to use a non-preemptive approach trying to avoid/ minimize the problem. We instead address the problem directly by modeling and measuring the effects of the jitter.

Jeffay and Stone [66] studied the feasibility and schedulability problems for periodic tasks that must compete for the processor with interrupt handlers, which are assumed to always have priority over application tasks. They developed conditions that solve the feasibility and schedulability problems, and demonstrated that their solutions are computationally feasible. However, the assumptions in their formal analysis remain

overly-simplistic. For example, they assume that interrupt handlers are strictly periodic, which is rarely the case in practice. By contrast, our work is more practical and can be a viable methodology in real-time application development.

3.5 Summary

A real-time open-architecture machine tool controller typically consists of a set of periodic tasks. These tasks need to be scheduled such that their deadlines can be met. However, it is not clear how theoretically-proven optimal scheduling algorithms (e.g., RM and EDF) will perform in the presence of RTOS unpredictability. Therefore, we first studied the performance of RM, EDF and FIFO using a combination of simulations and experimental measurements.

Based on our simulations, we find that theoretically-proven algorithms often will not work as expected in an actual real-time computing environment, simply because there are important factors ignored in the development of the algorithms. We showed that, among RM, EDF and FIFO, FIFO has worse deadline miss ratios than RM and EDF under high system load, while RM and EDF have similar performance. Therefore, we favor RM over EDF and FIFO, because of its implementation simplicity and no run-time scheduling overhead.

In addition, our simulations and experimental measurements showed that the original RM scheduling theory results are no longer valid in the presence of timing unpredictability. In order to handle such timing unpredictability, we proposed an empirical model, called RMTU, extending the RM scheduling theory. We further designed an approach that allows the application developer to systematically run and measure a set of simple tasks. The model parameters can then be derived from the measurement data. Our experiments verified the validity of RMTU. The results of this study should be useful not only to control application developers, but also to real-time practitioners at large.

CHAPTER 4

PROBABILISTIC DEADLINE GUARANTEES

4.1 Introduction

A real-time task is usually characterized by its arrival/release time, deadline, and worst-case nominal execution time. Depending on the severity of the consequence of missing its deadline, the task is classified as *hard*, *firm* or *soft* [142]. With this classification, only hard real-time tasks require absolute deadline guarantees. Firm and soft real-time tasks require no guarantees and receive only best-effort services, because the semantics of "firm" or "soft" deadline guarantees are not defined clearly. This classification is often inadequate for specifying requirements and characterizing the performance of many real-time applications. For example, tasks in machine tool controllers can tolerate deadline misses, but only to a certain degree [112]. This indicates that these controller tasks are not hard real-time tasks and hence, do not require absolute guarantees. On the other hand, best-effort approaches are insufficient for them because there is no guarantee whatsoever.

One simple-minded solution to this problem is to treat all tasks as hard real-time tasks. However, such a solution is not economical for those applications that can tolerate some deadline misses. Moreover, treating all tasks as hard will severely limit the number of tasks the system can accommodate. For example, suppose a hard real-time task τ_1 runs once every 10 *ms* for a duration of 5 *ms* each time and another task τ_2 runs every 15 *ms* for 6 *ms* each time but can tolerate deadline misses up to 50% of the time. Their deadlines are the same as their respective periods. If both tasks are released at time 0 initially, the

deadlines of τ_1 will be at time instants 10 *ms*, 20 *ms*, 30 *ms*, and so on. Similarly, the deadlines of τ_2 will be 15 *ms*, 30 *ms*, 45 *ms*, and so on. Clearly, these two tasks cannot be guaranteed using the RM scheduling algorithm [93] if both of their deadlines are treated as hard. On the other hand, their original timing requirements can be met by using the same RM scheduling algorithm if the tolerance of deadline misses is taken into account.

To address this problem, we propose a practical framework for probabilistic deadline guarantees. This framework includes several key components. We first define a *Probabilistic Real-Time Constraint Model* (PRTCM), with which the tolerance of application task deadline misses can be quantified in terms of *completion probability.*

While in theory the above two tasks could get hard deadline guarantees using the earliest-deadline-first (EDF) or the first-in-first-out (FIFO) scheduling algorithm, system unpredictability can make the guarantees difficult or impossible. Even if the system can provide hard deadline guarantees for all tasks in some cases, it would in general be severely under-utilized. Comprehensive assessment of scheduling algorithms is therefore necessary to determine which algorithm works well for making probabilistic deadline guarantees.

We propose a number of *completion-probability-cognizant* and *CPU-utilization-cognizant* heuristics, and evaluate their performance as well as popular scheduling algorithms -- RM, EDF and FIFO -- in the context of probabilistic deadline guarantees. An especially interesting situation occurs when the system is temporarily overloaded, as a result of system unpredictability (e.g., timer jitters [185]), hardware failure, or optimistic scheduling. Our simulation results show that most scheduling algorithms work well if the system is not overloaded. However, when the system is temporarily overloaded, RM generally performs best in terms of useful job ratio under all three system load patterns we simulate. UM (*Utilization-Monotonic*) performs best in terms of task completion probability miss ratio under variable CPU utilizations and fixed task execution times, but it is not effective under fixed task CPU utilizations. As an alternative, UM_CP

(*Utilization-Monotonic with Completion Probability*) generally performs well in terms of task completion probability miss ratio under all three system load patterns. We also show that the introduction of completion probability can improve CPU utilization as well as *job* and *task guarantee ratios*, by comparing the performance of these scheduling algorithms between probabilistic and deterministic deadline guarantees.

Another key component of our practical framework for probabilistic deadline guarantees is a measurement-based simulation technique to obtain task completion time distributions, which will be described in next chapter.

The remainder of this chapter is organized as follows. Section 4.2 describes the probabilistic model, PRTCM. Section 4.3 defines criteria for evaluating the performance of a real-time scheduling algorithm. Section 4.4 introduces new heuristics for probabilistic deadline guarantees. Sections 4.5 and 4.6 describe simulation parameters and results, respectively. Section 4.7 compares our results of evaluating scheduling algorithms for both probabilistic and deterministic deadline guarantees. Section 4.8 presents related work, while Section 4.9 summarizes the chapter.

4.2 Probabilistic Real-Time Constraint Model (PRTCM)

The hard, firm or soft deadline of a real-time task is classified based on the *residual value* imparted to the application when the task is completed after its deadline, as illustrated in Figure 20. As soon as a hard real-time task misses its deadline, its residual value becomes negative infinity. A task with a firm deadline has zero residual value if it is completed after its deadline. A soft real-time task may still provide a positive residual value after it misses its deadline. This value is typically a non-increasing function of time that goes to zero eventually. A non-real-time task has a constant residual value and no deadline.

However, such a categorization is often inadequate. For example, the sensor-reading task in a machine tool controller may be allowed to miss its deadline occasionally

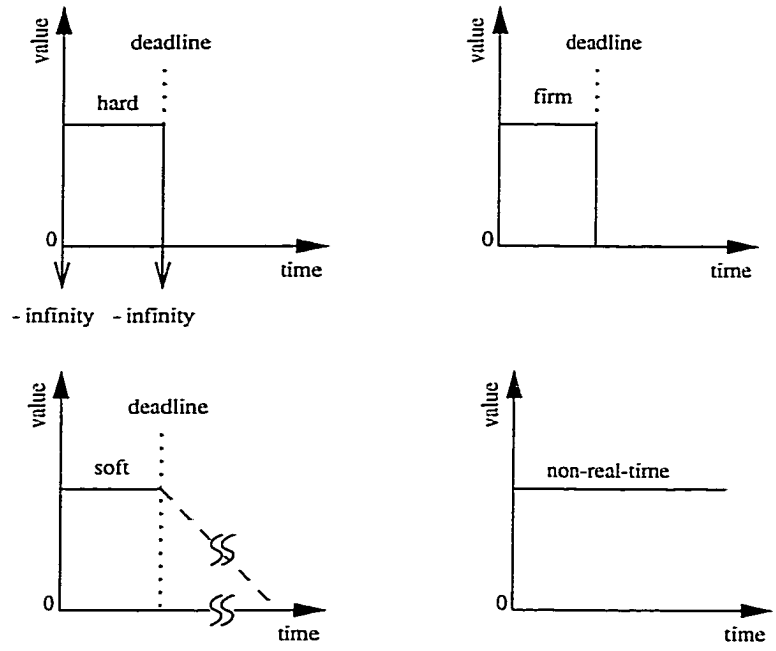


Figure 20: Residual-value-based categorization of real-time task deadlines.

but it must be completed 90% of the time by its deadline. Another example is the transmission of video data over a computer network. To maintain the picture quality, it is often required that at most one frame out of any K consecutive frames is allowed to miss its deadline. Using the residual-value-based categorization, these two applications would fall into the soft real-time category since they can tolerate some deadline misses. But it is clear that the simple notion of "soft" cannot fully convey the requirements of these two applications.

The problem with this categorization is that it mixes two distinct concepts: residual value and criticality. *Criticality* represents the application's tolerance of deadline misses (i.e., whether the application allows any deadline misses or to what degree it allows deadline misses). The residual value quantifies the utility of the task if it is completed after its deadline. While the two concepts have a strong correlation, one cannot always be substituted for the other. For example, if the residual value of a real-time task is $-\infty$, it must be a hard real-time task, disallowing any deadline miss. On the other hard, if a task can tolerate no deadline misses, its residual value could be $-\infty$ or 0, depending on whether the

deadline is hard or firm. Furthermore, the hard, firm and soft classification of criticality is too coarse to be useful for many applications.

We therefore separate criticality from residual value by introducing the concept of completion probability. The completion probability of a real-time task, denoted by P_c, is defined as the *required* probability with which the task must be completed by its deadline.[1] By "required," we emphasize that completion probabilities be a part of application requirements, as opposed to the system's capability to satisfy such requirements. The completion probability captures the criticality of a real-time task by quantitatively specifying its tolerance of deadline misses. A *probabilistic* real-time task is defined as a real-time task whose deadline is associated with a completion probability.

In the machine tool controller example we mentioned earlier, the sensor-reading task must be completed by its deadline 90% of the time, which corresponds to a constant completion probability of 0.90. For the video data transmission application that can tolerate at most one miss out of any K consecutive deadlines, the completion probability is a function of the history of transmitting the last K-1 frames. The completion probability for frame i is:

$$P_c(i) = \begin{cases} 1, \text{ if frame } j \text{ missed its deadline where } j \in \{i-K+1, i-K+2, \ldots, i-1\} \\ 0, \text{ otherwise} \end{cases}$$

Conceptually, the completion probability is 1.0 for tasks with hard deadlines, equal to or less than 1.0 for firm deadlines, and strictly less than 1.0 for soft deadlines. While both hard and firm deadlines can have the same completion probability of 1.0, their residual values are different: $-\infty$ for the former and 0 for the latter. Using both concepts of

1. We proposed a preliminary version of the concept of completion probability in [184]. Although the concept of completion probability is defined for real-time deadlines, it can be generalized to any real-time constraints, such as temporal relationships between data items. We call this generalized concept *constraint probability*, defined over a real-time constraint as the *required* probability with which the constraint must be met.

completion probability and residual value will allow us to better specify the requirements of real-time applications. Figure 21 shows where the traditional hard, firm and soft deadlines map to in the new categorization space.

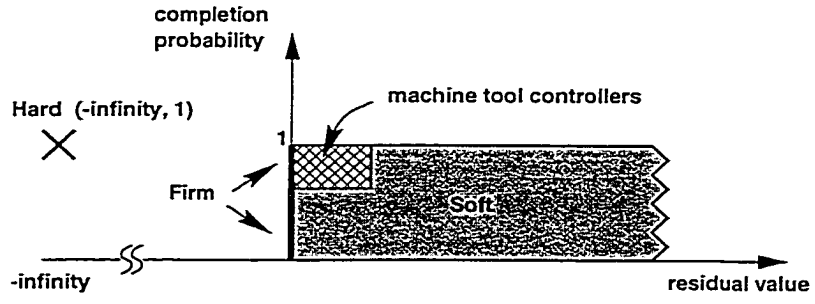


Figure 21: Completion-probability- and residual-value-based categorization.

Real-time tasks with hard deadlines correspond to a single point in Figure 21, namely, the one that has a completion probability of 1.0 and a residual value of $-\infty$. Tasks with firm deadlines have a residual value of 0 and could have any value of completion probability, corresponding to the line segment from (0,0) to (0,1) inclusive in the figure. Tasks with soft deadlines map to a region whose completion probability ranges from 0 to, but not including, 1.0, and whose residual value is non-negative, i.e., the shaded area. Most periodic tasks in open-architecture machine tool controllers fall into this region, as illustrated by the meshed area.

There could be another category of tasks corresponding to the region left of the vertical axis, whose completion probability ranges from 0 to 1 and whose residual value is negative. For example, the policy of some pizza delivery places states that the customer pays nothing if the pizzas are not delivered within a certain time limit after the order is received. Clearly, the residual values of such delivery tasks are negative. Because if the task is completed after the deadline, the company receives nothing from the customer instead of getting paid for its food and service. The completion probability of such tasks should be as close to 1 as possible to reduce potential losses.

Traditionally, if the deadline of a hard real-time task can always be met by the system, the task is said to have a hard-deadline guarantee provided by the system. There are usually no deadline guarantees given to soft or firm real-time tasks. Given the new notion of completion probability, we now introduce the accompanying concept of *probabilistic* deadline guarantee. A task has a probabilistic deadline guarantee if its completion probability is guaranteed by the system. In this context, a real-time scheduling algorithm is said to be *optimal* if no other scheduling algorithm can provide probabilistic deadline guarantees for a task set if this algorithm cannot.

In short, PRTCM introduces the concept of completion probability and adopts a categorization for real-time tasks based on completion probability and residual value. It further defines the semantics of probabilistic deadline guarantee and enables a new dimension for schedulability analysis. We want to stress that our PRTCM is based on application *requirements*. It establishes the foundation for the evaluation of the system's capability of providing probabilistic deadline guarantees. While there is a body of work on real-time scheduling using residual values [26, 67], we only need to consider the completion probability in providing probabilistic deadline guarantees.

4.3 Performance Evaluation Criteria

An important part of the design of real-time systems for probabilistic deadline guarantees is to select an appropriate scheduling algorithm. Unlike deterministic guarantees, evaluating scheduling algorithms is more difficult in the case of probabilistic guarantees. Part of this difficulty comes from the existence of abundant evaluation criteria. The relative performance of scheduling algorithms may change, depending on the criteria used for the evaluation.

There can be two types of evaluation criteria, task-oriented and job-oriented. In a real-time system, jobs are the units of work that are actually processed by the CPU, while

tasks are typically used to characterize logical groups of related jobs, e.g., regularly recurring jobs can be specified as a periodic task.

Both task-oriented and job-oriented objective functions can be useful for applications. For example, Internet Service Providers (ISPs) typically have retail customers with dramatically different usage patterns. Some customers may go on-line once a week and for a short time, while some may connect to the net many times per day and for a long duration. The usage of others may be in between. ISPs often try to keep as many customers happy as possible, as opposed to giving higher priorities to heavy users, especially when they are paying a flat fee. Each customer's use of the service can be considered as a job, while all uses by each customer can be considered a task. In this example, the objective function is to satisfy as many tasks (i.e., customers) as possible.

On the other hand, a job-oriented objective function may be more appropriate if the service pricing is mainly based on individual jobs, such as in express package delivery services. Every package delivery can be considered an equally important job, and the goal is to have as many of them delivered in time as possible.

For our evaluation of scheduling algorithms, we define two objective functions. One is task-oriented and the other is job-oriented. Both are designed to measure how well the completion probability requirements of PRTCM are met.

4.3.1 Task-oriented objective function

In the context of probabilistic guarantees, the most important task-oriented criterion in selecting a scheduling algorithm is the number of tasks whose completion probabilities are satisfied. We define the *task completion probability miss ratio* as the ratio of the number of tasks whose completion probability requirements are *not* met, to the total number of tasks. This metric provides a direct measurement of how well the completion probability requirements are met at the task level.

For those scheduling algorithms with the same task completion probability miss ratio, the secondary selection criterion can be the *task deadline miss ratio*, which measures how well the deadline requirements (regardless of completion probability requirements) are met at the task level. For example, we have two tasks whose completion probability requirements are both met. One task has a few jobs missing their deadlines while the other has none. Therefore, the task completion probability miss ratio is zero and the task deadline miss ratio is 50%.

We also distinguish the *task* deadline miss ratio from the *job* deadline miss ratio. The former is defined as the ratio of the number of tasks whose deadline requirements are not met, to the total number of tasks. The latter is defined as the ratio of the number of jobs missing their deadlines to the total number of jobs. For the same job deadline miss ratio, there can be quite different task deadline miss ratios. For example, if all jobs missing their deadlines belong to a single task, the corresponding task deadline miss ratio will be very small. However, if every task has some jobs missing deadlines, the task deadline miss ratio will be 100%. Similarly, there can be different job deadline miss ratios for the same task deadline miss ratio.

4.3.2 Job-oriented objective function

As a job-oriented objective function in the context of probabilistic guarantees, we define the *useful job ratio* as $\left(\sum_i J_i\right) / (\text{total number of jobs})$, where J_i is the number of jobs of task τ_i that meet their deadlines if the completion probability of the task is satisfied. J_i is equal to zero if the completion probability of τ_i is not satisfied. For example, tasks τ_1 and τ_2 have the same period and same completion probability of 0.75. Suppose 90% and 50% jobs of τ_1 and τ_2 meet their deadlines, respectively. The useful job ratio would then be 45% since no jobs of τ_2 count. The assumption behind this definition is that if the completion probability of a task is not satisfied, none of its jobs are useful even though some of them may meet their deadlines. The useful job ratio provides a direct

measurement of how well the completion probability requirements are met at the job level. Ideally, it should be 100%.

4.3.3 Temporary overload

Schedulability analysis is usually based on average CPU utilization, or the percentage of CPU time used by jobs over an infinite time interval. For a finite time interval, the CPU utilization may be higher or lower than the average CPU utilization. For example, in the RM analysis [93], the average CPU utilization over an infinite time interval is the same as that over the time interval which is the least common multiple (LCM) of individual task periods. But within the LCM time interval, CPU utilization can be higher or lower than the average. If we consider the LCM as the granule, CPU utilizations will be the same over any time interval for a given set of periodic tasks.

We define the CPU *demand* as the percentage of the total execution time of jobs released within a specified time interval. For a periodic task, its CPU demand over an infinite time interval is equal to the ratio of its execution time to its period. While CPU utilization cannot exceed 1.0, CPU demand has no such limit. Of course, a CPU demand of over 1.0 cannot be satisfied by a single processor.

For any given time interval, the load of a real-time system may range from light to medium, to heavy, and to overloaded. The meaningful granule of such time intervals is typically the LCM of the task set. Using CPU demand, we can provide a more quantitative definition of system load ranges as follows:

- Light: CPU demand is less than 0.4.
- Medium: CPU demand is between 0.4 and 0.7.
- Heavy: CPU demand is between 0.7 and 1.0.
- Overloaded: CPU demand is more than 1.0.

An interesting phenomenon is *temporary overload* where the CPU demand exceeds 1.0 for some time intervals. This can happen even when the granule is the LCM. Possible causes of temporary overload are:

- There is a processor failure in the system. Some tasks that were allocated to the failed processor may be transferred to this processor.
- Optimistic scheduling policy may be used. If not all tasks require hard guarantees, more tasks may be admitted than otherwise permitted. In certain time intervals, the CPU may be temporarily overloaded.
- There are timer jitters. The intervals between timer firings vary due to the contention with other higher-priority RTOS activities. When a timer firing is delayed, the effective CPU demand of the job for that period increases, since there is less time available for executing the job in the period before its deadline. This can cause a temporary overload. For example, let T and C be the period and execution time of a task, respectively. Suppose a job of the task should be released at time I and its deadline is $I+T$. Its CPU demand for the period from I to $I+T$ is C/T. However, if the job is released late at time $I+v$, it cannot make use of the time interval from I to $I+v$, even if the CPU is idle. Effectively, the CPU utilization increases to $C/(T-v)$.

While the CPU could be overloaded at times, it is not overloaded on average in any meaningful real-time system. The temporary overload condition is interesting, especially in the context of probabilistic guarantees, where some deadline misses can be tolerated. The system may be designed to provide satisfactory performance under average load, and acceptable performance under temporary overload by deliberately adopting optimistic task/job admission policies. Consequently, in our evaluation of scheduling algorithms, we will identify those that work well under the average condition and are also robust under temporarily-overloaded conditions.

4.4 Heuristics for Probabilistic Guarantees

The information about completion probability and CPU utilization are important for making probabilistic guarantees, especially when the system can be temporarily overloaded. We propose two types of scheduling algorithms for probabilistic guarantees: one cognizant of completion probability and the other cognizant of CPU utilization.

4.4.1 Completion-probability-cognizant heuristics

We first consider a spectrum of fixed-priority heuristics that put progressively more weight on completion probability when determining task priorities, as illustrated in Figure 22.

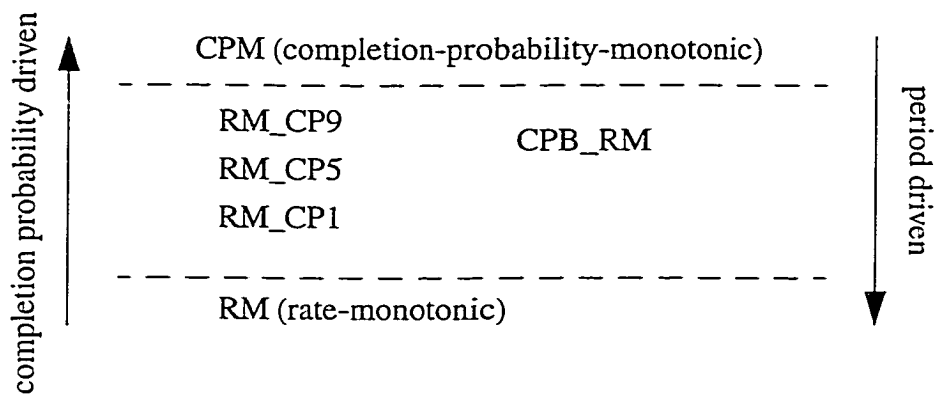


Figure 22: Spectrum of fixed-priority scheduling algorithms.

The first heuristic is the Completion-Probability-Monotonic (CPM) scheduling algorithm, which assigns higher priorities to tasks with larger completion probabilities, regardless of their periods. It represents one extreme of the spectrum of scheduling algorithms. At the other extreme of the spectrum is the RM scheduling algorithm that considers task periods without regard to completion probabilities. Since CPM may generate a task priority assignment different from RM, which is optimal in terms of job deadline miss ratio under idealized system conditions, CPM is likely to result in a job deadline miss ratio higher than RM. But the idea behind CPM is to shift the job deadline

misses to the tasks with smaller completion probabilities. RM and CPM are equivalent if the task with a shorter period always has a larger completion probability.

We introduce several other heuristics that use the information about both period and completion probability in assigning task priorities. The first class of heuristics is called RM_CP*n* (Rate Monotonic with Completion Probability) that assigns task priorities based on the following formula:

$$\text{priority} \propto \frac{CP^n}{T}$$

where CP and T are the completion probability and period of the task, respectively. RM_CP0 is equivalent to RM. Assuming that both completion probabilities and periods cannot be zero and the periods are bounded, it can also be shown that RM_CP*n* is equivalent to CPM when n is large enough.[2] RM_CP*n* modifies RM such that a task with a longer period but larger completion probability may possibly be assigned a priority higher than a task with a shorter period but smaller completion probability. As n increases, RM_CP*n* puts more weight on completion probability than period. We will evaluate RM_CP1, RM_CP5 and RM_CP9.

Another class of heuristics we propose is called CPB (Completion Probability Bucket). With CPB, tasks are first sorted into m buckets of completion probability. These m buckets divide the entire range (0, 1] of completion probability into m equal sub-ranges.[3] Tasks in a bucket with larger completion probabilities have higher priorities than those in another bucket with smaller completion probabilities. We will evaluate one

---

2. We need to prove that $\text{priority}_1 > \text{priority}_2$ if $CP_1 > CP_2$ and $n \geq N$, where N is a constant natural number. Let $\alpha = min(CP_i/CP_j)$ where $CP_i > CP_j$, and $\beta = max(T_i/T_j)$ where $T_i > T_j$. It is now easy to choose an N such that $\alpha^N > \beta$ and show that the following holds:

$$\left(\frac{CP_1}{CP_2}\right)^n \geq \left(\frac{CP_1}{CP_2}\right)^N \geq \alpha^N > \beta \geq \frac{T_1}{T_2} \Rightarrow \frac{{CP_1}^n}{T_1} > \frac{{CP_2}^n}{T_2} \Rightarrow \text{priority}_1 > \text{priority}_2$$

3. The equal sub-ranges are not a necessity. It is perfectly fine to divide the range differently if that makes sense for the given applications.

particular heuristic from this class—CPB_RM (Completion Probability Bucket with Rate Monotonic). Suppose m=10, then tasks whose completion probabilities are in (0.9, 1.0] are in one completion probability bucket, while (0.8, 0.9] is another bucket. Within each bucket, RM is used. CPB_RM modifies CPM such that tasks with similar completion probabilities are scheduled with RM, instead of strictly using their completion probabilities.

CPB_RM may result in a priority assignment different from RM_CPn, say, RM_CP2. Suppose there are three periodic tasks whose periods are 1 *ms*, 2 *ms* and 4 *ms* and whose completion probabilities are 0.5, 0.9 and 0.95, respectively. CPB_RM will have Priority 2 > Priority 3 > Priority 1, while RM_CP2 will have Priority 2 > Priority 1 > Priority 3.

Scheduling algorithms that assign higher priorities to tasks with shorter periods (e.g., RM) tend to favor job-oriented objective functions.

4.4.2 CPU-utilization-cognizant heuristics

We introduce two CPU-utilization-cognizant heuristics, both of which tend to favor task-oriented objective functions for performance evaluation. The first heuristic is the Utilization-Monotonic (UM) scheduling algorithm. It assigns higher priorities to tasks with lower CPU utilizations, regardless of other parameters (such as task periods or completion probabilities). The task CPU utilization is computed as the ratio of the nominal execution time of the task to its period.

$$\text{priority} \propto \frac{1}{C/T}$$

The idea behind UM is that tasks with lower CPU utilizations impose less competition for CPU cycles than other tasks. If we try to satisfy the requirements of tasks with lower CPU utilizations first, there is a better chance that the total number of tasks whose completion probability requirements can be met is larger.

Our second CPU-utilization-cognizant heuristic tries to improve on the UM heuristic by taking completion probability into account. It is called Utilization-Monotonic with Completion Probability (UM_CP), where tasks with smaller CPU utilization to completion probability ratios have higher priorities:

$$\text{priority} \propto \frac{CP}{C/T}$$

where C, T and CP are the nominal execution time, period and completion probability of the task, respectively. The idea is that the information about a task's completion probability could be as important as its CPU utilization.

4.5 Simulation Parameters

We use a task model similar to that described in [93]. All tasks are periodic with constant periods and worst-case execution times. Their deadlines are the end of periods. Table 24 lists the simulation parameters.

The seed for the random number generator is randomly chosen. A different seed represents a different sequence of random numbers of the same distribution. All tasks are initially released at the same time, which represents a critical instant. Task completion probabilities are generated between the specified minimum and maximum values with an equal probability.

Tasks are fully preemptive. The context switching overhead is assumed negligible. The run-time scheduling overhead of EDF is also ignored, but this makes no difference in our conclusions, because EDF does not perform better than the others, even when its run-time scheduling overhead (higher than others) is not considered.

Though the actual number of tasks in the system and load may change over time, we only need to use fixed number of tasks and load for each simulation and apply the results to appropriate time intervals under consideration. For example, a system may have

Simulation Parameter	Notation	Value		
seed for random number generator	s			-1
random start	r			no
number of tasks	n			various
minimum task execution time	C_{min}	0.5%	50 µs	1%
maximum task execution time	C_{max}	5%	50 µs	1%
execution time standard deviation	σ_C			0
minimum task period	T_{min}			1000 µs
maximum task period	T_{max}			100,000 µs
task period distribution	f			uniform and bimodal
period standard deviation	σ_P			0
task deadline	D			end of each period
minimum completion probability	CP_{min}			0.2
maximum completion probability	CP_{max}			1.0

Table 24: List of simulation parameters.

10 specific tasks from t_1 to t_2 and 20 different tasks from t_2 to t_3. The simulation results of the first 10 tasks can be applied to the first time interval from t_1 to t_2 and likewise for the second time interval. Obviously, the time intervals in consideration must be large enough to minimize the effects of transient or boundary conditions.

4.5.1 CPU utilization patterns

All tasks in our simulations have constant nominal execution times, which are randomly chosen between the specified minimum and maximum task execution times with an equal probability. The task execution times and periods are selected based on typical tasks in open-architecture machine tool controllers. In particular, we have three different CPU utilization patterns: variable CPU utilizations (where task execution times range from 0.5% to 5% of their respective task periods), fixed task execution times (where all task execution times are 50 µs regardless of their periods), and fixed task CPU utilizations (where all task execution times are equal to 1% of their respective periods). We do not

consider variations in execution time and period, since their effects may be reflected in the changes of CPU demand in different time intervals.

4.5.2 Task period distributions

While the uniformly-distributed task periods may be a good approximation for many applications, we also observe that task period distributions can be *bimodal* in some other cases. For example, in our prototype open-architecture machine tool controllers, servo tasks have shorter periods (e.g., 1 or 10 *ms*) while supervisory and display tasks have longer periods (e.g., 40 or 100 *ms*). There are typically few tasks with periods in the middle of the range.

To evaluate the performance of scheduling algorithms in such situations, we introduce a bimodal distribution that consists of two mirror-imaged unit-mean exponential distribution functions. The random numbers generated from the bimodal distribution are limited to a range between 0 and 100, which is then scaled to the specified range of task periods. Figure 23 shows the histograms of 90 task periods used in one of our simulations, generated by the bimodal and uniform distributions respectively. In the top graph, task periods are generated by the bimodal distribution and most of them are less than 5 *ms* or between 95 and 100 *ms*. In the bottom graph, task periods spread everywhere within the entire period range using the uniform distribution.

4.6 Evaluation Results

We now examine the performance of the proposed heuristics (in particular, RM_CP1, RM_CP5, RM_CP9, CPB_RM, CPM, UM and UM_CP), as well as EDF, FIFO and RM, under different system load conditions. The results of this evaluation can be used by real-time application developers in their selection of a scheduling algorithm that is suitable for probabilistic guarantees.

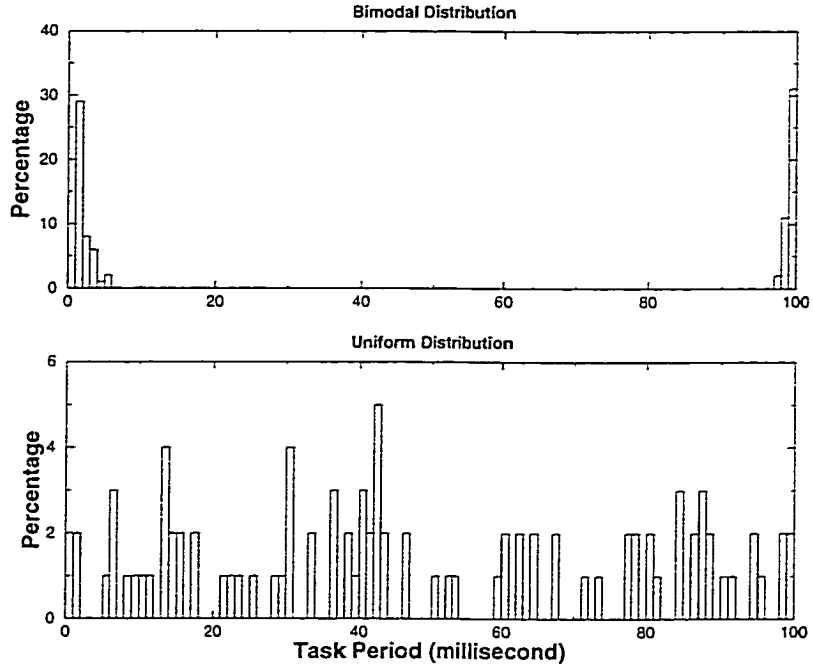


Figure 23: Bimodal and uniform task period distributions.

4.6.1 Variable CPU utilizations

Our first set of simulations uses variable CPU utilizations, where task execution times range from 0.5% to 5% of their respective task periods. Table 25 lists and Figure 24 plots the simulation results for the uniform task period distribution. The top graph uses the job-oriented objective function, showing the useful job ratios over number of tasks for individual scheduling algorithms. The second graph from the top instead uses our task-oriented objective function. The third graph plots the overall job deadline miss ratios, regardless of whether the completion probabilities of tasks are satisfied or not. The bottom graph plots the CPU demand over number of tasks.

From Table 25 and Figure 24, we have made the following observations:

1. When the system load is very light (e.g., CPU demand < 0.14), no job deadlines are missed under all scheduling algorithms. Thus, the useful job ratios and task completion probability miss ratios are 100% and 0%, respectively.

	# tasks	CPU demand	EDF	FIFO	RM	RM_CP1	RM_CP5	RM_CP9	CPB_RM	CPM	UM	UM_CP
Useful Job %	5	0.14	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Task CP Miss %			0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Job DL Miss %			0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Useful Job %	10	0.31	100.00	99.99	100.00	100.00	100.00	99.92	99.91	99.91	100.00	99.91
Task CP Miss %			0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Job DL Miss %			0.00	0.01	0.00	0.00	0.00	0.08	0.09	0.09	0.00	0.09
Useful Job %	15	0.47	100.00	99.89	100.00	100.00	99.96	99.76	99.72	99.58	100.00	99.56
Task CP Miss %			0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Job DL Miss %			0.00	0.11	0.00	0.00	0.04	0.24	0.28	0.42	0.00	0.44
Useful Job %	20	0.61	100.00	87.81	100.00	100.00	94.15	89.87	88.34	86.17	89.47	87.20
Task CP Miss %			0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Job DL Miss %			0.00	12.19	0.00	0.00	5.85	10.13	11.66	13.83	10.53	12.80
Useful Job %	25	0.76	100.00	83.95	100.00	100.00	91.60	87.01	85.11	81.03	88.22	82.03
Task CP Miss %			0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Job DL Miss %			0.00	16.05	0.00	0.00	8.40	12.99	14.89	18.97	11.78	17.97
Useful Job %	30	0.94	100.00	61.27	99.99	99.96	87.30	81.57	79.49	71.25	83.79	72.82
Task CP Miss %			0.00	3.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Job DL Miss %			0.00	30.76	0.01	0.04	12.70	18.43	20.51	28.75	16.21	27.18
Useful Job %	35	1.07	0.00	0.00	97.12	97.16	77.05	70.15	66.83	51.64	63.90	55.47
Task CP Miss %			100.00	100.00	14.29	11.43	11.43	11.43	11.43	14.29	14.29	11.43
Job DL Miss %			99.64	99.88	2.88	2.84	22.95	29.84	33.17	42.45	24.36	39.82
Useful Job %	40	1.19	0.00	0.00	95.57	93.10	72.43	63.84	49.83	45.02	65.08	54.47
Task CP Miss %			100.00	100.00	20.00	22.50	20.00	20.00	22.50	25.00	15.00	12.50
Job DL Miss %			99.85	99.95	4.43	6.90	27.57	36.16	41.90	52.29	30.14	45.54
Useful Job %	45	1.35	0.00	0.00	93.71	91.63	66.41	45.64	52.65	42.75	67.39	45.37
Task CP Miss %			100.00	100.00	28.89	28.89	28.89	35.56	28.89	31.11	20.00	28.89
Job DL Miss %			99.92	99.97	5.82	8.36	33.58	44.69	47.35	55.55	32.30	48.18
Useful Job %	50	1.51	0.00	0.00	93.20	90.70	64.82	44.90	36.81	34.27	72.03	54.12
Task CP Miss %			100.00	100.00	34.00	36.00	34.00	38.00	36.00	36.00	26.00	34.00
Job DL Miss %			99.92	99.98	6.80	9.03	34.94	48.31	63.20	65.73	27.73	42.89
Useful Job %	55	1.62	0.00	0.00	91.15	89.86	57.59	37.32	34.96	31.26	70.96	54.70
Task CP Miss %			100.00	100.00	40.00	38.18	40.00	43.64	43.64	43.64	27.27	34.55
Job DL Miss %			99.93	99.98	8.52	10.14	40.80	54.79	63.95	66.15	29.04	44.48
Useful Job %	60	1.76	0.00	0.00	89.48	88.25	53.27	36.21	32.14	30.89	69.37	53.69
Task CP Miss %			100.00	100.00	43.33	41.67	43.33	45.00	46.67	46.67	31.67	36.67
Job DL Miss %			99.93	99.98	10.50	11.75	46.07	59.81	65.45	67.33	30.60	46.08
Useful Job %	65	1.90	0.00	0.00	88.41	87.35	45.90	36.38	33.33	28.79	61.92	52.51
Task CP Miss %			100.00	100.00	44.62	43.08	49.23	47.69	47.69	49.23	38.46	40.00
Job DL Miss %			99.95	99.98	11.59	12.65	51.81	63.63	65.71	68.21	33.35	46.81
Useful Job %	70	1.97	0.00	0.00	86.87	86.41	45.57	34.10	30.89	26.65	60.36	49.86
Task CP Miss %			100.00	100.00	47.14	45.71	51.43	50.00	51.43	52.86	38.57	41.43
Job DL Miss %			99.95	99.98	13.13	13.33	53.91	65.24	67.88	70.42	34.74	46.38

Table 25: Performance of scheduling algorithms under variable CPU utilizations and uniform period distribution.

2. As the load increases but is still in the light-to-medium range (e.g., CPU demand < 0.76), some deadlines are missed under certain scheduling algorithms (such as FIFO, RM_CP5, RM_CP9, CPB_RM, CPM, UM and

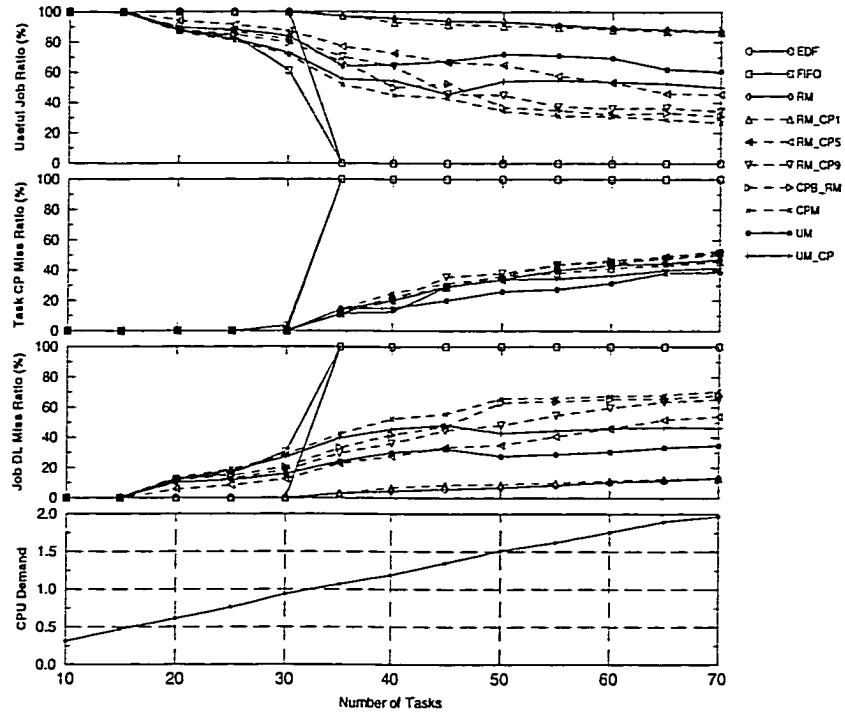


Figure 24: Performance of scheduling algorithms under variable CPU utilizations and uniform period distribution.

UM_CP) but not the others (such as EDF, RM and RM_CP1). So far, all tasks have met their completion probability requirements. The useful job ratios are equal to 100% minus their corresponding job deadline miss ratios.

3. When the CPU becomes overloaded, the completion probability miss ratios for EDF and FIFO jump to 100%, while those of other scheduling algorithms are more resilient and degrade gradually. There is no significant performance difference between our completion-probability-cognizant heuristics and RM, in terms of task completion probability miss ratio, even though there is a significant difference in their job deadline miss ratios. Comparing with RM and the completion-probability-cognizant heuristics, UM has better task completion probability miss ratios (UM_CP is also better but not as much), as the system load increases.

4. Similarly, when the CPU becomes overloaded, the useful job ratios for EDF and FIFO drop significantly, while those of other scheduling algorithms are more resilient, degrading gradually. Unsurprisingly, RM has the best useful job

ratio since it essentially gives higher priorities to tasks that invoke more jobs. Among completion-probability-cognizant heuristics, the more it deviates from RM, the worst its useful job ratio. Among CPU-utilization-cognizant heuristics, UM performs better than UM_CP. Their performances are in the middle range of RM and completion-probability-cognizant heuristics.

Figure 25 plots the simulation results for the bimodal task period distribution. Compared to Figure 24, useful job ratios, task completion-probability miss ratios, and job deadline miss ratios all show similar trends, though the relative performance of scheduling algorithms varies slightly. When the system is overloaded, RM and RM_CP1 remain the best in terms of the useful job ratio while UM and UM_CP are the best in terms of the task completion probability miss ratio.

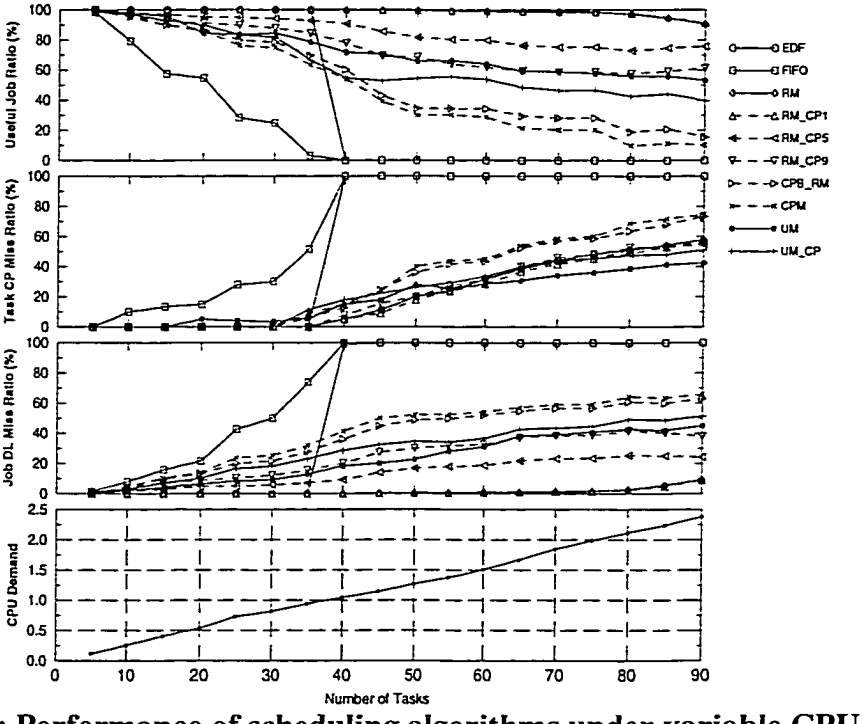


Figure 25: Performance of scheduling algorithms under variable CPU utilizations and bimodal period distribution.

4.6.2 Fixed task execution times

Our second set of simulations uses fixed task execution times, where all task execution times are 50 μs regardless of their periods. The simulation results are plotted in Figures 26 and 27 for the uniform and bimodal task period distributions, respectively. Since all task execution times are the same, tasks with shorter periods have larger CPU utilizations. Therefore, UM is the exact opposite of RM in this case. The performance of scheduling algorithms may be summarized as follows:

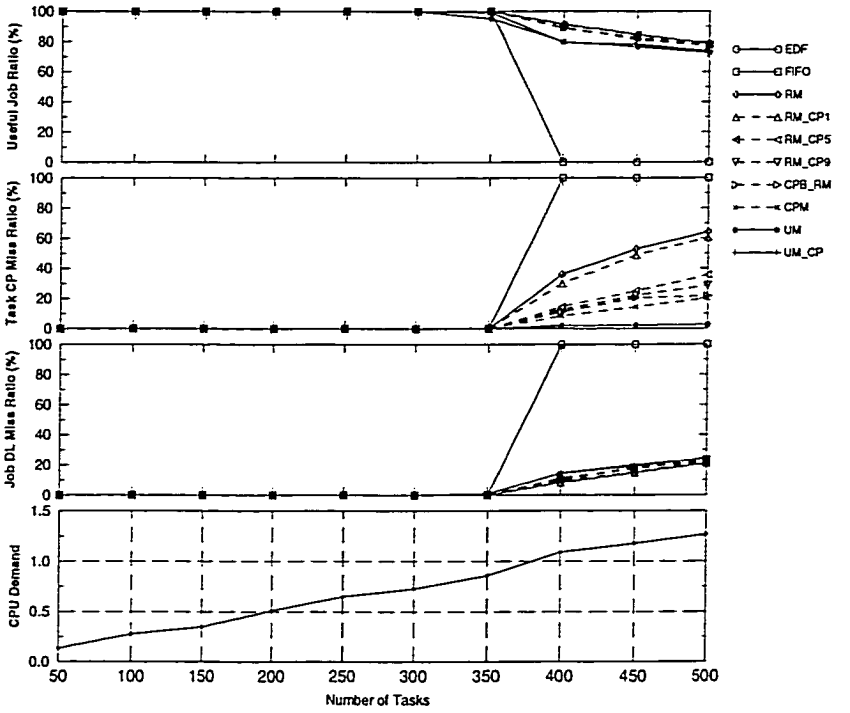


Figure 26: Performance of scheduling algorithms under fixed task execution times and uniform period distribution.

1. When the system is not overloaded, all scheduling algorithms perform very well in terms of both useful job ratio (which are 100% or close to 100%) and task completion probability miss ratio (which are zero or close to zero).
2. When the system is overloaded, the task completion probability miss ratios of UM and UM_CP are significantly lower than those of CPM and CPB_RM, which are significantly lower than those of RM_CP1 and RM, which in turn

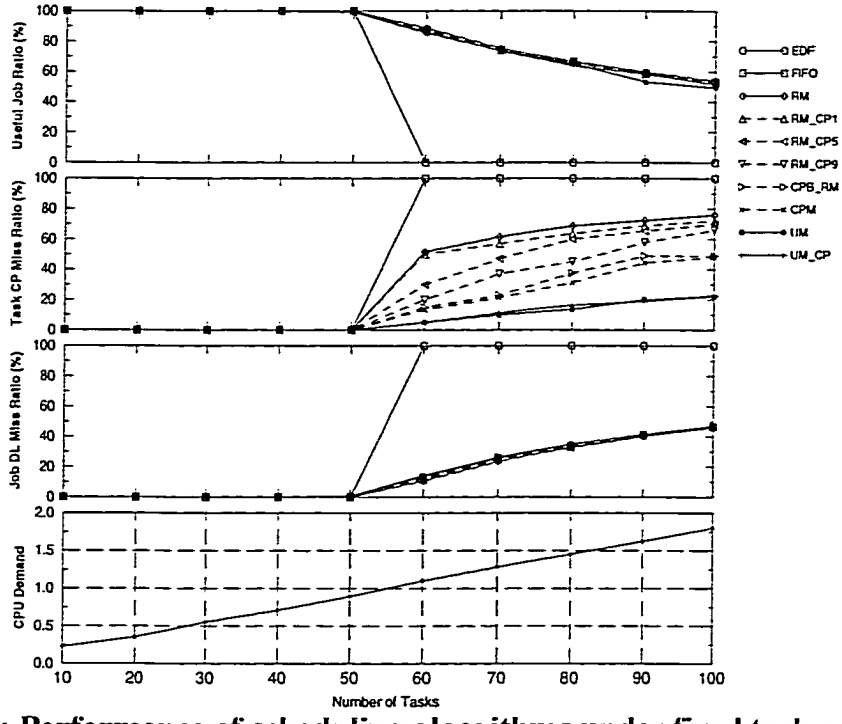


Figure 27: Performance of scheduling algorithms under fixed task execution times and bimodal period distribution.

are significantly lower than those of EDF and FIFO, which are almost 100%. The task completion probability miss ratios of RM_CP5 and RM_CP9 are between those of RM_CP1 and CPB_RM. Clearly, CPU-utilization-cognizant heuristics perform best. Giving higher priorities to tasks with lower CPU utilizations allows more tasks to meet their completion probability requirements. Among completion-probability-cognizant heuristics, ones that take task completion probability requirements more into account perform better than those that do not or consider less.

3. When the system is overloaded, the useful job ratios of all scheduling algorithms are similar except EDF and FIFO (which are zero). The lower task completion probability miss ratios of CPU-utilization-cognizant heuristics are offset by the smaller job numbers of the tasks that meet their completion probability requirements. Completion-probability-cognizant heuristics have a similar situation.

4.6.3 Fixed task CPU utilizations

Our last set of simulations uses fixed task CPU utilizations, where all task execution times are equal to 1% of their respective periods. When all tasks have the same fixed task CPU utilization, UM and UM_CP become the same as FIFO and CPM, respectively. The simulation results are shown in Figures 28 and 29 for the uniform and bimodal task period distributions, respectively.

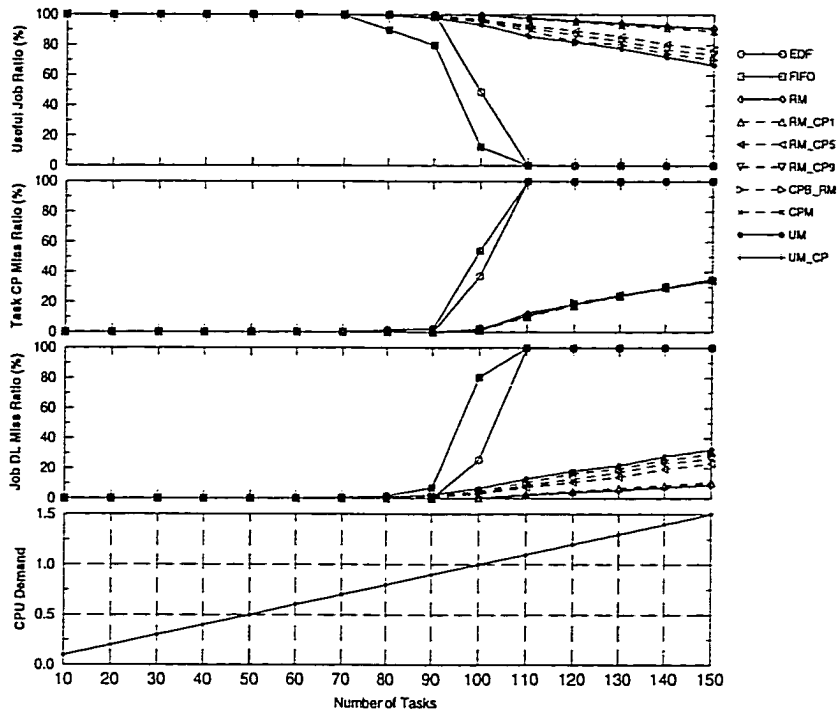


Figure 28: Performance of scheduling algorithms under fixed task CPU utilizations and uniform period distribution.

The useful job ratios and task completion probability miss ratios of FIFO, UM and EDF deteriorate quickly as the system load increases from medium to overloaded. All other scheduling algorithms perform similarly in terms of task completion probability ratio. In terms of useful job ratios, RM performs best while CPM or UM_CP is the worst. The performance of other heuristics is in between.

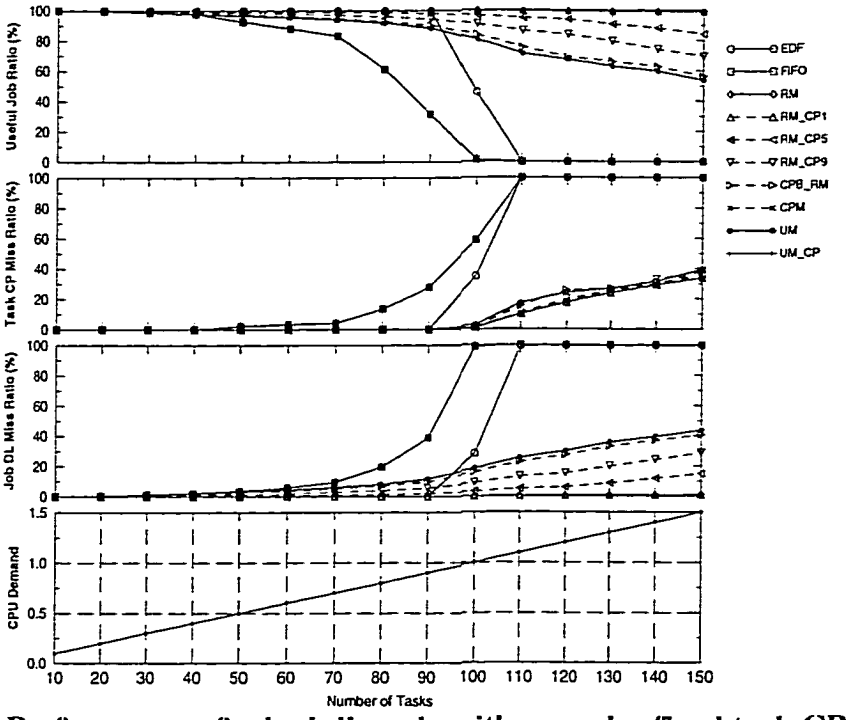


Figure 29: Performance of scheduling algorithms under fixed task CPU utilizations and bimodal period distribution.

4.6.4 Summary

From these simulations, we made the following observations for three different CPU utilization patterns:

1. Variable CPU utilizations

- When the system is lightly loaded, all scheduling algorithms perform well.
- When the system is temporarily overloaded, EDF and FIFO perform badly. RM and UM work best in terms of useful job ratio and task completion probability miss ratio, respectively. RM_CP1 performs close to RM while UM_CP does close to UM. The other completion-probability-cognizant heuristics (e.g., CP_RM5 and CP_RM9) perform average with respect to both job-oriented and task-oriented objective functions.

2. Fixed task execution times

- As long as the system is not overloaded, all scheduling algorithms perform well.

- When the system is temporarily overloaded, EDF and FIFO perform badly. All other scheduling algorithms perform similarly in terms of useful job ratio (though UM and UM_CP are slightly worse). CPU-utilization-cognizant heuristics (i.e., UM and UM_CP) are significantly better than RM and completion-probability-cognizant heuristics in terms of task completion probability miss ratio.

3. Fixed task CPU utilizations

- As long as the system is not overloaded, all scheduling algorithms perform well.

- When the system is temporarily overloaded, EDF and FIFO perform badly. UM and UM_CP are the same as FIFO and CPM, respectively. RM performs best in terms of useful job ratio. All scheduling algorithms (except EDF, FIFO and UM) perform similarly in terms of task completion probability miss ratio.

From the above observations, when the system is not overloaded, almost all scheduling algorithms perform well in terms of both useful job ratio and task completion probability miss ratio, regardless of load patterns. However, EDF and FIFO should not be used under temporary overload conditions. The choice among RM, completion-probability-cognizant heuristics and CPU-utilization-cognizant heuristics should depend on objective functions and load patterns. In general, RM and UM_CP perform well under all three system load patterns in terms of useful job ratio and task completion probability miss ratio, respectively.

4.7 Comparison between Probabilistic and Deterministic Guarantees

We have evaluated several scheduling algorithms in the context of probabilistic deadline guarantees. In particular, the performance of these scheduling algorithms is characterized in terms of useful job ratio (which is the percentage of jobs which meet their deadlines and whose tasks meet their completion probability requirements) and task completion probability miss ratio (which is the percentage of tasks whose completion probability requirements are not satisfied). We now compare the performance of these scheduling algorithms in the context of probabilistic and deterministic deadline guarantees. The difference between probabilistic and deterministic deadline guarantees is that the deadline of a task can be probabilistically guaranteed if the task completion probability is satisfied even when some jobs of the task miss their deadlines, while the task's deadline can only be deterministically guaranteed if there is not a single job missing its deadline. In other words, the same scheduling algorithms are evaluated with different objective functions.

For our comparison, we use three performance metrics. The first metric is CPU utilization when *all* task requirements are satisfied, similar to the achievable CPU utilization defined in Chapter 3 and [185]. It means zero task completion probability miss ratio for probabilistic deadline guarantees and zero job or task deadline miss ratio for deterministic deadline guarantees. This metric is intended to compare the effects of probabilistic and deterministic deadline guarantees for one special situation when all task requirements are met.

The second metric is the *task guarantee ratio*, which is defined as the percentage of tasks that can be guaranteed. This calculates the percentage of tasks whose completion probabilities are met for probabilistic deadline guarantees, which is equal to one minus the task completion probability miss ratio. For deterministic deadline guarantees, the task guarantee ratio is the percentage of tasks whose deadlines are met, which is equal to one

minus the task deadline miss ratio. Our second metric is intended to gauge the performance difference from a task-oriented perspective as the system load changes.

The third metric is the *job guarantee ratio*, which is defined as the percentage of jobs that can be guaranteed when the corresponding tasks of these jobs can meet their completion probability requirements. The job guarantee ratio is equal to the useful job ratio for probabilistic deadline guarantees. For deterministic deadline guarantees, the job guarantee ratio is the percentage of jobs whose deadlines are met when no other jobs of their corresponding tasks miss their deadlines. Our third metric is intended to gauge the performance difference from a job-oriented perspective as the system load changes.

We will illustrate the performance differences between probabilistic and deterministic deadline guarantees under three different CPU utilization patterns—variable CPU utilizations, fixed task execution times, and fixed task CPU utilizations.

Figures 30 and 31 show the performance with variable CPU utilizations of different scheduling algorithms for probabilistic and deterministic deadline guarantees, under the uniform and bimodal task period distributions, respectively.

In Figure 30, the bottom graph shows the CPU utilizations for the scheduling algorithms when all task requirements are satisfied. The three rightmost bars, labeled EDF_h, FIFO_h and RM_h, represent the CPU utilizations for EDF, FIFO and RM when all (hard) task deadlines are met, respectively. All others are for probabilistic deadline guarantees. In this case, the introduction of probabilistic deadline guarantees significantly improves the CPU utilizations for FIFO and RM over deterministic deadline guarantees, though there is little difference for EDF. All our heuristics for probabilistic guarantees have high CPU utilizations.

The middle graph of Figure 30 plots the task guarantee ratio (Y axis) over system load (X axis). When the system is not overloaded, scheduling algorithms for both probabilistic and deterministic deadline guarantees perform very well, except FIFO for deterministic deadline guarantees. When the system is temporarily overloaded, EDF and

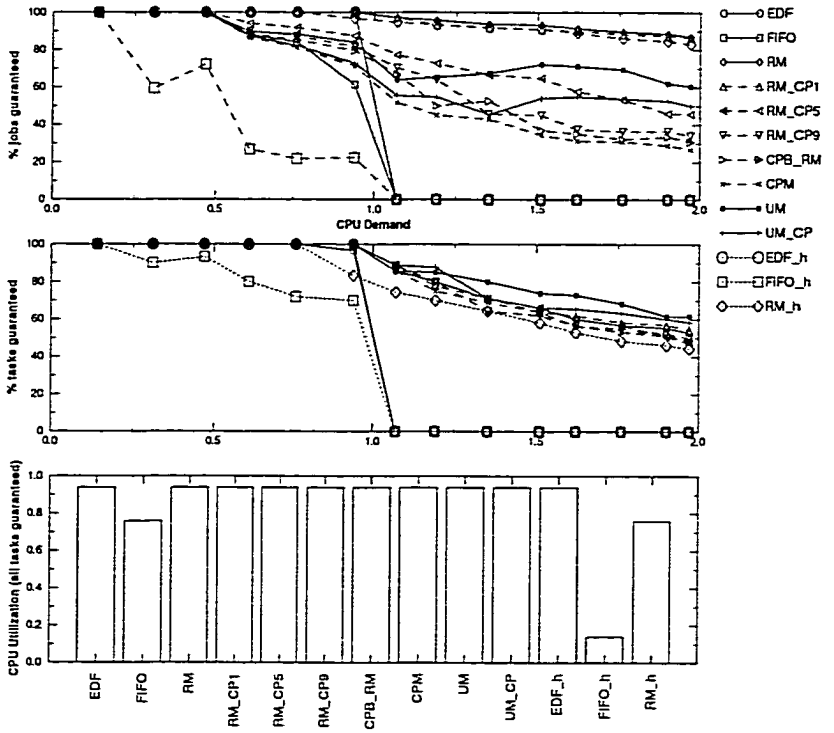


Figure 30: Performance comparison between probabilistic and deterministic guarantees under variable CPU utilizations & uniform task period distribution.

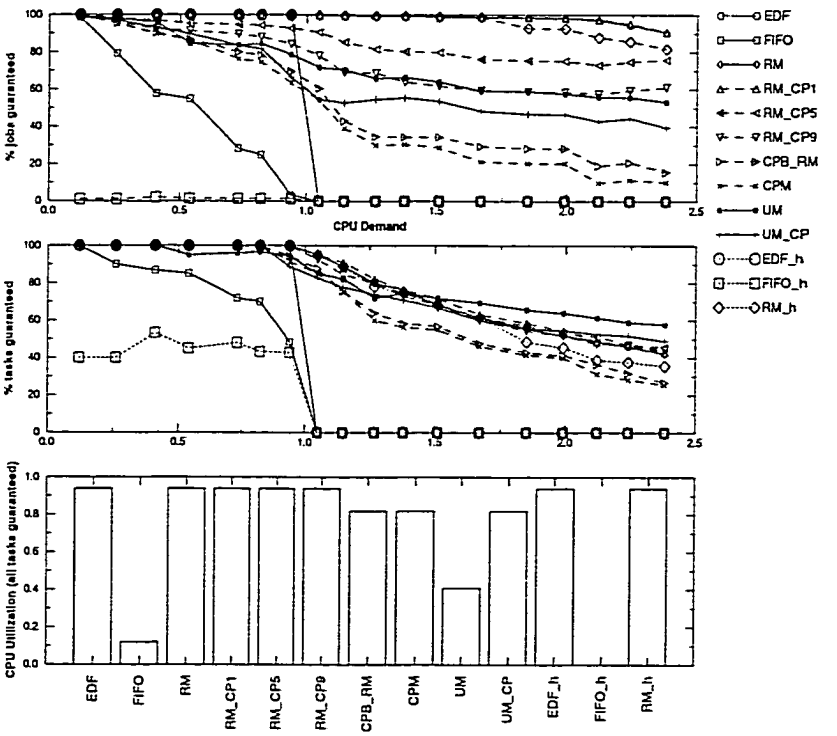


Figure 31: Performance comparison between probabilistic and deterministic guarantees under variable CPU utilizations & bimodal task period distribution.

FIFO for both probabilistic and deterministic guarantees can no longer guarantee any deadlines. Both are prone to the *domino effect*, where the first job that misses its deadline may cause all subsequent jobs to miss their deadline. This is because they are trying to do the impossible—giving fair chance for all tasks to run when there are simply not enough CPU cycles available to do so. The performance of other scheduling algorithms degrades more gradually because they assign fixed priorities to tasks. Consequently, low-priority tasks may never get a chance to run and thus give higher-priority tasks some opportunity to meet their requirements. The performance of RM for deterministic guarantees is not as good as that of RM for probabilistic guarantees, simply because the task requirements of the former is more stringent than the latter. Our CPU-utilization-cognizant heuristic UM is more resilient as the system load increases.

The top graph of Figure 30 plots the job guarantee ratio over system load. When the system is lightly-loaded, both scheduling algorithms for probabilistic and deterministic deadline guarantees perform very well, except FIFO for deterministic deadline guarantees. As the system load increases, performance of all scheduling algorithms gets worse. When the system becomes temporarily overloaded, EDF and FIFO for both probabilistic and deterministic guarantees can no longer guarantee any deadlines. The performance of other scheduling algorithms again degrades more gradually. The performance of RM for both probabilistic and deterministic guarantees and RM_CP1 is significantly better than the others. The performance of RM for probabilistic guarantees is slightly better than that of RM for deterministic guarantees.

Figure 31 shows the result of the same comparison under the bimodal task period distribution. The CPU utilizations of EDF, FIFO and RM for probabilistic guarantees are better than or the same as their counterparts for deterministic guarantees. The CPU utilizations of CPB_RM, CPM, UM and UM_CP when all tasks are guaranteed are not as good as those in the uniform period distribution case. The performance of scheduling

algorithms under the bimodal task period distribution shows the similar trend as that under the uniform distribution.

Figures 32 and 33 show the scheduling algorithm performance with fixed task execution times, under uniform and bimodal task period distributions, respectively. Under both task period distributions, the CPU utilizations of the scheduling algorithms EDF, FIFO and RM with probabilistic deadline guarantees are equal to or better than their counterparts with deterministic deadline guarantees when all task requirements are met. When the system is not overloaded, all scheduling algorithms perform relatively well except FIFO_h. When the system becomes overloaded, the CPU-utilization-cognizant heuristics (i.e., UM and UM_CP) perform better than the completion-probability-cognizant heuristics (i.e., CPM, CPB_RM, RM_CP9, RM_CP5 and RM_CP1) in terms of task guarantee ratio. The latter perform better than RM with probabilistic guarantees, which perform better than RM with deterministic guarantees. All heuristics and RM have similar performance in terms of job guarantee ratio. EDF and FIFO with either probabilistic or deterministic deadline guarantees have the worst performance in terms of both task and job guarantee ratios.

Figures 34 and 35 show the scheduling algorithm performance with fixed task CPU utilizations, under uniform and bimodal task period distributions, respectively. In this case, UM is equivalent to FIFO for probabilistic deadline guarantees, while UM_CP is the same as CPM. All scheduling algorithms except FIFO_h perform well when the system is not overloaded. When the system is overloaded, EDF, FIFO and UM quickly become ineffective while others have similar, relatively good performance in terms of task guarantee ratio. RM and completion-probability-cognizant heuristics perform better than CPU-utilization-cognizant heuristics in terms of job guarantee ratio. EDF, FIFO and RM for deterministic deadline guarantees perform worse than their counterparts for probabilistic deadline guarantees.

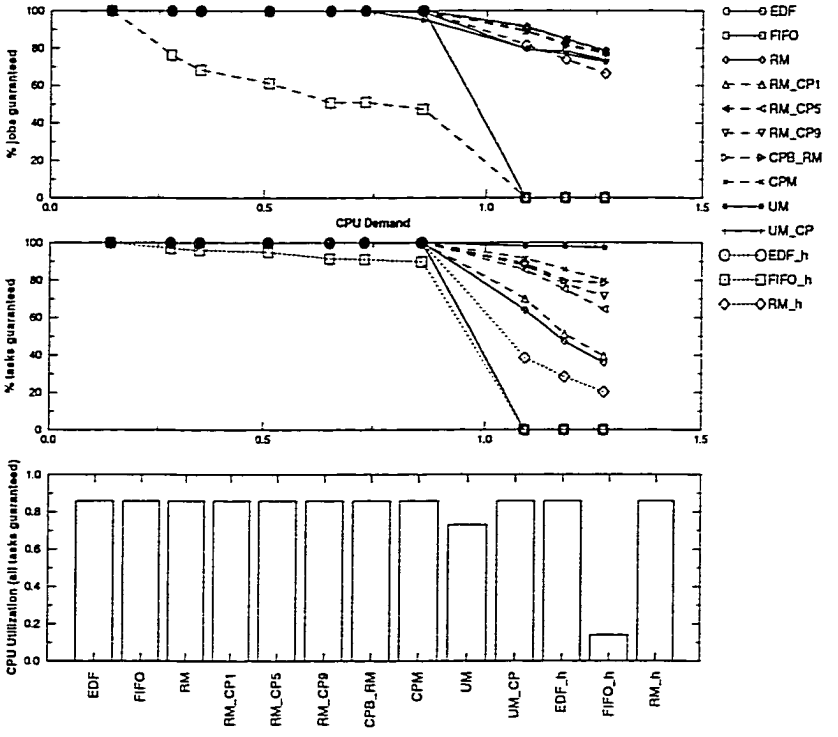


Figure 32: Performance comparison between probabilistic and deterministic guarantees under fixed task execution times & uniform task period distribution.

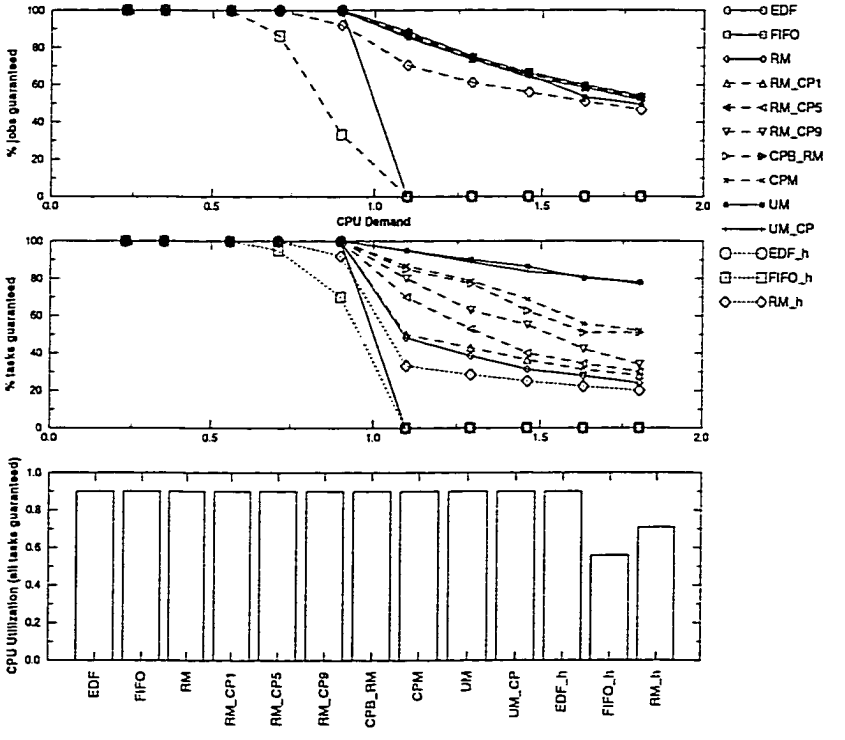


Figure 33: Performance comparison between probabilistic and deterministic guarantees under fixed task execution times & bimodal task period distribution.

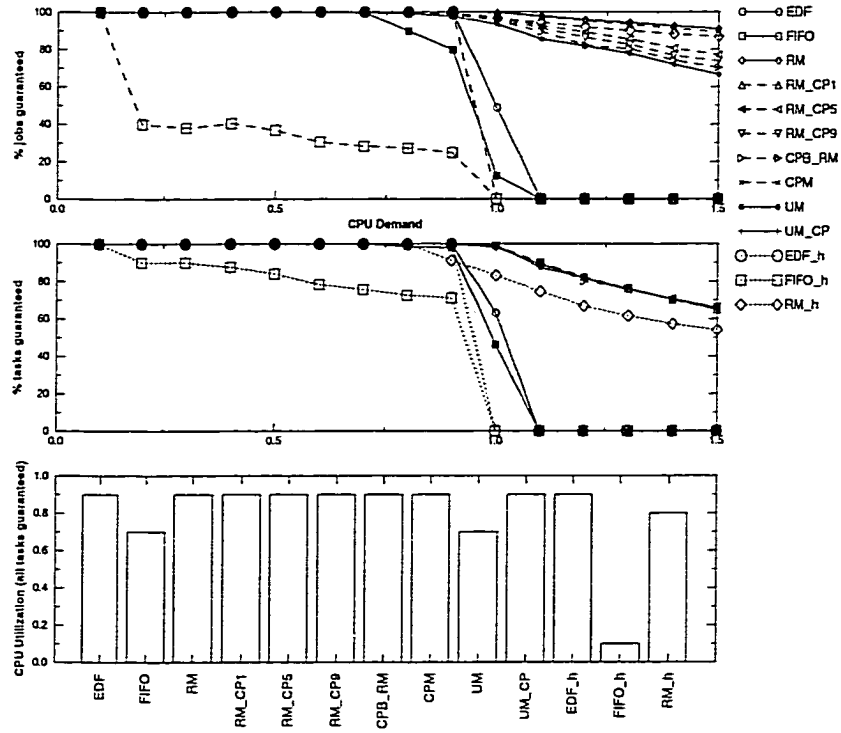


Figure 34: Performance comparison between probabilistic and deterministic guarantees under fixed task CPU utilizations & uniform task period distribution.

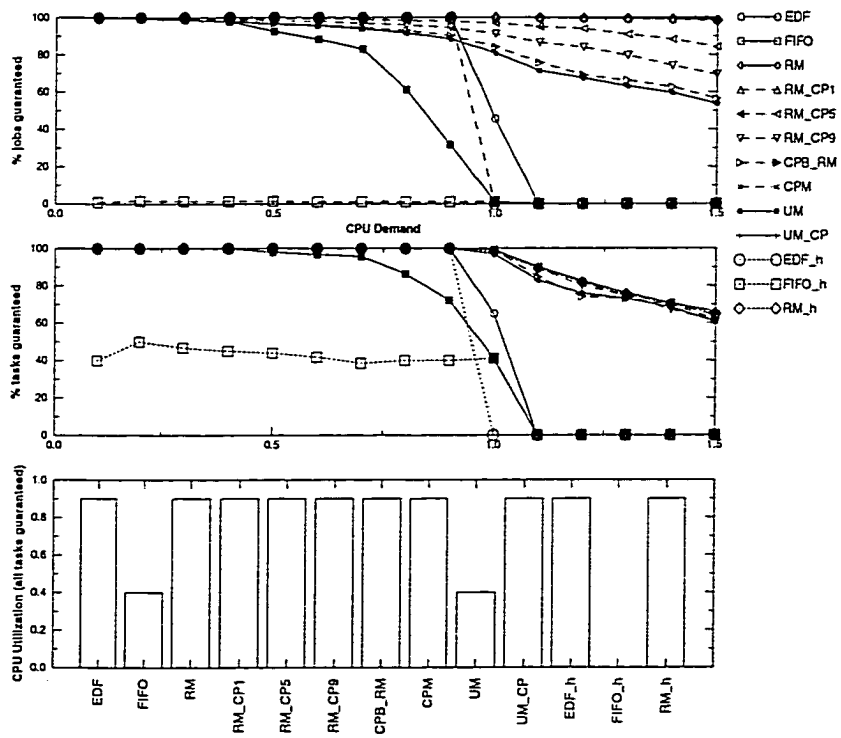


Figure 35: Performance comparison between probabilistic and deterministic guarantees under fixed task CPU utilizations & bimodal task period distribution.

In short, by introducing the concepts of completion probability and probabilistic deadline guarantee, EDF, FIFO and RM can provide the same or better performance than their counterparts for deterministic guarantees, in terms of CPU utilization, task and job guarantee ratios. When the system is temporarily overloaded, in general, RM and completion-probability-cognizant and CPU-utilization-cognizant heuristics are particularly effective for providing probabilistic deadline guarantees.

4.8 Related Work

Concepts similar to the completion probability can be found in the literature. Our completion probability associates the deadline of a real-time task with the required probability that the task must complete by its deadline. This concept is similar to the *guarantee level* proposed by Kim and Song [75, 76] and the *guarantee probability* by Kamat, Malcolm and Zhao [69]. They essentially describe the same concept differently. The guarantee level is defined as the degree of criticality of the constraints of a real-time transaction. The main difference between completion probability and guarantee probability is that the former emphasizes requirements while the latter emphasizes system capability. For example, a real-time task may require a 0.9 probability of meeting its deadline (i.e., a completion probability of 0.9 or a guarantee level of 9), but the real-time system may be able to guarantee only a 0.8 probability of completing the task by its deadline (i.e., a guarantee probability of 0.8).

Our PRTCM can also be considered a superset of the concept of *(n m)-hard deadlines* introduced by Koren and Shasha [79] and expanded by Bernat and Burns [14]. PRTCM allows not only the specification of (n m)-hard deadlines but also semantics

beyond them. For example, a *skip parameter* [79] of 3 means that after missing a deadline at least 2 jobs must meet their deadlines. In PRTCM, this is equivalent to the following:

$$P_c(i) = \begin{cases} 1, \text{ if job } j \text{ missed its deadline where } j = i-2, i-1 \\ 0, \text{ otherwise} \end{cases}$$

We could extend the concept of (n m)-hard deadlines to *(n m)-probabilistic deadlines* and PRTCM would still be able to express the semantics. For example, we could extend the above example to mean that after missing a deadline at least 2 jobs must meet their deadlines with a probability of 0.99; otherwise a job must meet its deadline with a probability of 0.2. Such requirements would not be uncommon results of Quality-of-Service (QoS) based Service Level Agreements (SLAs). Our example may correspond to a SLA that guarantees not only a minimum service level (with a few escape clauses) but also an average (better) service level. In PRTCM, such a SLA may be described as follows:

$$P_c(i) = \begin{cases} 0.99, \text{ if job } j \text{ missed its deadline where } j = i-2, i-1 \\ 0.2, \text{ otherwise} \end{cases}$$

There is significant amount of work on scheduling under overload conditions. For example, Locke [95] has shown that EDF is prone to the domino effect and its performance rapidly degrades during overload intervals. Maruchek and Strosnider [102] also evaluated the graceful degradation properties of real-time scheduling algorithms. Their findings are consistent with our simulation results for EDF, FIFO and RM.

A number of heuristic algorithms have been proposed to improve the performance of EDF under overload conditions [21, 28, 51, 79, 155, 161]. Baruah *et al.* [13] have

shown that there exists an upper bound on the performance of any on-line preemptive scheduling algorithms under overload conditions. This upper bound is in terms of the cumulative value guaranteed by a clairvoyant scheduler (i.e., one that knows the future). The value associated to each job is equal to the job's execution time if it completes by its deadline; otherwise, the value is zero. Buttazzo, Spuri and Sensini [25] compared EDF and several value-based heuristics with different guarantee mechanisms (e.g., admission control and resource reclamation) to avoid the domino effect and achieve graceful degradation during transient overloads. Our focus is instead on fixed-priority heuristics to deal with overloads using completion probability based metrics (as opposed to cumulative values).

Heidmann [54] and Tia *et al.* [163] provided probabilistic schedulability analyses of periodic tasks where the computation time of each job is a random variable. Our work focuses on the probabilistic requirements model and scheduling heuristics to deal with overload conditions.

Although there exists work on the probabilistic approach in database systems (e.g., [46, 166, 170]), StarBase [75, 76] is the only real-time database (RTDB) we are aware of that addresses the probabilistic deadline guarantee issue. The StarBase RTDB model includes three classes of real-time transactions: Class I (hard transactions) requires a 100% guarantee, Class II (critical transactions) requires probabilistic (or statistical) guarantees, and Class III (soft or firm transactions) requires only best-effort. StarBase seeks to minimize the number of high-priority transactions which miss their deadlines, and discards tardy transactions at their deadline points. Because it assumes no *a priori* knowledge about the transaction load, StarBase does not provide any deadline guarantees.

Scheduling with completion probability requirements is also similar to stochastic scheduling in operations research, e.g., [23, 122]. However, stochastic scheduling typically deals with the situation where the system services may be unavailable with

certain probability. Our work differs in that we are concerned with the unpredictability of service performance but assume that the services are always available.

4.9 Summary

We presented three of the four components of our proposed practical framework for probabilistic deadline guarantees:

- *PRTCM*: allows the tolerance of application task deadline misses to be specified in terms of completion probability, in addition to other requirements. This also clearly describes the semantics of probabilistic deadline guarantees.
- *Heuristics for probabilistic deadline guarantees*: that are completion-probability-cognizant, such as RM_CP*n*, CPB_RM and CPM, as well as CPU-utilization-cognizant, such as UM and UM_CP.
- *Evaluation of heuristics and scheduling algorithms*: in terms of useful job ratio and task completion probability miss ratio. In particular, we investigated how these scheduling algorithms perform when the system is temporarily overloaded using three system load patterns: variable CPU utilizations, fixed task execution times, and fixed task CPU utilizations. Our simulations showed that, in general, all scheduling algorithms for probabilistic deadline guarantees perform well when the system is not overloaded, regardless the system load patterns. However, when the system is temporarily overloaded, there can be a significant performance difference between the algorithms. EDF and FIFO typically have almost zero useful job ratios and 100% task completion probability miss ratios under such overload conditions due to the domino effect. Therefore, EDF and FIFO should not be used to handle temporary overload conditions. The relative performance of RM, completion-probability-cognizant heuristics and CPU-utilization-cognizant heuristics vary, depending on system load patterns and evaluation objective functions. In general, RM performs best in terms of useful job ratio under all three system load patterns.

UM performs best in terms of task completion probability miss ratio under variable CPU utilizations and fixed task execution times, but it is not effective under fixed task CPU utilizations. As an alternative, UM_CP generally performs well in terms of task completion probability miss ratio under all three system load patterns. We also showed that, by introducing the concepts of completion probability and probabilistic deadline guarantee, EDF, FIFO and RM can provide the same or better performance than their counterparts for deterministic guarantees.

The last component of the framework, MBST, will be described in next chapter.

CHAPTER 5

MEASUREMENT-BASED SIMULATION TECHNIQUE

5.1 Introduction

To provide probabilistic deadline guarantees, the task completion time distributions must be known. There can be many approaches to obtaining task completion time distributions. At one extreme of the spectrum is to use formal analyses or simulations that assume idealized computing environment, as in the case of the RM analysis [93]. However, RTOSs have significant unpredictability. This makes any approaches that assume idealized system environment difficult and unreliable.

At the other extreme of the spectrum is to rely solely on actual measurements. However, if one measures the application performance without understanding the system or the application, the results will be good only for that particular configuration of the application and its computing environment. Such a method will be of little value because it cannot predict the application performance in a different computing environment (e.g., using a different scheduling algorithm) or a slightly different application in the same environment.

To achieve a balance between the two extremes and maximize the benefits of both, we propose a *Measurement-Based Simulation Technique* (MBST) for making probabilistic deadline guarantees. MBST uses individual application task execution times (measured in isolation) as inputs, models task interaction and system overhead, and generates task completion time distributions to determine whether probabilistic deadline guarantees can

be made. Applying MBST to our prototype open-architecture milling machine controllers, MBST is shown to produce simulation results that match very well the actual measurements. It can also be used to predict the performance of tasks that have not yet been fully implemented.

The remainder of this chapter is organized as follows. Section 5.2 gives details of the components and strategies of MBST. In Section 5.3, we demonstrate the validity and usefulness of MBST by applying it to our prototype open-architecture machine tool controllers. Sections 5.4 includes discussions and related work, while Section 5.5 summarizes the chapter.

5.2 MBST: Measurement-Based Simulation Technique for Probabilistic Deadline Guarantees

A common characteristic of embedded real-time systems is that information about the application tasks and computer system configurations is known *a priori*. For example, before a machine tool controller is put in place, the number and type of controller tasks that can possibly run simultaneously are known, as well as the computer setup. With this information, it is possible to provide probabilistic deadline guarantees by evaluating the real-time system performance. In order to determine whether probabilistic deadline guarantees can be provided, we must obtain the completion time distributions of individual tasks. For example, given the deadline and completion probability requirements of a periodic task, we will be able to check whether its deadline can be guaranteed if the task completion time (relative to its release times) is known.

Given both application tasks and the computer system, the objective of MBST is to determine if the completion probability requirements of individual application tasks can be met. To achieve this objective, MBST uses *measured* (in isolation) application task component performance data as input, models task interaction and system overhead, and

generates task completion time distributions. It takes two stages to apply MBST to any real-time system: validation and operation.

In the validation stage, the validity of the MBST is checked against actual task execution time measurements of the target real-time application. While in theory the task release times are observable, they are low-level system events (e.g., firings of POSIX timers) that are not easily accessible to user processes. In our experiments with open-architecture machine tool controllers, we used easily measurable VMEbus events, which are generated by a simple function call and recorded by the VME StopWatch (see Section 5.3). We place such a VMEbus event-generating function at the start of the code segment that we want to measure and another one at the end of the segment. Since a task may or may not start execution immediately after it is released, the resulting measurement is the start and end times of the task segment, against which MBST will be validated. More specifically, we will compare the measured execution time distributions of individual tasks and the corresponding simulated distributions.

Once MBST is validated using task execution time distributions, it is in the operation stage, where the task release times and end times from the simulations are used for providing probabilistic deadline guarantees. The key assumption here is that if the simulated task execution time distributions from MBST are valid, the corresponding task completion time distributions are also valid for the following reasons. First of all, the calculations of task execution time and completion time share a common element, task end time. Second, the other two elements, task release and start times, are closely correlated. A task cannot start execution until it is released. In the absence of preemption and blocking, a task will start execution immediately after it is released, resulting in the same value for release time and start time. Therefore, if we model the task preemption and blocking appropriately, valid task execution time distributions will lead to valid task completion time distributions. Furthermore, the task preemption and blocking affect task execution time distributions as well. An inappropriate task preemption and blocking

model will be unable to produce valid task execution time distributions in all cases. Consequently, valid task execution time distributions by MBST will result in valid task completion time distributions and validate probabilistic deadline guarantees.

MBST consists of three models: task model, run-time model and simulation model, as illustrated in Figure 36. We will describe these models in details in the following sections.

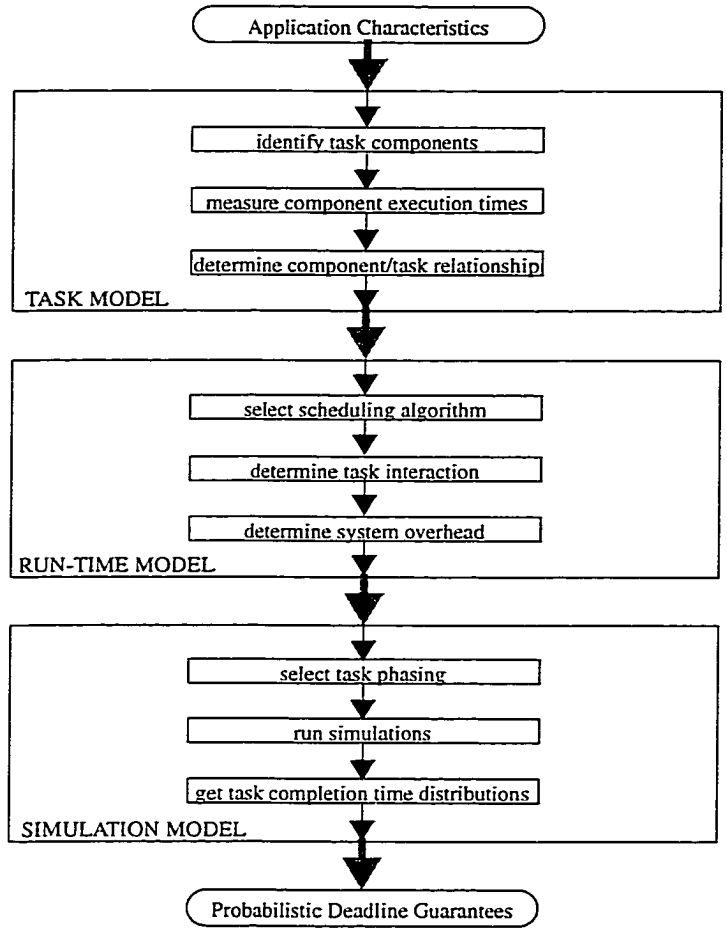


Figure 36: Measurement-based simulation technique (MBST).

5.2.1 Task model

An important feature of MBST is to predict *overall* task performance by using *individual* task component data. The task model is crucial to this feature. The first step is to identify

logical components of application tasks. A logical component is a part of the application task that implements some specific functionality. The reason we call it "logical" is that it may not have been implemented when we apply MBST. This allows MBST to be used in "what if" analyses. For example, we may want to know a task's performance when it needs to read twice as many sensors before implementing it. With MBST, we can approximate the new task by adding corresponding task components using existing data, as shown later in Section 5.3.4.6. The task components may be application-dependent.

The next step of the task model is to measure the actual execution times of individual task components. Because all practical systems have inherent unpredictability, using measured data as inputs to our simulation model produces more realistic results.

To measure the execution times of individual task components, the corresponding task is run *in isolation* with the specified period in the target computer system. Running tasks in isolation gives us better control of the measurement conditions. The interaction among tasks will be addressed in the run-time model.

Given the execution time measurement data of individual task components, the simulation software must be able to generate overall task execution times that would be close to the measured overall task execution times. This is not as simple as it may appear. Initially, we assume that the task component execution times follow normal distributions. We feed the statistics of each task component execution time (mean, standard deviation, minimum and maximum) to the simulator. Whenever a task execution time is needed, the simulator will generate individual task component execution times independently, based on their respective statistics. The overall task execution time is the summation of the individual task component execution times. However, task component execution times may not necessarily follow normal distributions. Consequently, the simulator-generated task execution time distribution may differ from the actual measurement data, which is the case in our prototype open-architecture milling machine controller tasks. Figure 37 compares the measured task execution time distribution with several simulator-generated

ones for the XYZ Servo task. We can see that the task execution time distribution using the normal distribution assumption does not match the actual measurement well.

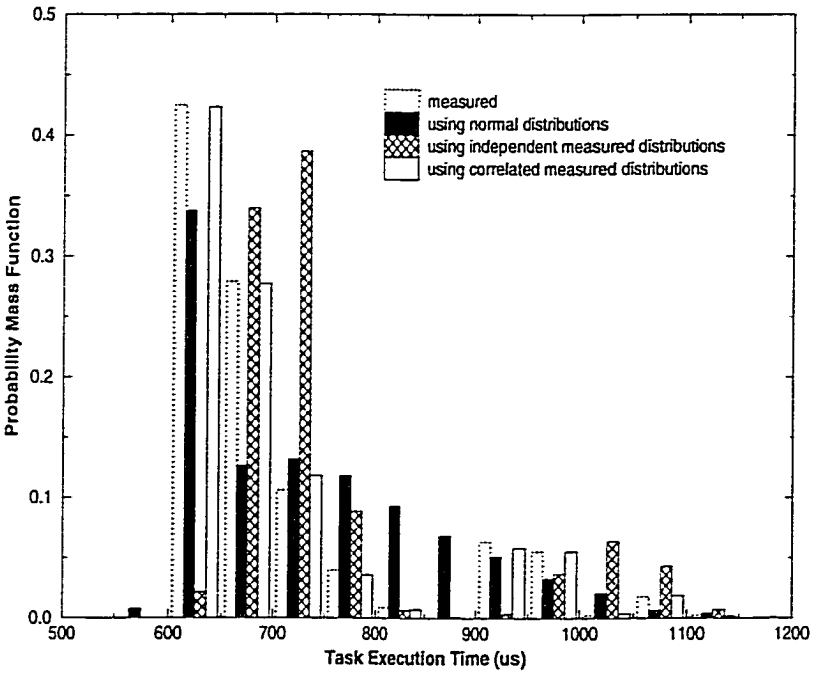


Figure 37: Comparison of data-generating approaches (XYZ Servo task).

Considering that the execution time of a real task may not necessarily follow any particular distribution, we modify our simulator to use the actual task component measurement data directly. An ordered sequence of measured execution time is stored in the simulator for each task component. To obtain a task component execution time, the simulator will generate an integer between 1 and the length of the sequence with equal probability and use the integer as an index to retrieve a stored execution time. An overall task execution time is the summation of individual task component execution times, retrieved *independently* (i.e., the indices are generated independently). We call this an "independent combination" approach. Figure 37 shows the XYZ Servo task execution time distribution generated by this approach, which is labeled "using independent measured distributions."

Though the above approach does not assume any specific statistical distribution, its result is still far from being satisfactory, because the execution times of individual task

components of the same task are not independent. For example, when the communication component of the XYZ Servo task (see Section 5.3.4.1) gets reference inputs from the message queues, the interpolator component may need to process the data to generate a series of desired values. When the communication component does not read message queues, the interpolator component also has less work to do. To solve the problem, we need to preserve the dependencies in the original task component measurement data. The measurement data of individual task components must be ordered appropriately: the execution time of the first task component of the first invocation of the task is measured, then the second component of first invocation, ..., last component of first invocation, first component of second invocation, ... Therefore, the overall execution time of the first task invocation is comprised of the first measurement of all task components. Using this ordering information, we modify the simulator to use a single index for all task components, as opposed to using independent indices. We call this "synchronized combination." The XYZ Servo task execution time distribution generated by this approach matches the measured distribution very well, as shown in Figure 37.

5.2.2 Run-time model

The next major part of MBST is the run-time model. The first step is to choose an appropriate real-time scheduling algorithm, based on application characteristics. The results of our research described in Chapter 4 can be used for this selection process.

Once the scheduling algorithm is chosen, we need to determine the relationships among tasks. In order words, which tasks may be blocked or preempted by which tasks. Task interaction may depend on the scheduling algorithm. For example, there will be no blockings among tasks if FIFO is used. On the other hand, blockings and possibly priority inversion must be considered if a priority-based preemptive scheduling algorithm is used. This is one of the most crucial and difficult steps in MBST and requires application domain and system knowledge.

The next step is to determine the system overhead, such as scheduling overhead and context switching overhead. Fixed-priority scheduling algorithms (such as RM and FIFO) incur very low run-time scheduling overhead, while dynamic-priority ones (such as EDF) incur high overhead. Similarly, this step requires in-depth system knowledge.

5.2.3 Simulation model

To establish the simulation model, the key step is to determine task *phasing*—initial task release times relative to each other. Depending on the phasing, task completion time distributions can be quite different. Suppose two periodic tasks τ_1 and τ_2 have constant periods of 10 *ms* and 20 *ms* and constant nominal execution times of 2 *ms* and 3 *ms*, respectively. Using the RM scheduling algorithm, τ_1 will have a higher priority than τ_2. If τ_2 is released initially at the same time as τ_1 or less than 3 *ms* before τ_1, τ_2 will have a constant completion time of 5 *ms* (relative to its release time). This is the worst phasing for τ_2, because τ_2 will always be preempted by τ_1 for the longest time—the entire execution time of τ_1. However, if τ_2 is released between 2 *ms* and 7 *ms* after τ_1, τ_2 needs only 3 *ms* from its release to completion. This represents the best phasing for τ_2 where two tasks have no contention at all.

Task phasing selection plays an important role in our model validation strategy. Because of timer interval variation, for example, the actual task phasing may be unpredictable in practical systems. Therefore, it is very difficult for MBST to replicate the actual task phasing. Consequently, it does not make sense to directly compare any *one* task execution time distribution generated by MBST with the actual measurement. To overcome this difficulty in validating MBST, we instead use a *bounding* strategy. In our simulations, we choose one best task phasing where there is least contention among tasks and one worst phasing where there is most contention. If the resulting two task execution time distributions can bound the actual measurement for every task, MBST is validated.

MBST uses two task phasing simulation configurations, best and worst, to bound the actual task execution time measurements. Both simulation configurations use constant task intervals to obtain deterministic task phasing. In the best configuration, the initial task release times are manually[1] adjusted to minimize contention among tasks. Suppose there are two tasks τ_1 and τ_2. Their nominal periods are 1 *ms* and 10 *ms* and their nominal execution times are 80 μs and 750 μs, respectively. The best phasing would be that the release time of τ_2 is between 80 μs and 250 μs after that of τ_1. Considering variations of their respective task execution times, we may set the interval between their release times to, say, around 165 μs, such that the two tasks are least likely to content with each other. In the worst configuration for task execution times, every task is released just before all higher priority tasks. Therefore, the task will always be preempted by all tasks that have higher priorities, resulting in the longest execution time.

While the best and worst simulation configurations use constant task intervals, task intervals have variations in reality. Such variations could increase task contention for the best configuration and decrease contention for the worst configuration. To approximate the actual best and worst task performance, MBST uses two more simulation configurations, random start and same start, respectively. These two configurations use the measured task intervals instead. In the random start configuration, the initial release of each task is selected randomly between 0 and its task period minus its execution time, while all tasks are released initially at the same time in the same start configuration. Table 26 summarizes the characteristics of these simulation configurations.

5.2.4 Probabilistic deadline guarantees

After MBST is validated, we can then obtain completion time distributions of individual tasks, based on the worst task phasing to be conservative. Using this information and task

1. When the number of tasks is large, a tool that uses optimization techniques such as linear programming will be helpful.

simulation configuration	task phasing	task interval	task execution time
best	least contention	constant	measured
random start	less contention	measured	measured
same start	more contention	measured	measured
worst	most contention	constant	measured

Table 26: MBST simulation configurations.

requirements, we will be able to determine if probabilistic deadline guarantees can be provided. Section 5.3 demonstrate the validity and usefulness of MBST in the development of machine tool controllers.

5.3 Application of MBST to Open-Architecture Machine Tool Controllers

For open-architecture machine tool controllers, we apply MBST based on a generic open-architecture model with a (conceptual) central data repository, as illustrated in Figure 38. Rectangles are functional units of an open-architecture controller. They provide and obtain data from the central data repository, which can be either conceptual or physical.

The part program provides reference inputs, such as positions and velocities, to the controller. An interpolator takes the reference inputs and decomposes them into a series of desired values. Sensors provide information about the actual machine status, such as its current positions, velocities and force. The supervisory control unit uses some or all sensor values to determine if and how certain desired values generated by the interpolators should be overridden. The servo control unit takes sensor values, desired values and overriding values as inputs and computes control commands, which are sent to the actuators.

With this architectural model, any control functional units can be easily replaced by other units with similar functionality as long as their interface with the data repository

remains the same. For example, a PID servo control unit can be substituted with a fuzzy logic servo control unit.

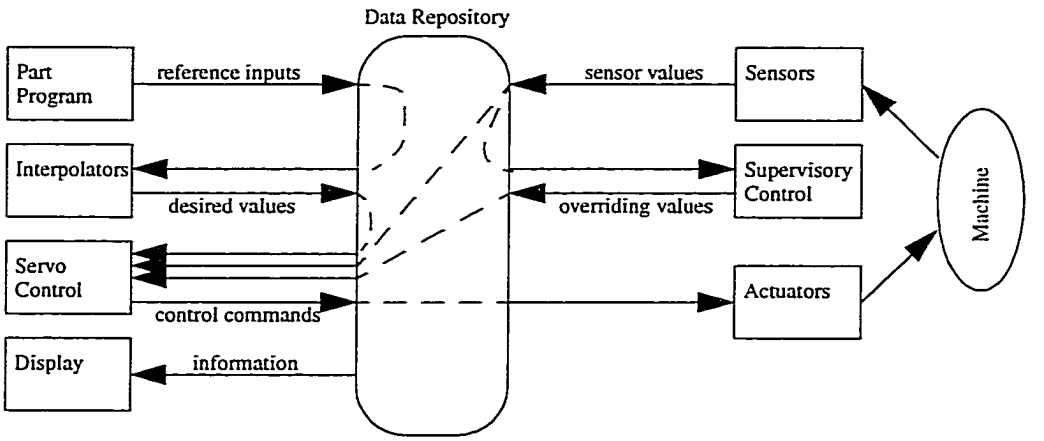


Figure 38: Open-architecture controller model.

5.3.1 UMOAC Testbed

Our prototype controllers are developed in the *University of Michigan Open-Architecture Controller* (UMOAC) testbed for a milling machine, as shown in Figure 39. VMEbus [121] is used to connect control computers (CPU-1 and CPU-2) and I/O cards, while Ethernet provides connection to the outside world. VMEbus offers up to 64-bit address and data buses, multiprocessing capability and a seven-level interrupt protocol. It can handle data transfers at a speed up to 80 Mbytes/second. VMEbus provides easy system integration and flexibility by allowing systems to be built with standard parts and minimum tooling costs.

Control tasks are executed on VMEbus-based processor boards (e.g., CPU-1 and CPU-2) running a RTOS in order to achieve good performance and timing predictability. Sensors and actuators on the milling machine are accessed through VMEbus-based IO interface boards (e.g., IO-1 & IO-2). Control software may be compiled directly on the control processors (e.g., CPU-1), or downloaded from a remote PC or workstation (e.g., PC-1 or Workstation-1). This testbed architecture allows easy adoption of new hardware

components as they become available, and thus provides good hardware openness. Well-defined interfaces and the support for performance polymorphism [180] form a foundation of software openness.

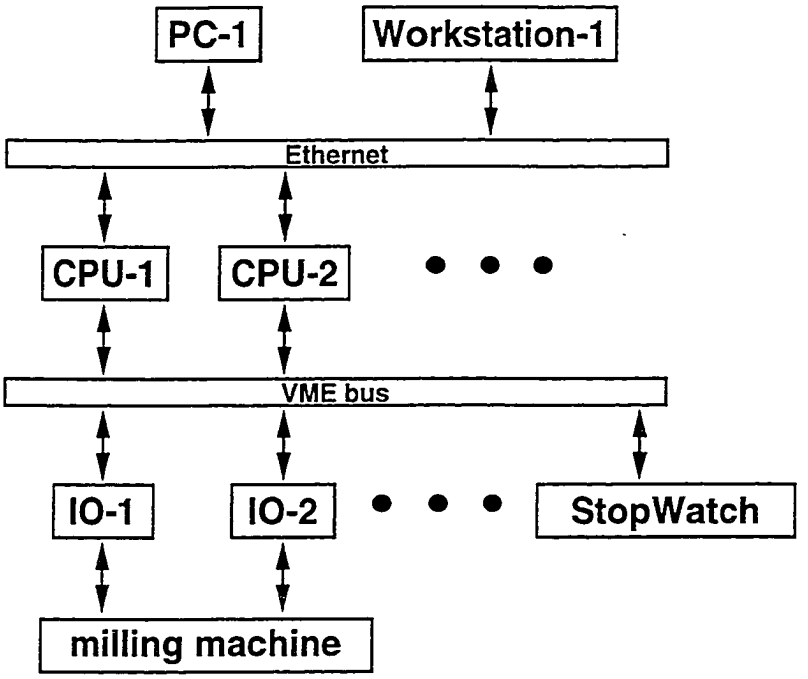


Figure 39: University of Michigan Open-Architecture Controller testbed.

The experiments are conducted on a XYCOM XVME-675/19 VMEbus PC/AT processor module (i.e., CPU-1 in Figure 39). The processor board has a 100 MHz 486DX4, with 32 Mbytes dual-access DRAM, SVGA and IDE controllers. A commercial real-time operating system QNX [127] is used.

The default QNX system clock resolution is 10 milliseconds. This is too coarse because our target control applications often require tasks with a period of 1 *ms*. On the other hand, if the resolution were too fine, the CPU would spend most of its time managing system resources without getting much useful work done. Given UMOAC testbed hardware and software configurations, we found that 50 μs (or 49447 nanoseconds to be exact) is the finest resolution with which QNX can function stably. For our experiments, we choose a clock resolution of 100 μs.

We use VME StopWatch [49] for timing measurements. It is a piece of VMEbus-based hardware that can timestamp read or write events to specific VME extended addresses. Its clock resolution is 25 nanoseconds. In our experiments, a simple inline function call is used to generate the events.

5.3.2 Controller Tasks

Figure 40 shows a milling machine control application. It consists of several tasks: Human-Machine Interface (HMI), Display, Motion Controller and Force Controller (which in turn consists of two tasks: Force Acquisition and Force Supervisor). The HMI task controls the start and stop of the motion controller and displays the milling machine status (e.g., positions). The Motion Controller task controls the movement of the milling machine. It reads sensors and sends control commands to the actuators on the milling machine. The Motion Controller task also makes the current machine status data available for the Display task to collect and send to the HMI task. The Force Controller reads the dynamometer (i.e., force sensor) on the milling machine and may override the feedrates of the Motion Controller.

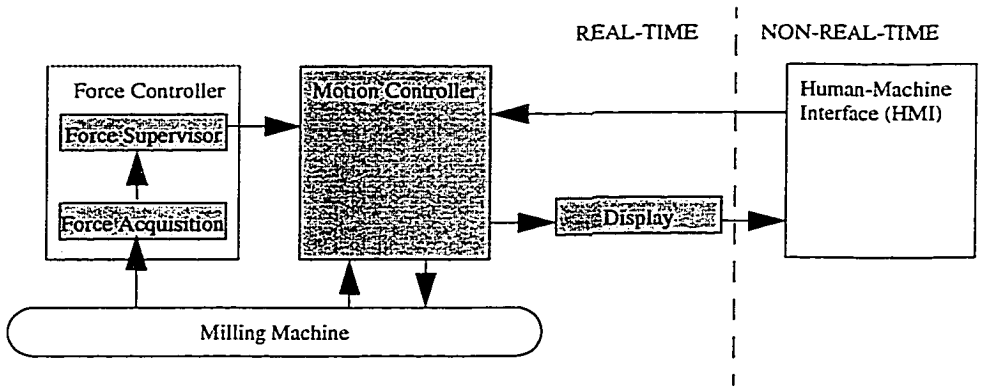


Figure 40: Milling machine control application.

Since the HMI task is not a real-time task, it is not run on the same processor with other real-time control tasks. It communicates with other tasks via nonblocking message queues. Therefore, we will only consider real-time control tasks (4 shaded boxes in

Figure 40) in our experiments. In our prototype controllers, the Force Acquisition, Force Supervisor, Motion Controller and Display tasks have periods of 1 *ms*, 40 *ms*, 10 *ms* and 40 *ms*, respectively.

Our prototype controllers are designed to have progressively more functionality and complexity. The simplest prototype controller consists of only two control tasks: a single-axis Motion Controller (X Servo task) and the Display task. The 2- and 3-axis controllers have 2- and 3-axis Motion Controllers (XY Servo and XYZ Servo tasks), respectively, as well as the Display task. The final prototype is a 3-axis controller with supervisory force control. Table 27 lists all prototype controllers and their constituent control tasks.

Prototype Controller	Control Task
1-axis motion control	X Servo, Display
2-axis motion control	XY Servo, Display
3-axis motion control	XYZ Servo, Display
3-axis motion control with force control	XYZ Servo, Display, Force Acquisition, Force Supervisor

Table 27: Prototype controllers and their respective tasks.

5.3.3 Tasks with constant nominal execution times

Determining system overhead is one of the crucial steps in establishing the run-time model of MBST. The accuracy of system overhead parameters has direct impact on the reliability of the overall simulation results of MBST. However, it is difficult to set and evaluate the accuracy of these parameters using the measurement data of real controller tasks for the following reasons. First, controller tasks are complex. They may have interaction among them, e.g., contention for the VMEbus. Determining such interaction is another crucial step in establishing the run-time model. Task execution times are an unknown function of both system overhead and task interaction. Trying to determine both appropriately at the same time can be extremely difficult, if not impossible.

Second, the nominal execution time of a real controller task may vary from one invocation to another. Some task components may run less frequently than others. For example, a servo motion control task may only need to read the reference position and velocity every 7 control loop periods. The task can interpolate the reference values into a series of desired values internally. The execution time of a function in the task may also be input data dependent. Because of such *nominal* execution time variations, the direct effects of adjusting system overhead parameters may not be easily separated from other factors.

Because of these reasons, we want to simplify the problem by dealing with independent tasks with *constant* nominal execution times. The objective is to isolate the effects of system overhead and determine proper system overhead parameters for the run-time model. Note that while the *nominal* task execution times are constant, the observed or measured ones are not. Therefore, the use of such simplified tasks is nontrivial. By choosing appropriate specifications for these simplified tasks, we can ensure the system overhead parameters determined here are valid for real controller tasks as well.

The simplified tasks must be run and measured in the same computer system as the real controller tasks. To generate similar load and operating conditions, we use one simplified task as a "proxy" for each real controller task. The simplified task has the same period as the real one and its constant nominal execution time is close to the *average* execution time of the real task. Furthermore, the simplified tasks are scheduled with RM, as in the real controllers.

The following sections describe the step-by-step application of MBST to the simplified tasks.

5.3.3.1 Establishing task model

For these tasks with constant nominal execution times, the task model is trivial. Each task consists of only one task component. Each task is run on the control processor (CPU-1 in

the UMOAC testbed of Figure 39) in isolation and its execution time is measured. Table 28 lists the statistics of the measurement data of overall task execution times of individual tasks.

task	mean (μs)	standard deviation (μs)	min (μs)	max (μs)	sample size	period (*ms*)
Display	44.0	1.88	41.1	90.2	4096	40
X Servo	312	2.31	310	363	4096	10
XY Servo	477	8.15	469	535	4096	10
XYZ Servo	722	15.7	709	819	4096	10
Force Acquisition	80.6	7.13	74.6	127	4096	1
Force Supervisor	631	2.78	627	695	4096	40

Table 28: Statistics of execution times of simplified tasks.

In MBST, the measured execution time distributions of individual task components running in isolation are used as inputs for the simulations of multiple tasks running together with possible interaction among them. To ensure the validity of the simulations, we first need to verify that the *isolated* task execution time distributions generated by MBST simulation software are consistent with the measurement data. Figures 41, 42, 43, 44, 45 and 46 compare the measured and simulated task execution time distributions for Display, X Servo, XY Servo, XYZ Servo, Force Acquisition and Force Supervisor running in isolation, respectively. For example, the top graph in Figure 41 shows the probability mass functions of the measured and simulated Display task execution times, while the bottom graph shows the corresponding cumulative distribution functions. These figures show that the MBST-generated distributions match the measurement data very well. Other figures show the similar results and they are included here for completeness.

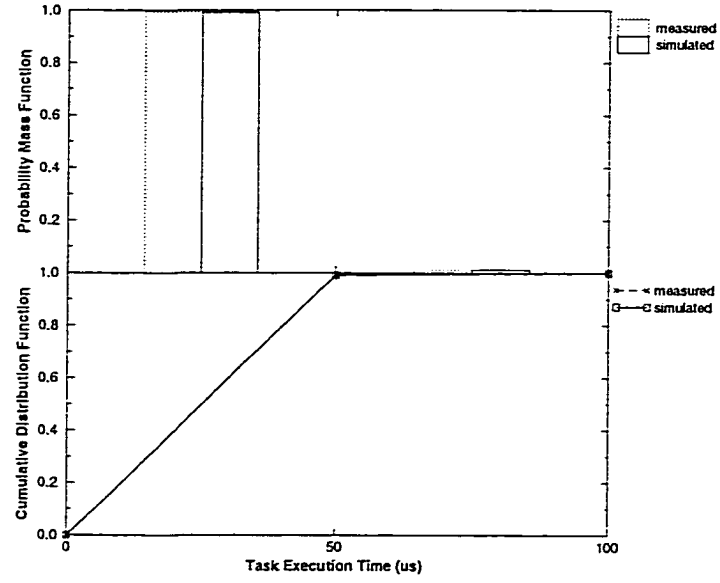


Figure 41: Execution time of simplified Display task running in isolation.

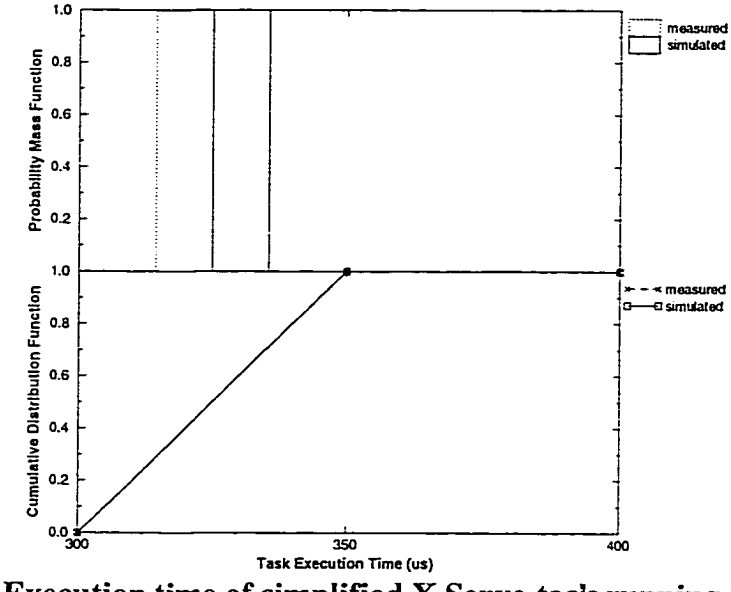


Figure 42: Execution time of simplified X Servo task running in isolation.

5.3.3.2 Establishing run-time model

The first step of establishing the run-time model is to select a scheduling algorithm. Based on the evaluation results in Section 4.4.1, the RM scheduling algorithm is chosen for our prototype controllers. Therefore, RM is also used for the simplified tasks. Because these tasks are independent, there is no interaction among them except for the contention for

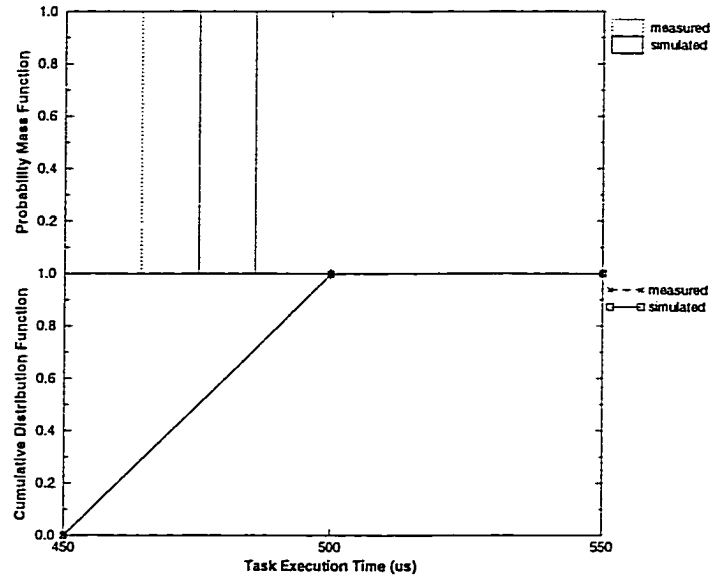


Figure 43: Execution time of simplified XY Servo task running in isolation.

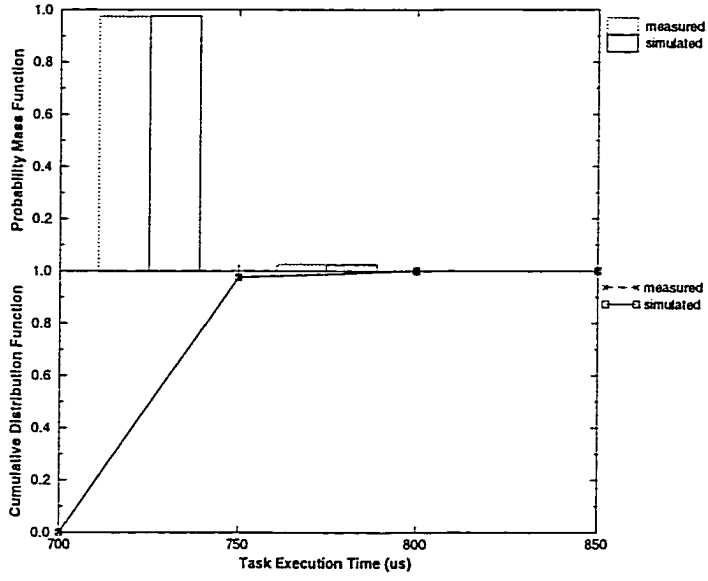


Figure 44: Execution time of simplified XYZ Servo task running in isolation.

CPU cycles. Setting system overhead parameters is the only step left in establishing the run-time model.

There are two major overheads associated with running multiple periodic tasks. The first is the timer overhead. Each periodic task uses a POSIX timer to run periodically. After each invocation of the task, it goes into sleep (i.e., the task process is in the suspended mode) until it is waken up by a signal generated by the timer. The timer

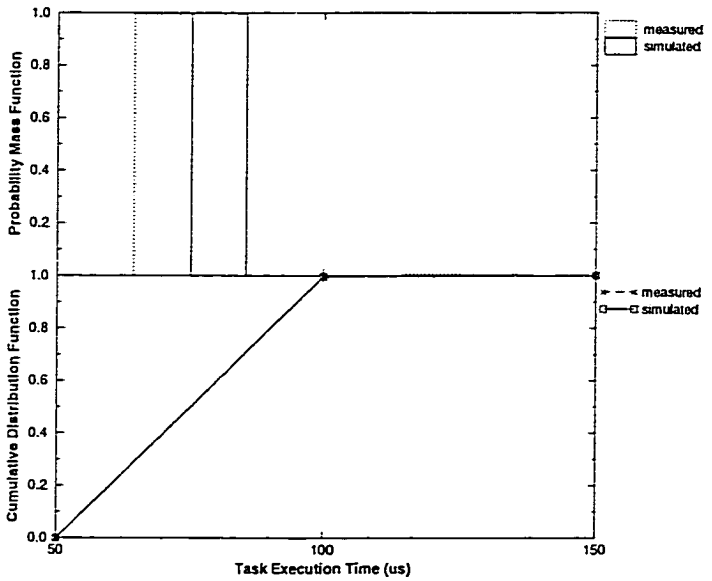


Figure 45: Execution time of simplified Force Acquisition task running in isolation.

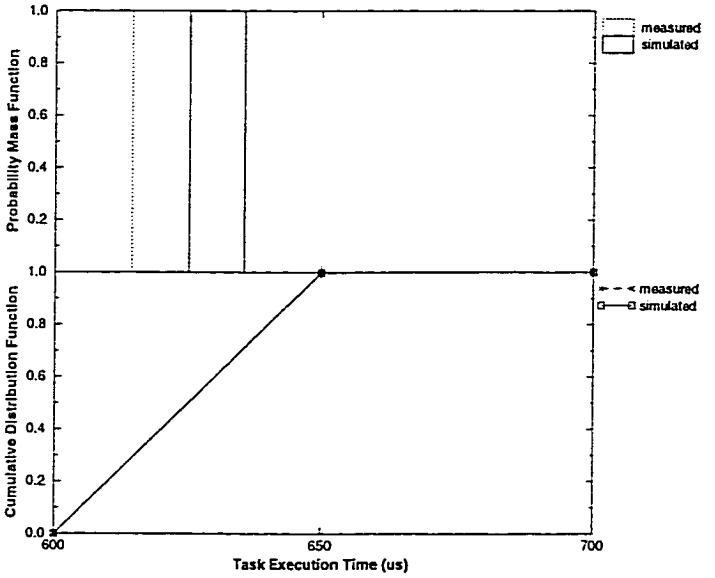


Figure 46: Execution time of simplified Force Supervisor task running in isolation.

overhead includes the CPU time it takes for the timer to signal the suspended task, for the task to change its status from suspended to ready, and for the timer to re-arm itself.

Table 29 lists the statistics of our measurement data for the timer overhead. We will use the minimum measured value of 11 μs as the nominal timer overhead in our simulations.

overhead	mean (μs)	standard deviation (μs)	min (μs)	max (μs)	sample size
timer	15	8.2	11	45	1997

Table 29: Statistics of timer overhead.

Another major overhead is the context switching overhead—the CPU time it takes to save the state of the task process being preempted and restore the state of the task process to be run. For an Intel 486DX4 processor, this overhead is about 8 μs [127].

5.3.3.3 Establishing simulation model

Four different simulation configurations are used: best, random start, same start and worst. In the case of best and worst configurations, the task phasings are manually adjusted such that the tasks in each task set will have shortest and longest execution times, respectively. Simulations of the simplified tasks are then run for the 1-, 2-, and 3-axis motion controllers and the 3-axis controller with force control.

5.3.3.4 Validating MBST

If all parameters of MBST are set appropriately, the resulting task execution time distributions from the best and worst simulation configurations should bound the actual measured data. The task execution time distributions from the random start and same start simulation configurations should be between the best and worst as well, with random start closer to the best and same start closer to the worst.

Figures 47, 48 and 49 compare the measurement and simulation results of the X, XY and XYZ Servo tasks in the 1-, 2-, and 3-axis motion controllers, respectively, where the only other task Display has a lower priority. Since the servo tasks are the highest priority task, the results of 4 simulation configurations should be the same, which is indeed the case.

For the X and XY Servo tasks, the measurement and simulation results are almost identical. For the XYZ Servo task (Figure 49), the simulation results are slightly better

than the measurement. 92.3% measured task execution times lie between 700 μs and 750 μs, while it is about 97.5% from simulations.

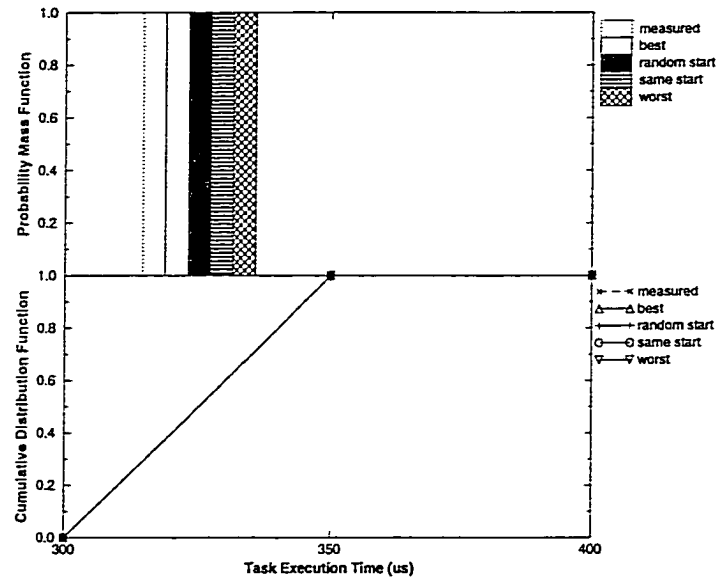


Figure 47: Execution time of simplified X Servo task running with Display.

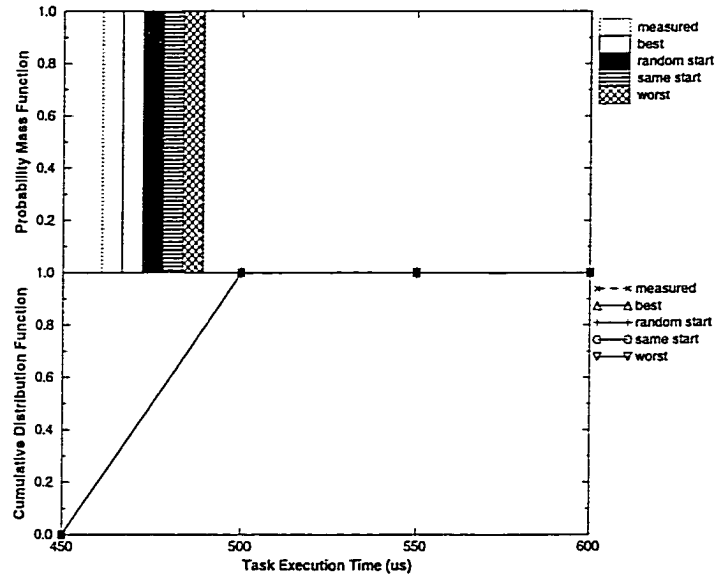


Figure 48: Execution time of simplified XY Servo task running with Display.

Figures 50, 51 and 52 compare the measurement and simulation results of the Display task in the 1-, 2-, and 3-axis motion controllers, respectively. The Display task has a lower priority and can be preempted by the motion control tasks. Since the CPU utilization is low in all three controllers (0.03, 0.05 and 0.07, respectively), there is little

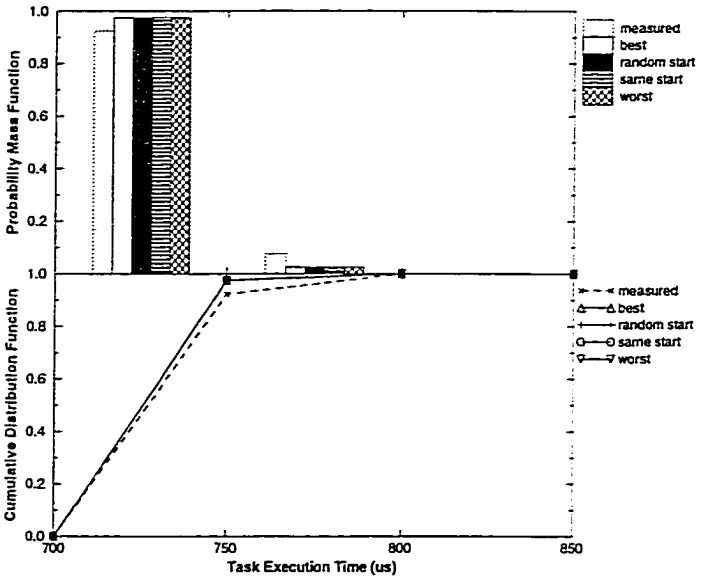


Figure 49: Execution time of simplified XYZ Servo task running with Display.

contention for the CPU cycles. Therefore, the measured task execution time distributions are almost identical to the simulation results from the best and random start configurations. The simulation results from the same start configuration are between those from the best and worst configurations, as expected.

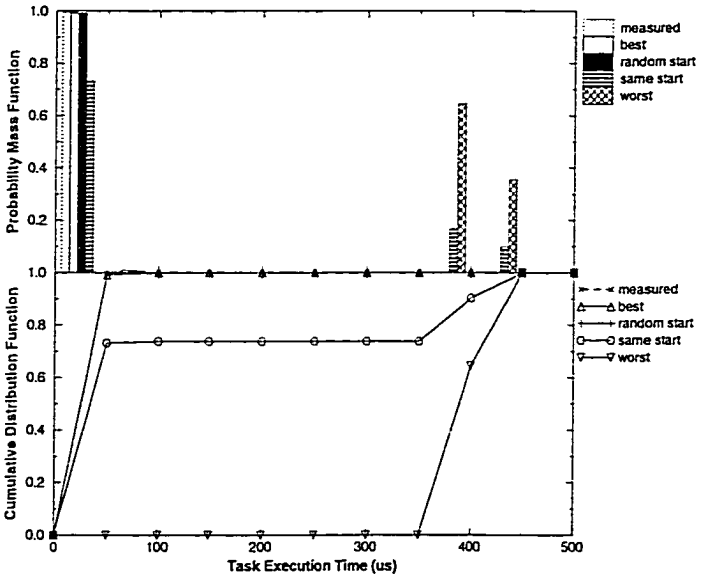


Figure 50: Execution time of simplified Display task running with X Servo.

For the 3-axis motion controller with force control, Figures 53, 54, 55 and 56 compare the measurement and simulation results of the Force Acquisition, XYZ Servo,

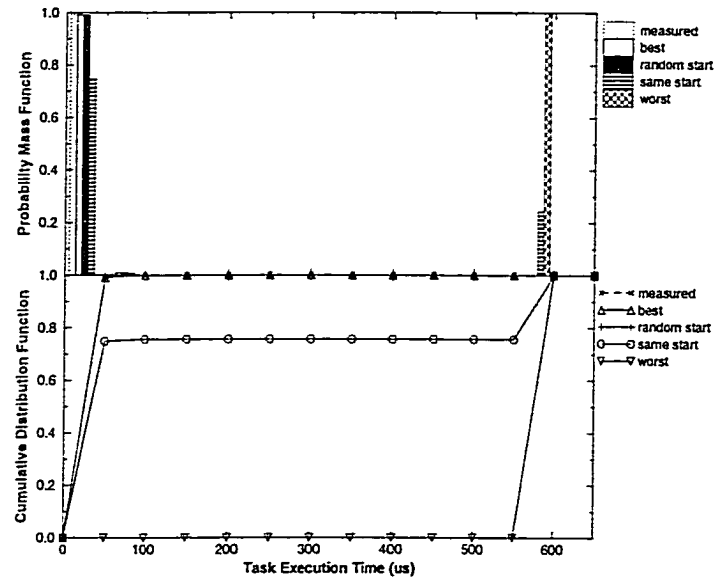


Figure 51: Execution time of simplified Display task running with XY Servo.

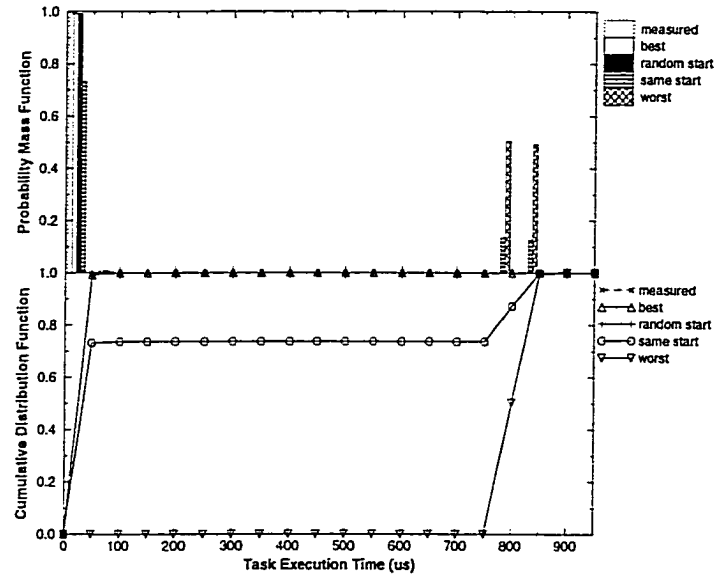


Figure 52: Execution time of simplified Display task running with XYZ Servo.

Force Supervisor and Display tasks, respectively. Using RM, the Force Acquisition task is assigned the highest priority and XYZ Servo a lower priority. Force Supervisor and Display share the same lowest priority.

All execution time distributions of the Force Acquisition task are almost identical, as it is not preempted by any other controller tasks. In Figures 54 and 55, 56, The simulation results from the best, random start, same start and worst configurations clearly range from the best (the leftmost curve of the cumulative distribution function) to the

worst (the rightmost curve), in that order. The measured XYZ Servo task execution time distribution is very close to that of the worst simulation configuration (Figure 54), because it is likely to be preempted by the Force Acquisition task. While the measured Force Supervisor distribution is similar to that of the random start configuration (Figure 55), the results of the Display task (Figure 56) are similar to those of Figures 50, 51 and 52.

In all the above experiments, the measured task execution time distributions are bounded by the simulation results from the best and worst task phasing configurations. While in some cases, the measured task execution time distributions are very close to those of the best simulation configuration, in some other cases, they are very close to those of the worst simulation configuration. This indicates that our models are accurate and provide tight bounds for the actual task performance. Therefore, MBST is validated and we will use the system overhead parameters set here for our experiments with real controller tasks.

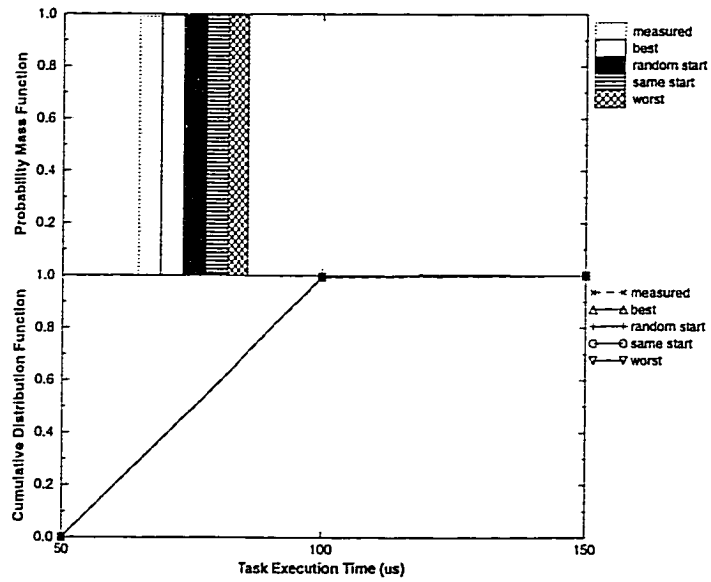


Figure 53: Execution time of simplified Force Acquisition task running with XYZ Servo, Force Supervisor and Display.

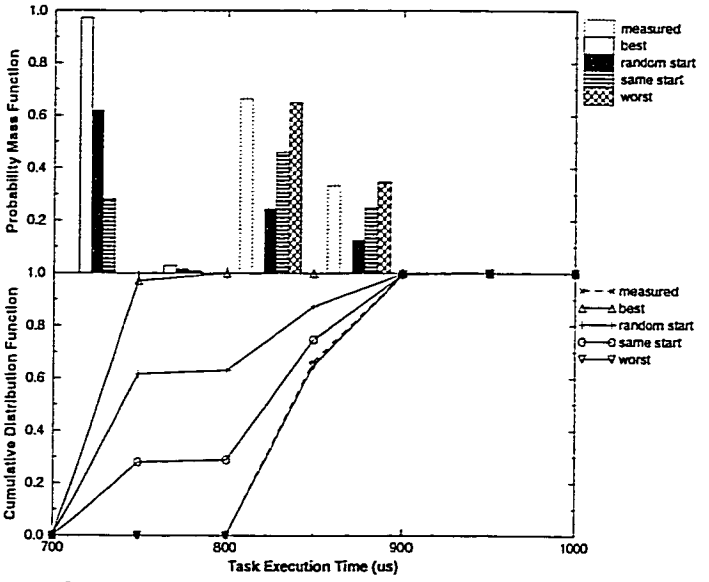


Figure 54: Execution time of simplified XYZ Servo task running with Force Acquisition, Force Supervisor and Display.

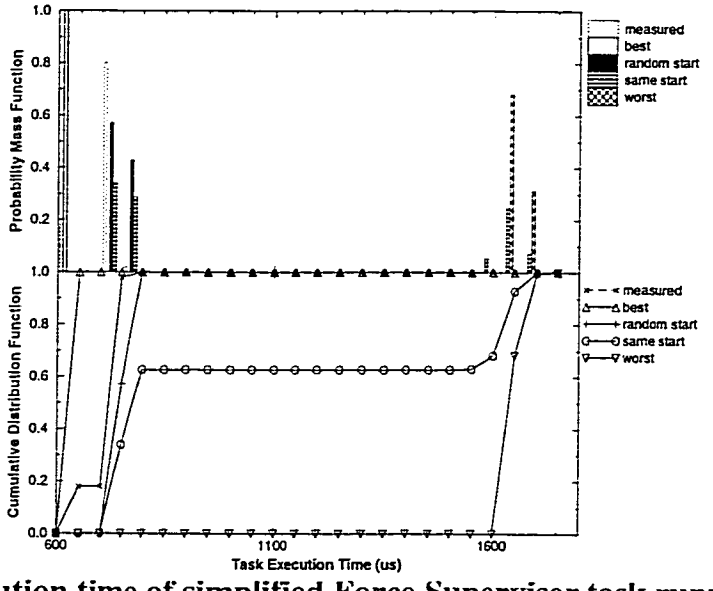


Figure 55: Execution time of simplified Force Supervisor task running with Force Acquisition, XYZ Servo and Display.

5.3.4 Machine tool controller tasks

Having determined system overhead parameters in our experiments with simplified tasks, we now apply MBST to real controller tasks.

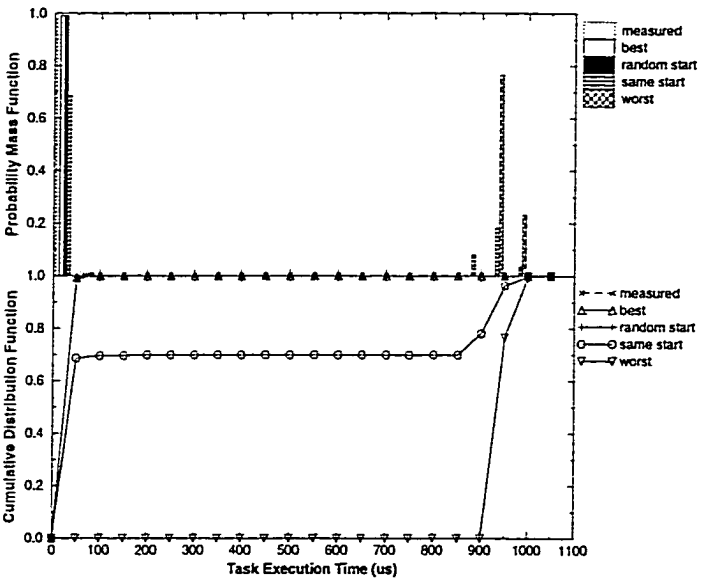


Figure 56: Execution time of simplified Display task running with Force Acquisition, XYZ Servo and Force Supervisor.

5.3.4.1 Establishing task model

Open-architecture machine tool controller tasks are typically periodic. During each period, a control task may get data, read sensors, process data, update actuator signals and send data, as illustrated in Figure 57. These 5 logical task components are not necessarily present in all controller tasks. Furthermore, the execution frequencies of task components may be different, though they are typically integral multiples of the task period.

For example, a servo motion control task may consist of all 5 components. Its "get data" component may involve reading reference positions from message queues once every 7 task periods and running interpolators to obtain desired positions. It reads the encoders and tachometers on the controlled machine to get current positions and velocities. It then processes the above information, as well as any overriding values the supervisory controllers may provide. The resulting actuator signals are sent to the actuators on the milling machine. Finally, it may send out contour errors. On the other hand, a display task may have only two components—"get data" from the data repository and "send data" to the HMI task on a different computer.

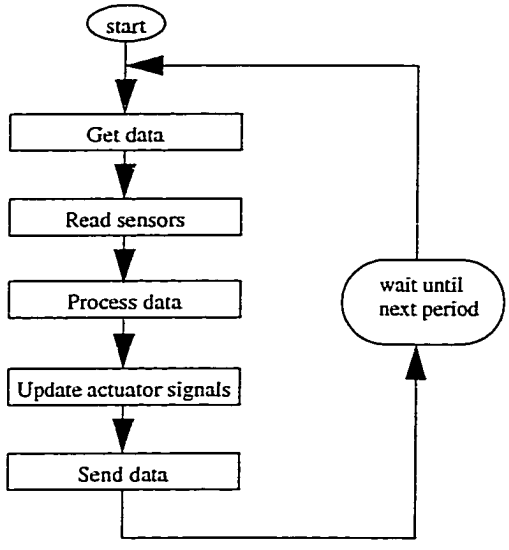


Figure 57: Typical components of an open-architecture controller task.

Table 30 lists the task components of our prototype controller tasks. Each logical task component may in turn consist of several physical task components. For example, the "get data" component of a servo task includes a communication component that reads the message queues to obtain reference values and an interpolator component to generate a series of desired values. The execution times of all physical components are measured when individual tasks are running in isolation.

The overall task execution times of individual isolated tasks are generated using measured task component execution time distributions with the "synchronized combination" approach. Figures 58, 59, 60, 61, 62 and 63 show the simulated and measured execution time distributions for the controller tasks running in isolation. The simulation results match the actual measurements very well.

task	get data	read sensors	process data	update actuator signals	send data
Display	get current positions	N/A	N/A	N/A	send batch positions
X Servo	get 1-axis reference values and interpolate	read 1-axis encoder and tachometer	compute 1-axis actuator signal	send signal to 1-axis actuator	send contour errors
XY Servo	get 2-axis reference values and interpolate	read 2-axis encoders and tachometers	compute 2-axis actuator signals	send signals to 2-axis actuators	send contour errors
XYZ Servo	get 3-axis reference values and interpolate	read 3-axis encoders and tachometers	compute 3-axis actuator signals	send signals to 3-axis actuators	send contour errors
Force Acquisition	N/A	read dynamometer	find maximum in a window	N/A	send maximum force value
Force Supervisor	get maximum force value	N/A	compute overriding feedrates	N/A	send overriding feedrates

Table 30: Logical components of prototype controller tasks.

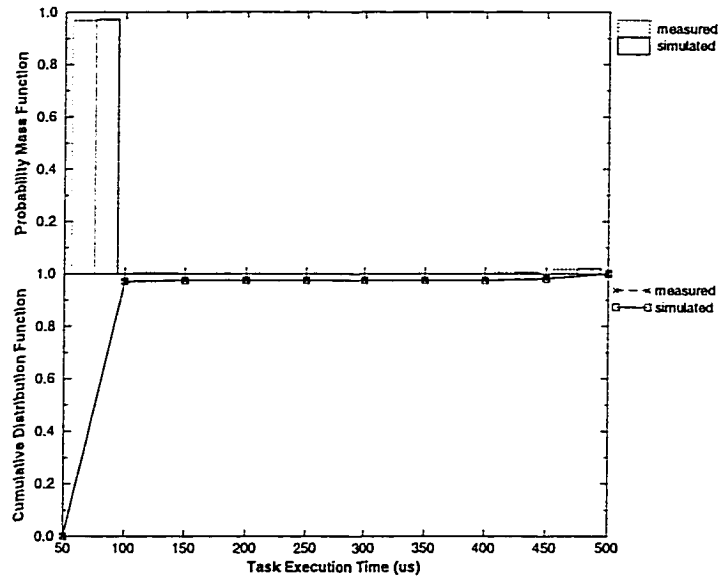


Figure 58: Simulated and measured task execution time of Force Acquisition.

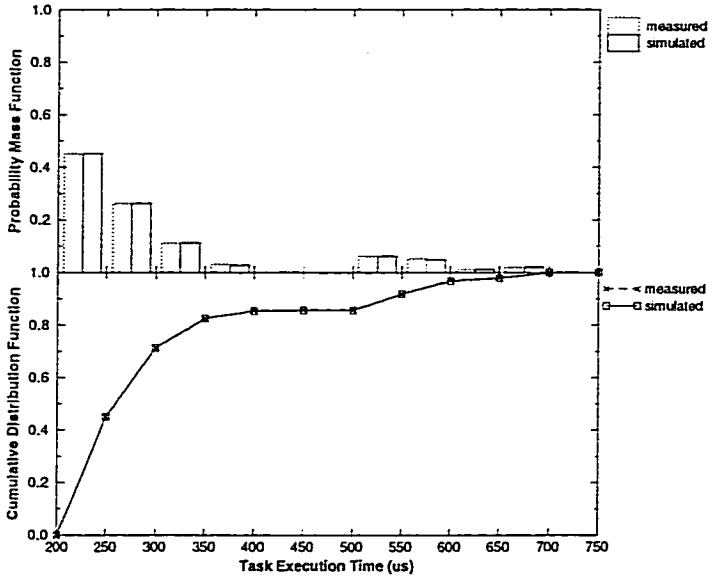


Figure 59: Simulated and measured task execution time of X Servo.

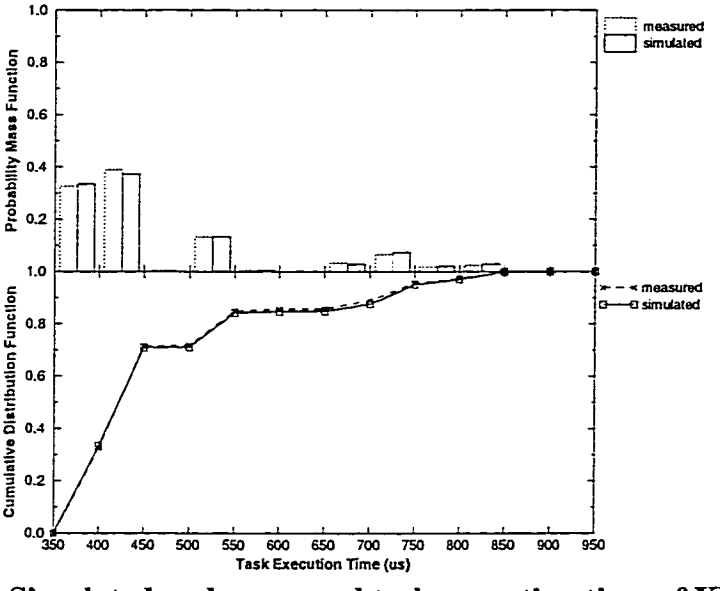


Figure 60: Simulated and measured task execution time of XY Servo.

5.3.4.2 Establishing run-time model

Our prototype open-architecture machine tool controllers use the RM scheduling algorithm.[2] We will use the system overhead parameters in Section 5.3.3, since they were derived for the same system environment.

2. For other scheduling algorithms, such as those evaluated in Chapter 4, the steps of applying MBST will be the same but the run-time models will be different.

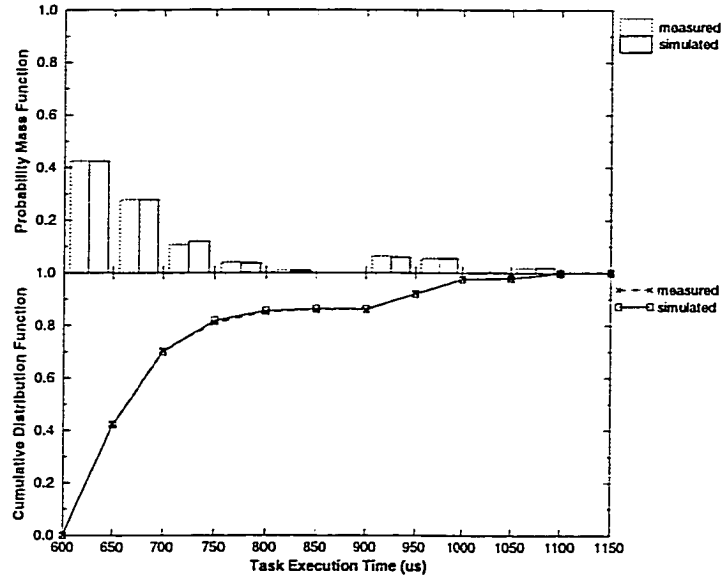


Figure 61: Simulated and measured task execution time of XYZ Servo.

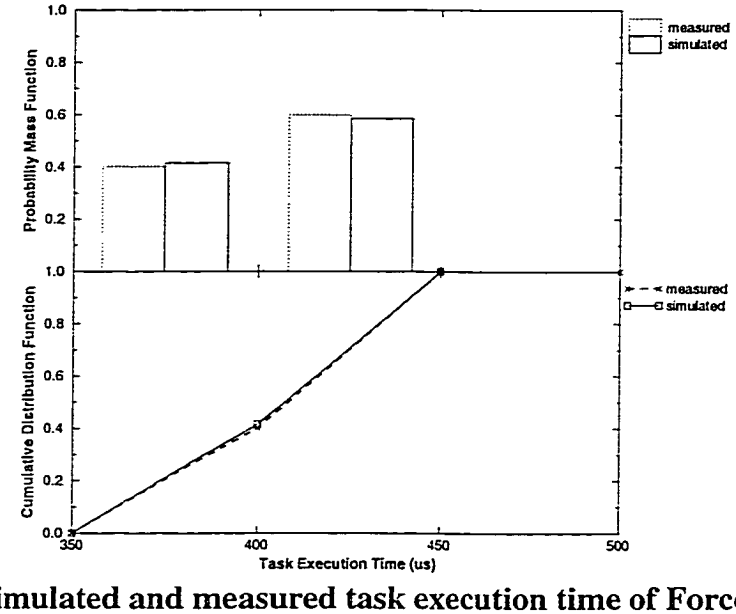


Figure 62: Simulated and measured task execution time of Force Supervisor.

Though there are inter-process communications between controller tasks, they use non-blocking mechanisms such as message queues. Like independent tasks, controller tasks compete for CPU cycles if they are on the same processor and are subjected to preemptions based on their priorities. However, unlike independent tasks, some controller tasks need to access sensors and actuators via I/O cards on the shared VMEbus (see Figure 39). VMEbus uses a master-slave architecture. Functional modules called masters

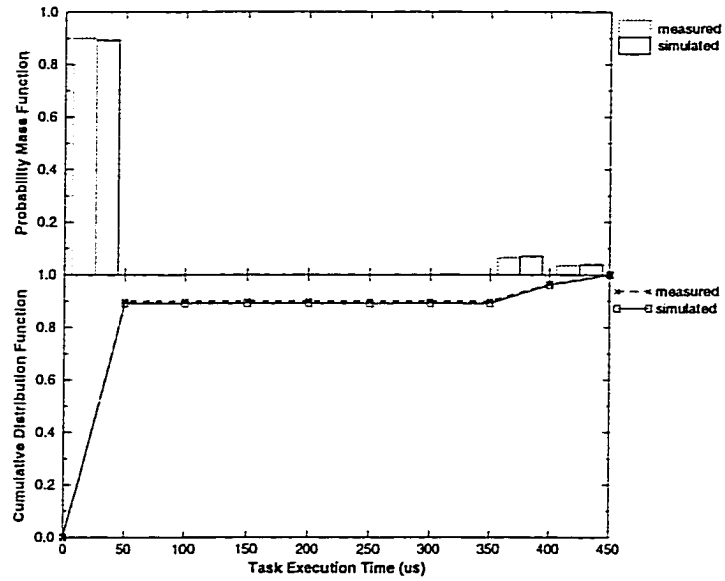


Figure 63: Simulated and measured task execution time of Display.

(e.g., CPU) transfer data to and from modules called slaves (e.g., I/O cards). During a typical read/write cycle, the master acquires the bus, addresses a slave, and then transfers data. Therefore, a higher priority task needing to access the VMEbus may have to wait for any bus read/write cycle in progress by a lower priority task. Note that this is not a case of priority inversion, because the blocking of the higher priority task is at most one VMEbus cycle. For a lower priority task with VMEbus access in progress, it may have to redo the bus cycle if it is preempted by a higher priority task. These possibilities will be considered in our simulation model.

5.3.4.3 Establishing simulation model

Again, four different simulation configurations are used: best, random start, same start and worst. In the case of best and worst configurations, the task phasings are manually adjusted such that the tasks in each task sets will have shortest and longest execution times, respectively. In addition, in the worst configuration, the contention for the VMEbus by tasks with bus access needs are taken into account. Simulations of the controller tasks are then run for the 1-, 2-, and 3-axis motion controllers and the 3-axis controller with force control and compared with measurement results.

5.3.4.4 Validating MBST

Figures 64, 65 and 66 compare the measurement and simulation results of the real X, XY and XYZ Servo tasks in the prototype 1-, 2-, and 3-axis motion controllers, respectively, where the only other task is Display with a lower priority. In all cases, the results from different simulation configurations are almost identical and match the measurement data very well.

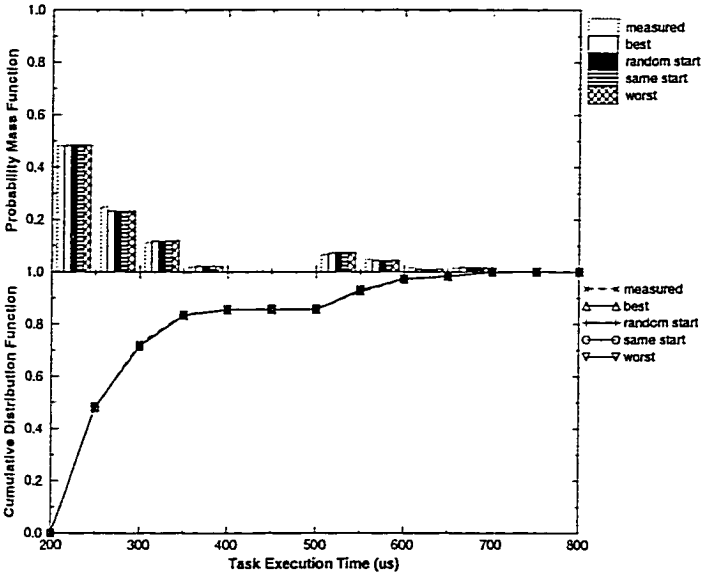


Figure 64: Execution time of X Servo task running with Display.

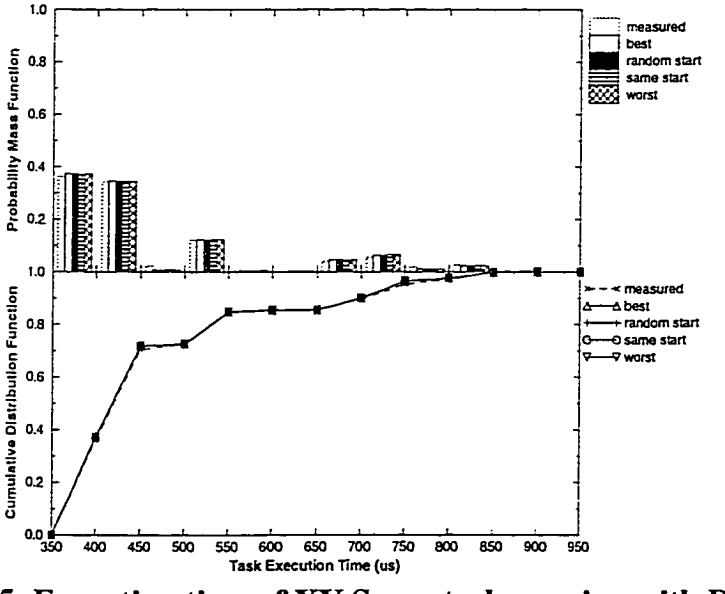


Figure 65: Execution time of XY Servo task running with Display.

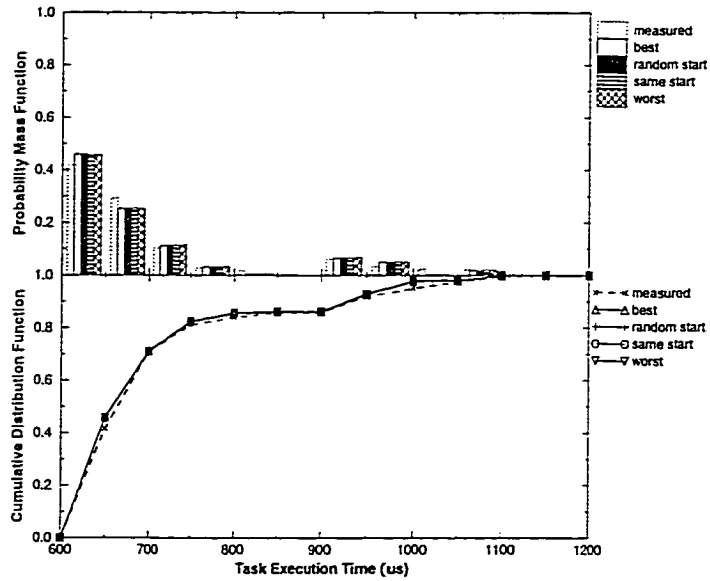


Figure 66: Execution time of XYZ Servo task running with Display.

The results of the Display task in these 3 prototype controllers are shown in Figures 67, 68 and 69, respectively. They are very similar to each other and to those of the simplified Display task (Figures 50, 51 and 52). Because of low CPU utilizations, the Display task has a very small probability of being preempted by the servo tasks. Therefore, the measured execution time distributions are very close to those from the best simulation configuration. The results from the random start and same start configurations are between the best and the worst. Again, because interval variations and low CPU utilizations, the distributions from the best and random start configurations are almost identical.

Finally, we examine simulation and measurement results of the 3-axis controller with supervisory force control. Figures 70, 71, 72 and 73 show the execution time distributions of the tasks Force Acquisition, XYZ Servo, Force Supervisor and Display, respectively.

For the Force Acquisition task (Figure 70), the results of the best, random start and same start simulation configurations are almost identical, as expected. These configurations do not consider the VMEbus contention. On the other hand, the worst simulation configuration assumes the longest possible blocking due to bus contention.

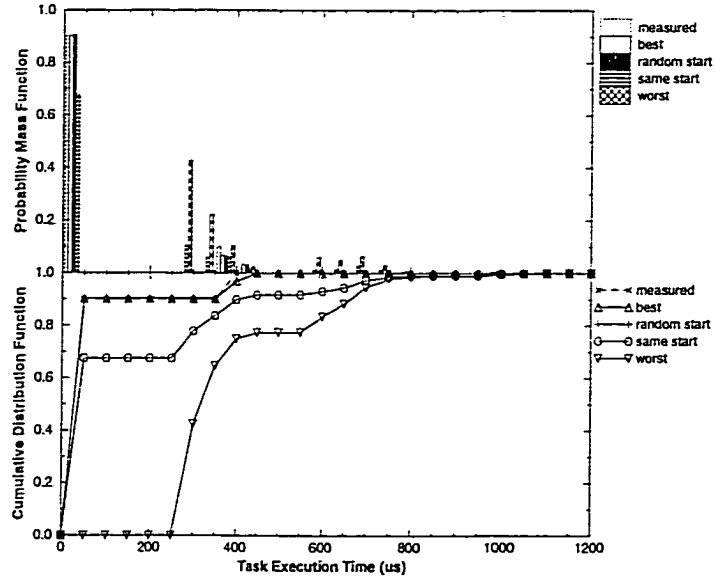


Figure 67: Execution time of Display task running with X Servo.

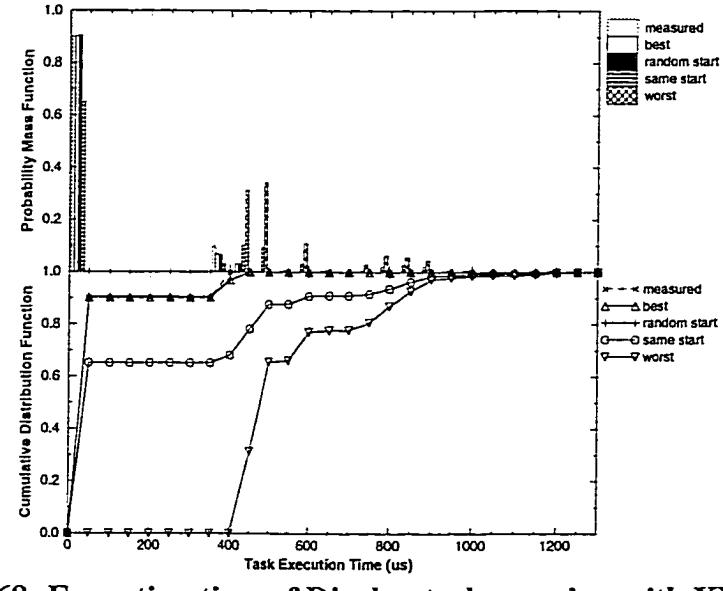


Figure 68: Execution time of Display task running with XY Servo.

Figure 70 shows that, in the task execution time range between 50 μs and 250 μs, the worst configuration produces a cumulative probability function that is about 10% lower than those of other simulation configurations (a CDF of 0.88 versus 0.97). The actual measurement verifies the effect of VMEbus contention (a CDF of 0.89 between 50 μs and 250 μs), though the delay is not as long as the worst case.

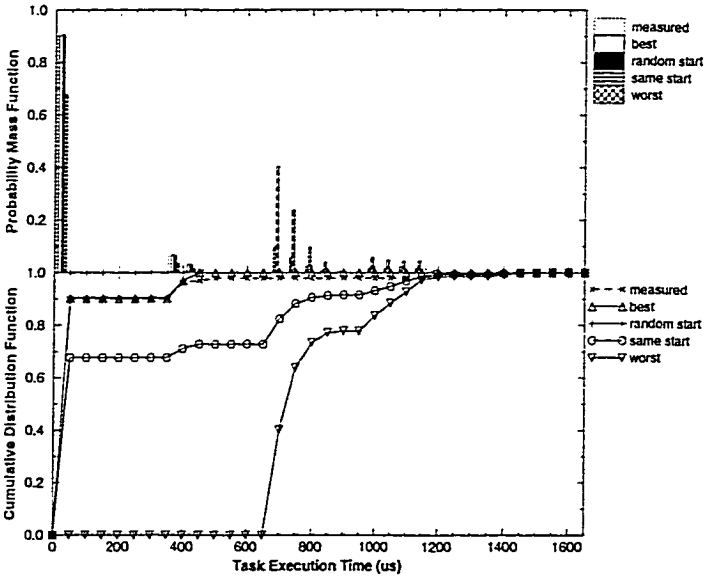


Figure 69: Execution time of Display task running with XYZ Servo.

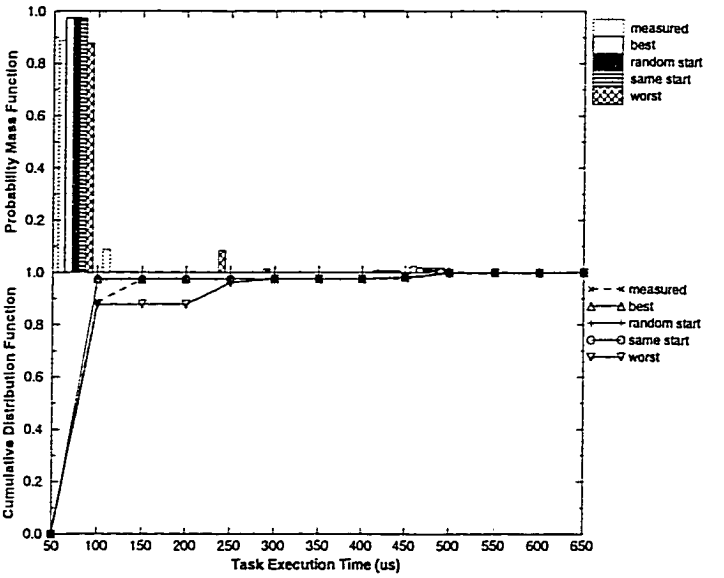


Figure 70: Execution time of Force Acquisition task running with XYZ Servo, Force Supervisor and Display.

For tasks XYZ Servo, Force Supervisor and Display, respectively, Figures 71, 72 and 73 clearly show the effects of different phasings because these tasks could be preempted. In general, the task execution time distributions generated by the best, random start, same start and worst simulation configurations are progressively worse (i.e., longer execution time), in that order. Occasionally, the distributions of the random start

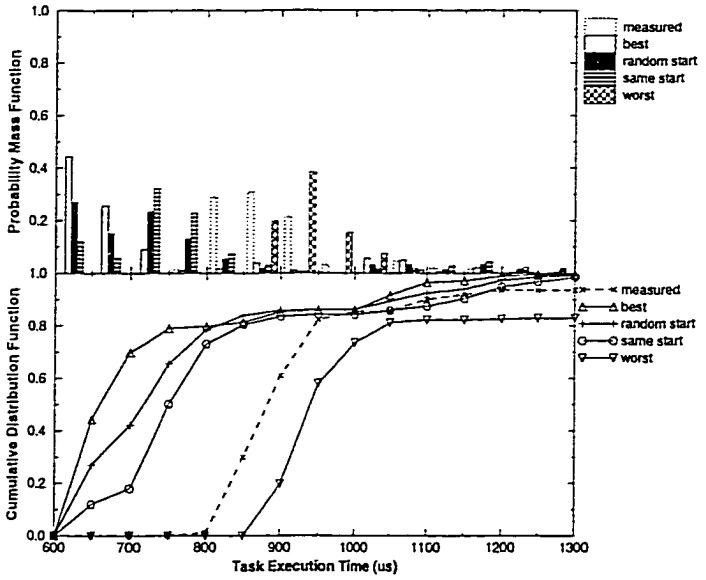


Figure 71: Execution time of XYZ Servo task running with Force Acquisition, Force Supervisor and Display.

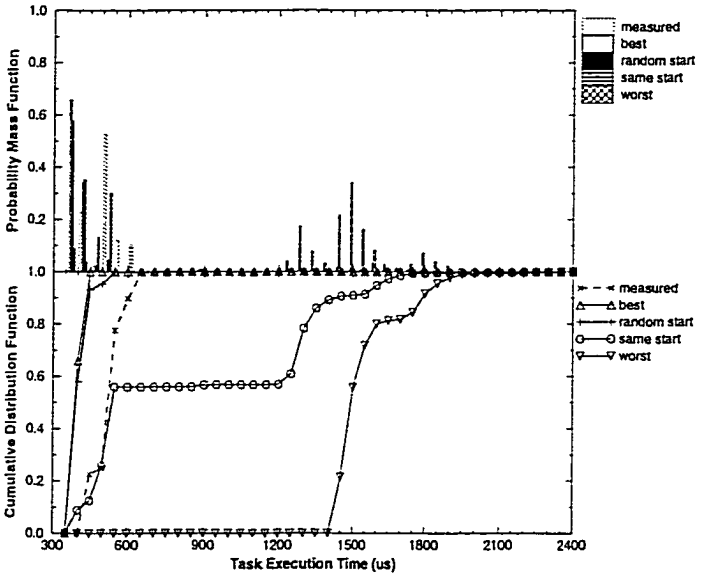


Figure 72: Execution time of Force Supervisor task running with Force Acquisition, XYZ Servo and Display.

configuration can be slightly better than those of the best. Figure 71 shows such a case. The CDF of the XYZ Servo task execution time at 850 μs is about 0.81 by the best configuration and about 0.84 by the random start. The reason for this phenomenon is that there are variations in task intervals and execution times. The best configuration uses constant intervals and actual (varying) execution times, while the random start

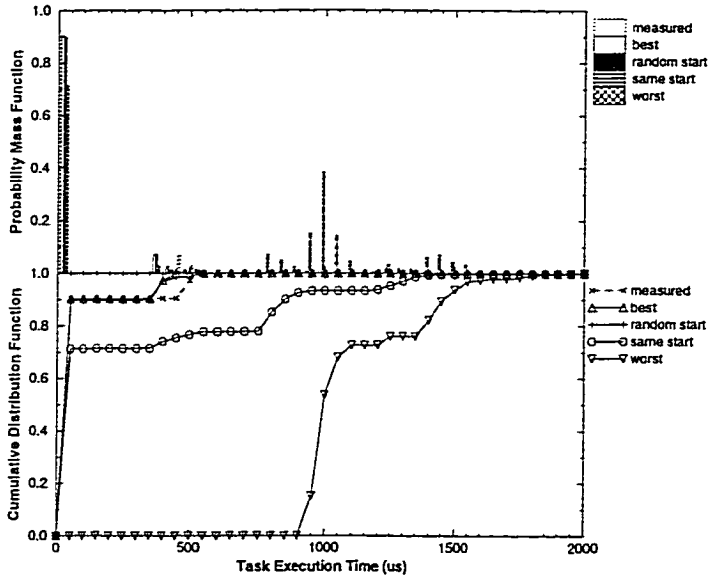


Figure 73: Execution time of Display task running with Force Acquisition, XYZ Servo and Force Supervisor.

configuration uses both actual (varying) intervals and execution times. Though the best configuration attempts to minimize contention by manually adjusting the task phasing, there are situations where varying intervals lead to better phasings.

For example, suppose there are two tasks whose periods are 1 *ms* and 10 *ms* and whose execution times are 100 μs and 950 μs, respectively. For the best configuration, any phasing would cause the 10 *ms* period task to be preempted by the 1 *ms* period task, since it assumes constant intervals. However, with random start, the following scenario could occur. The first task (1 *ms* period) is released at time 0, completed at time 100 μs. The second task is released at time 100 μs. However, unlike the best configuration case, the next release of the first task could be at time 1050 μs due to interval variations, thus allowing the second task to finish without being preempted.

Even though the best simulation configuration may not be truly "best" because of the variations in task intervals and execution times, the effects of those special situations as illustrated above are very small. In Figure 71, except for that point where the CDF of the best configuration is smaller than that of the random start by less than 5% (0.81 versus

0.84), the CDF of the best configuration is consistently equal to or larger than that of the random start.

In all cases, the measured execution time distributions of controller tasks are bounded by those generated by the best and worst simulation configurations. Therefore, MBST proved to be effective in modeling and predicting the performance of open-architecture controller tasks.

5.3.4.5 Providing probabilistic deadline guarantees

Since we have validated the simulation results of MBST with measured task execution times, we can now make probabilistic deadline guarantee decisions based on the task completion time distributions generated by MBST.

We will use our 3-axis milling machine controller with supervisory force control as an example. Figures 74, 75, 76 and 77 show the completion time distributions of the tasks Force Acquisition, XYZ Servo, Force Supervisor and Display, respectively. Again, we use 4 different simulation configurations: best, ransom start, same start and worst. The best and worst configurations are slightly different from those used in bounding measured task execution time distributions. Here the phasings are adjusted to reflect scenarios where the task completion times are the shortest and longest, respectively. To obtain shortest completion times, task phasings are selected such that there is minimal contention among tasks. On the contrary, all tasks are released at the same time for the worst phasing, a critical instant where the task completion times are the longest.

To be on the conservative side, we will use the results from the worst simulation configuration. Table 31 lists a few completion probability requirements and the corresponding task completion times of individual tasks. For example, if the deadline of the XYZ Servo task is 2315 μs or later relative to its release time, a probabilistic guarantee for up to a completion probability of 1.0 can be provided. Note that whether a probabilistic deadline guarantee is possible depends on two requirements: deadline and completion

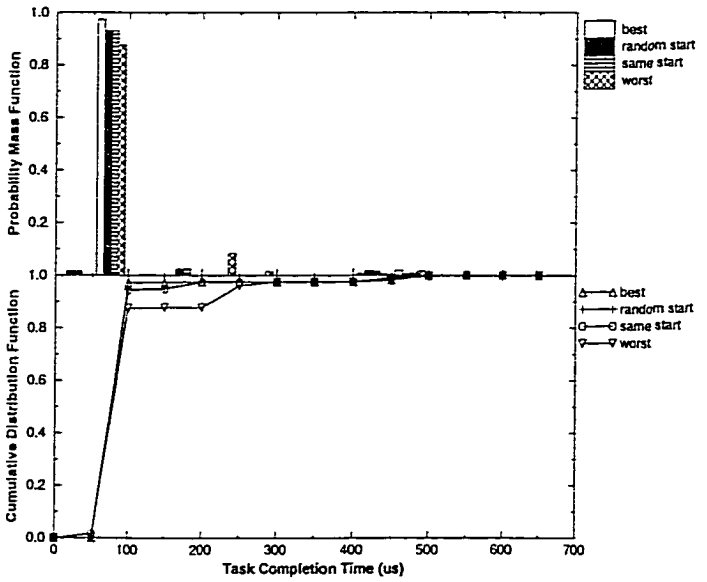


Figure 74: Completion time of Force Acquisition task running with XYZ Servo, Force Supervisor and Display.

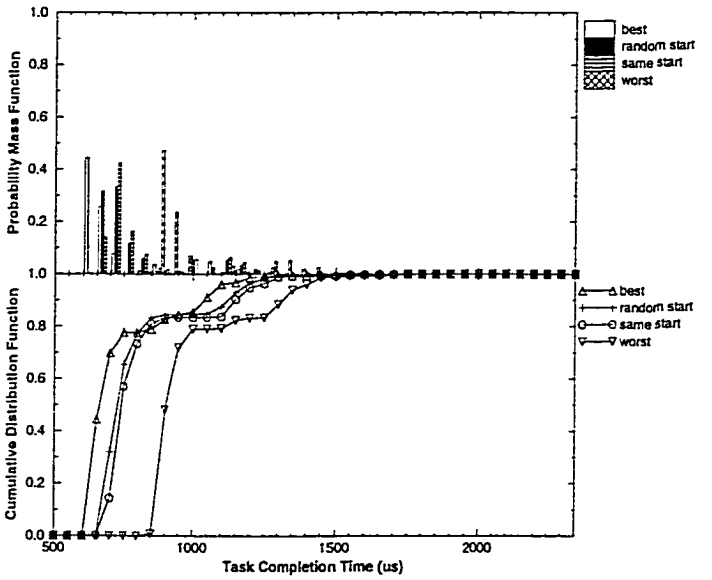


Figure 75: Completion time of XYZ Servo task running with Force Acquisition, Force Supervisor and Display.

probability. If the deadline of the XYZ Servo task is 1200 μs and its completion probability is 0.9, no probabilistic deadline guarantee can be provided based on our simulation results. However, if the completion probability is 0.8 or smaller, the probabilistic deadline guarantee can then be provided.

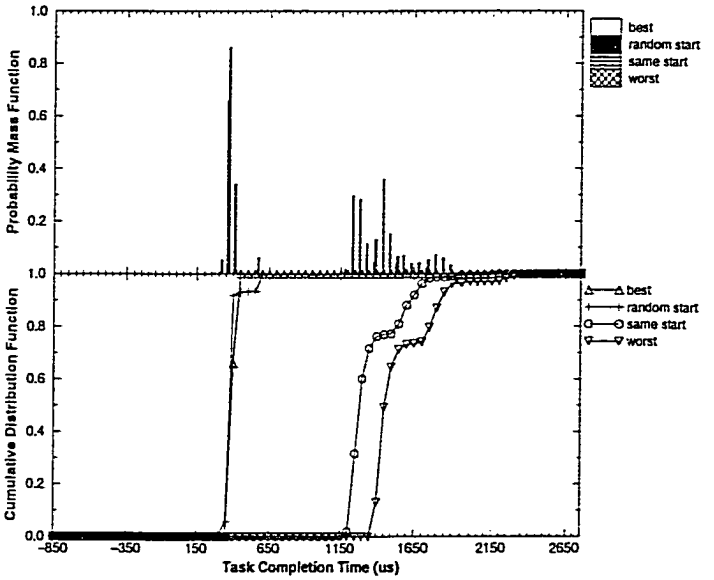


Figure 76: Completion time of Force Supervisor task running with Force Acquisition, XYZ Servo and Display.

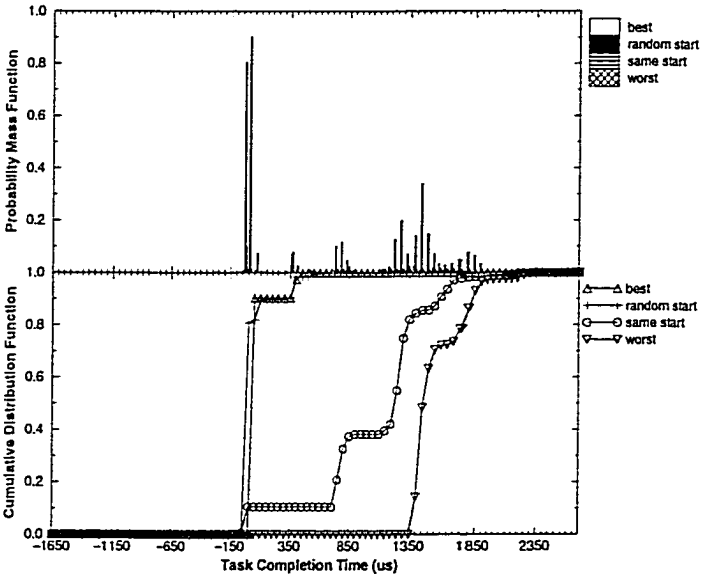


Figure 77: Completion time of Display task running with Force Acquisition, XYZ Servo and Force Supervisor.

5.3.4.6 Predicting 5-axis controller performance

Another important application of MBST is to estimate the performance of the controllers that have not yet been implemented. Suppose we expect to add a joint with 2 degrees of freedom to the milling machine. Our 3-axis controller will then need to expand its control to 5 axes. To do that, the servo motion control task requires reading the sensors and

Completion Probability	Task Completion Time (μs)			
	Force Acquisition	XYZ Servo	Force Supervisor	Display
1.0	630	2314	2318	2661
0.9	225	1316	1739	1830
0.8	77	1112	1550	1758
0.7	65	943	1470	1544
0.6	64	918	1440	1480
0.5	64	902	1421	1453

Table 31: Task completion times of the 3-axis controller with force control.

driving the actuators of 2 additional axes, as well as additional data interpolation and updates. Since we have not implemented the 5-Axis Servo task, we will use the corresponding task components in the XY and XYZ Servo tasks to approximate the additional work required in the 5-Axis Servo task. Other tasks (i.e., Force Acquisition, Force Supervisor and Display), remain pretty much the same.

Table 32 lists the performance prediction of the 5-axis milling machine controller with supervisory force control, using the worst simulation configuration. Comparing with the performance of the 3-axis counterpart in Table 31, we can see that the effect on the performance of the Force Acquisition task is minimal, because it has the highest priority. For the 5-Axis Servo task, it generally takes more time than the XYZ Servo task to achieve the same completion probability,[3] because it has more work to do now. Since the Force Supervisor and Display tasks may be preempted by both the Force Acquisition and 5-Axis Servo task, the longer-running servo task will cause a delay in the completion times of both Force Supervisor and Display, as evident in Table 32.

3. Though Table 31 indicates that to guarantee a completion probability of 1.0, the deadline of the XYZ Servo task needs to be 2315 μs or later, there are actually only two out of 4000 task completion times that are over 1900 μs by examining the simulation data. To guarantee a completion probability of 0.99, for example, a deadline of 1514 μs or later would suffice.

Completion Probability	Task Completion Time (μs)			
	Force Acquisition	5-Axis Servo	Force Supervisor	Display
1.0	642	2575	2979	3453
0.9	225	1859	2375	2481
0.8	80	1726	2260	2395
0.7	66	1631	2144	2246
0.6	64	1584	2097	2147
0.5	64	1556	1963	2109

Table 32: Predicted task completion times of 5-axis controller with force control.

5.4 Discussion and Related Work

MBST attempts to achieve a balance between assuming an idealized computing environment and assuming nothing. While modeling of non-ideal system resources has been the subject of some recent research [10, 66, 72], it remains to be a practically-difficult problem because of the complex and unpredictable nature of disturbance sources, such as interrupts. As the trend of building open systems with commercial-of-the-shelf (COTS) components continues, this modeling work becomes more difficult and less cost-effective. The use of open systems is expected to reduce system cost by leveraging commercial development and to facilitate upgrades of system components over time. Examples of such efforts include the University of Michigan Open-Architecture Controller (UMOAC) project [112] and US Navy's New Attack Submarine (NSSN) Command Control Communication and Intelligence (C3I) system [42].

However, the adoption of COTS technology means potentially frequent changes in system components. Each of these components may be engineered by different companies with different technologies. The modeling results of individual components for one system may be difficult to be reused for another system since the system constituents may be different.

While MBST does not assume an idealized computing environment, it does not attempt to model every system component either. MBST uses the measurement data for individual task components when tasks are running in isolation. Such measurement data encompass information about system unpredictability, though the information may not necessarily be able to be separated from the data. MBST requires the modeling of major system overhead and task interaction, which are more observable and understandable.

MBST improves on measurement-based approaches for hard deadline guarantees, such as MDARTS [96], by introducing probabilistic deadline guarantees. MDARTS uses the worst-case measurement data to provide hard deadline guarantees, but it has no support for probabilistic guarantees. MBST allows tasks with less stringent timing requirements to be guaranteed when they might not be guaranteed in MDARTS. For example, to guarantee the deadline of the XYZ Servo task in Table 31 with the MDARTS approach, the deadline of the task needs to be 2314 μs or later. However, by examining the data, we discover that there are only two task completion times that are over 1900 μs. To guarantee a completion probability of 0.99, the task deadline needs to be only 1513 μs or later.

5.5 Summary

We presented the last key component of our proposed practical framework for probabilistic deadline guarantees (other components are described in the previous chapter):

- *MBST*: uses measured application task component performance data as inputs, models task interaction and system overhead, and generates task completion time distributions to determine whether probabilistic deadline guarantees can be provided. Applying MBST to our prototype open-architecture milling machine controllers, MBST is shown to produce simulation results matching very well with actual measurement of real tasks. It can also be used to predict performance of tasks that have not yet been fully implemented.

CHAPTER 6

APPLICATION DEVELOPMENT

6.1 Introduction

While our studies presented in previous chapters have addressed the real-time performance issues *given* an application implementation, we have not examined how a real-time application should be implemented in order to better achieve its requirements in the presence of RTOS unpredictability, which is the focus of this chapter. We will conduct our research using prototype open-architecture modular controllers in the UMOAC testbed.

Historically, the software structure of many commercial and research machine tool controllers is monolithic [20]. There is no clear task structure in these monolithic controllers and the source code of individual tasks is interwoven. Therefore, it is extremely difficult to incorporate third-party software into such a controller. On the contrary, tasks in an open-architecture modular controller are well-defined entities. They can be compiled and linked independently. Tasks can execute as separate computer processes. Each periodic task uses a software timer of its own in order to run at its specified interval.

Due to their modularity, open-architecture controllers rely more on the services provided by the underlying RTOS, such as the software timer, scheduler and inter-process communication (IPC). In reality, no RTOS can provide ideal services. Furthermore, contemporary microprocessor architectures rely on hardware interrupts to manage system

resources, which can cause significant unpredictability in RTOS services, such as timer interval variation.

While there has been extensive work on RTOSs (e.g., [155, 157, 165]) and their modeling (e.g., [72]) or open-architecture controllers (e.g., [5, 6, 7, 20, 53, 117, 125, 158, 160, 174]), it is unknown how well real-time controllers with such a modular structure perform in practice in an open-architecture environment and what kind or magnitude of impact the underlying RTOS unpredictability has on the controllers. Addressing this need is the main intent of this chapter.

In particular, we evaluate two application implementation strategies for minimizing the effects of the RTOS unpredictability. A real-time system can be divided into two parts: the application software and the system environment (i.e., everything else) that the application runs in. Our first strategy aims to optimize the computer system environment for the given application. An example of such optimization is to run only the system processes necessary for the application. Our next strategy attempts to optimize the structure of application software. Regrouping software functional modules is an example of such optimization. We conclude the chapter with related work and a summary of our research results.

6.2 Prototype Open-Architecture Milling Machine Controllers

We have developed a prototype modular real-time milling machine controller on the UMOAC testbed, as shown in Figure 78. The rectangles in the figure are tasks. The ovals represent data objects shared by different tasks, while the arrows to and from shared data show the data dependencies.

This prototype three-axis controller has a linear or circular interpolator on the X and Y axes and a one-axis interpolator on the Z axis. The X and Y axes correspond to a mechanical platform on which parts can be affixed. The Z axis is a spindle. The servo

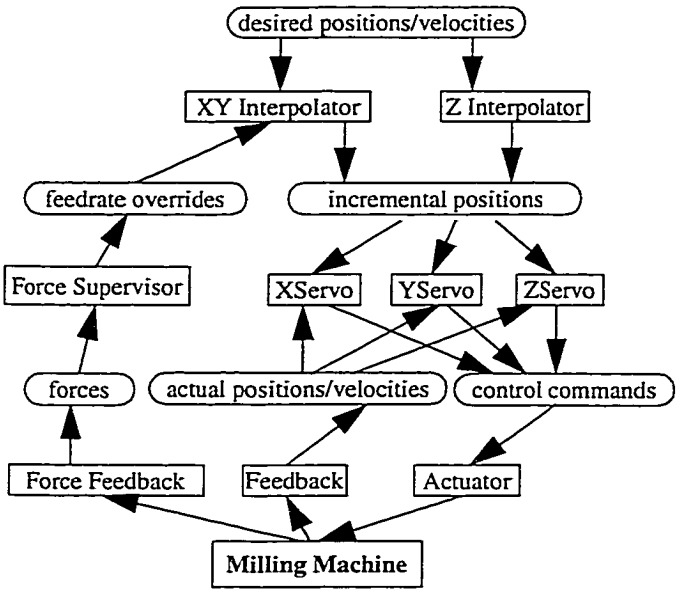


Figure 78: Prototype milling machine controller.

motion control of all axes uses either PID or fuzzy logic control law [112]. In addition, a supervisory force controller is added to the system. The Force Feedback task periodically reads the dynamometer mounted on the milling machine. It finds the maximum force reading within a window of 40 samples and sends it to the Force Supervisor. This Supervisor controls the feedrates of the X and Y axes in order to maintain a constant force while the milling machine is cutting a metal part.

The real-time tasks in our prototype milling machine controller have deadline and interval constraints. To run control tasks in our UMOAC testbed, we need to select a scheduling algorithm that performs well in terms of satisfying both constraints. Our evaluation of RM, EDF and FIFO scheduling algorithms indicates that RM is the preferred choice.

The Force Feedback task has a period of 1 *ms* and a priority of 27, which is the highest priority among all controller tasks using RM. The Force Supervisor task has a period of 40 *ms* and a priority of 24. All other tasks in Figure 78 run with a period of 10 *ms* and at the priority of 26. A larger number represents a higher priority. Tasks with the same priority are executed on a first-in-first-out (FIFO) basis.

Without loss of generality, we use a simpler version with two independent axes (Figure 79) for our performance evaluation experiments. Another reason is for safety, because we do not want to actually cut a part using the spindle (the Z axis) during our experiments. The force controller does not function without the Z axis, since its input is the force related to the cutting action.

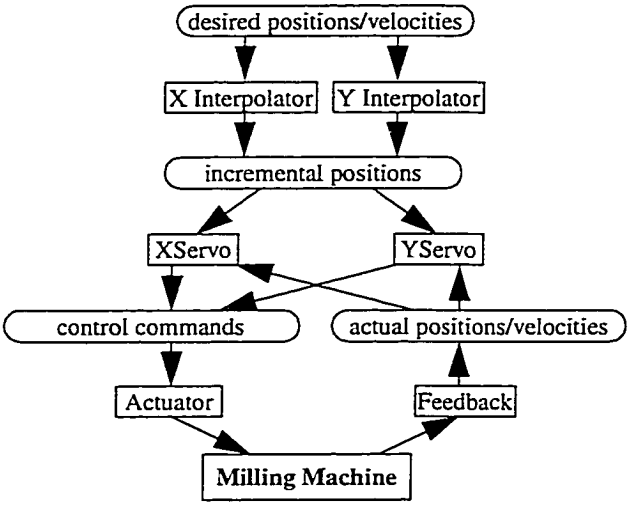


Figure 79: Prototype two-axis modular controller with separate task structure.

6.3 Two-Axis Controller with Separate Task Structure

We now identify and evaluate strategies to optimize our control application, so as to minimize the effects of RTOS unpredictability. Our two-axis open-architecture milling machine controller has a separate task structure (Figure 79). It consists of a set of cooperating tasks (rectangles in Figure 79), which are compiled and linked independently. X and Y Interpolator tasks translate desired positions and velocities into incremental X and Y positions. The incremental positions are updated and fed to X and Y Servo tasks periodically. The Feedback task periodically reads the sensors (e.g., encoders and tachometers) attached to the milling machine and posts actual positions and velocities of individual axes. Based on the incremental and actual position and velocity information, X

and Y Servo tasks compute new control commands using a specified control law for X and Y axes, respectively. Two motion control laws are currently implemented—PID and fuzzy logic [112]. In this experiment, the PID control law is used for both servo tasks. The Actuator task takes the digital control commands and sends them in either digital or analog form to the appropriate motors of the milling machine. Communications among tasks are achieved via shared memory.

An advantage of having such a separate task structure is flexibility. Tasks may be allocated to multiple nodes in a distributed computer system. One allocation strategy is to run different types of tasks on different computer nodes. Time-critical tasks may be assigned to dedicated nodes, while non-real-time tasks may be run on a general-purpose node. For example, user interface tasks (if any) may be carried out on Workstation-1 in the UMOAC testbed, while the real-time task Feedback is run on CPU-1 and other real-time tasks on CPU-2 (Figure 39).

With the separate task structure, individual tasks may also be added, removed or modified, while the remaining tasks of the application are still running.[1] One example of such flexibility is to add the Y Servo task at run-time. We first run all tasks except Y Servo. At this point, the controller has only control over the X axis. While the controller is running, we start Y Servo and inform the controller of the change. The controller then adds Y Servo to its control loop and becomes a two-axis controller.[2]

We now measure the performance of our controller. All tasks of the prototype controller are periodic and invoked once every 10 milliseconds. Each task is run as a separate computer process. The RM scheduling algorithm is used.[3] Since these processes

---

1. Of course, changes that affect the application stability should be avoided. For example, before adding a new task, we need to check if existing tasks can continue performing satisfactorily. This can be accomplished by dry running all tasks (including the new task) and measuring their performance.

2. To provide deadline guarantees, we need to consider the case where both X and Y Servo tasks are running.

have the same period of 10 *ms*, they are assigned the same priority of 26. No other process except the QNX Process Manager has a higher priority. All tasks are run on CPU-1, which is a 100 MHz 486DX4 with 32 Mbytes of RAM.

Table 33 shows the performance measurements of these tasks. The mean intervals of all tasks are very close to their nominal period of 10 *ms*, but with some variations. We now examine how RTOS unpredictability affects controller performance.

		Sample Size	Mean (μs)	Std. Dev. (μs)	Min (μs)	Max (μs)
XServo	exec. time	313	209.1	8.0	194.2	230.5
	interval	313	9996.3	33.2	9726.4	10314.7
YServo	exec. time	313	208.5	7.8	193.1	228.3
	interval	313	9996.2	22.9	9938.6	10074.2
Interval between starts of XServo and YServo		313	594.9	13.8	373.2	643.4
X Interpolator	exec. time	312	93.0	7.4	77.2	132.7
	interval	312	9996.1	24.2	9911.4	10116.1
Y Interpolator	exec. time	311	95.1	6.5	86.6	119.2
	interval	311	9996.0	21.2	9948.1	10049.9
Feedback	exec. time	313	681.5	24.3	649.6	771.0
	interval	313	9996.2	28.2	9899.9	10151.0
Actuator	exec. time	313	153.0	8.7	141.5	208.3
	interval	313	9996.3	22.5	9916.4	10090.9

Table 33: Statistics for controller with separate task structure.

We found that the controller operates stably when the PID control law is used. However, when the fuzzy logic control law [112] is used in the X and Y Servo tasks, the milling machine exhibits some jitters. One possible reason is because the fuzzy logic control law is more sensitive to timing variations. With the separate task structure,

3. A discussion of the effect of three different scheduling algorithms (EDF, RM and FIFO) can be found in Sections 3.2.

individual controller tasks are on their own timers. Therefore, the temporal relationships among the tasks are more unpredictable than those in a monolithic controller.

Ideally, for example, tasks X and Y Interpolators, Feedback, X and Y Servos, and Actuator should be executed in that order at the beginning of each period. The execution order is necessary to ensure the logical correctness of the application (see below for an example of the effect of an undesired task interleaving). The aggregation of tasks at the beginning (or any other point) of each period is to ensure that the sensor readings are consistent and as close to the *current* conditions as possible. Such relationships among tasks cannot always be guaranteed in controllers with a separate task structure. The starting times of the first few invocations of the tasks in our prototype controller are listed in Table 34. The initial execution sequence is Actuator, Feedback, X Servo, Y Servo, X Interpolator, Actuator, Y Interpolator, Feedback, X Servo, Y Servo, X Interpolator, Actuator, Y Interpolator, ..., etc.

Actuator (*ms*)	Feedback (*ms*)	X Servo (*ms*)	Y Servo (*ms*)	X Interpolator (*ms*)	Y Interpolator (*ms*)
222.33	225.56	226.68	227.18	230.09	235.15
232.46	235.61	236.75	237.23	240.14	245.17
242.36	245.59	246.73	247.22	250.08	255.16
252.34	255.55	256.71	257.19	260.07	265.15
262.35	265.54	266.69	267.19	270.05	275.12

Table 34: Start times of controller tasks.

Another possibility for the jitters is undesired task interleaving. We observe that, in Table 33, the intervals between the starts of X and Y Servo tasks have a mean of 594.9 μs and a standard deviation of 13.8 μs. Since all tasks are running as separate computer processes and there is a sizeable interval between the X and Y Servo tasks, it is conceivable that the Actuator task may run between the servo tasks and thus obtain an updated X-axis but an old Y-axis control command. This situation is illustrated in

Figure 80. If the Actuator task runs before the Y Servo task, as shown in Figure 80 (dotted arrows), it receives an updated X-axis control command ("goto x2") but an outdated Y-axis control command ("goto y1" instead of "goto y2"). As a consequence, the trajectory of the machine movement deviates from the desired course, causing a deterioration in the precision of the controlled process.

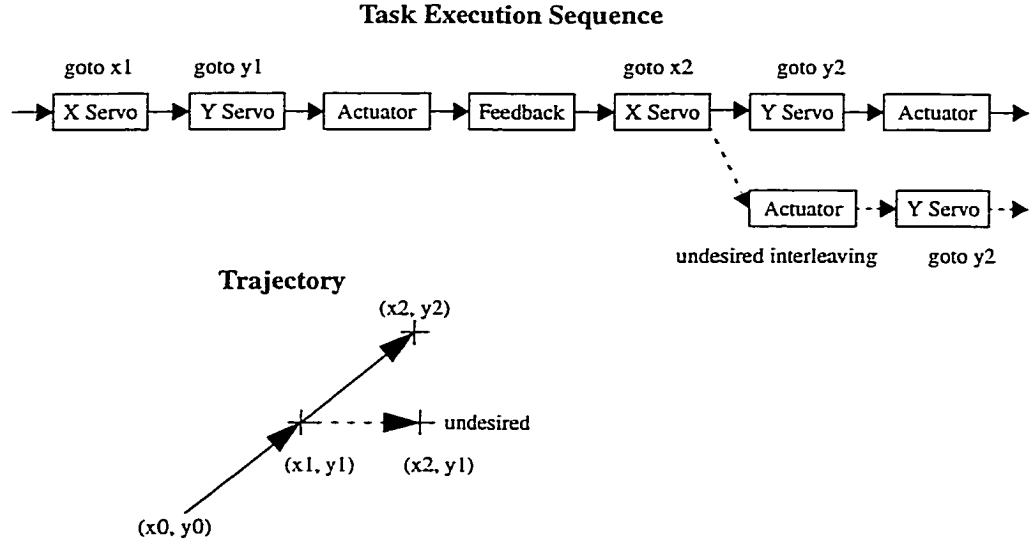


Figure 80: Task execution sequence.

While controllers with a separate task structure generally have great flexibility, they are vulnerable to timing irregularity and undesired task interleaving. In our prototype controller, such vulnerability causes jitters when using the fuzzy logic control law. We need to identify and evaluate strategies to reduce the vulnerability.

6.4 Testbed without Network

In the above experiment, we found that the actual temporal relationship between the controller tasks with a separate task structure may not always be the desired one, which can affect the performance of the controller. For example, there is a sizeable time interval between the starts of servo tasks. Ideally, the mean and standard deviation of the interval

between the X and Y Servo tasks should both be zero. This is obviously impossible in reality, but we should try to close the gap and minimize timing variation. One strategy is to locate and eliminate the cause of the timing variation.

From our experience with the testbed, we discovered that one major source of timing disturbance comes from the QNX network drivers (e.g., *Net*, *Net.ether1000*, and *Socket*). Other sources, though much less significant, include floppy and terminal drivers. Therefore, in this experiment, we want to measure the performance of the same controller in the absence of QNX network drivers. We disable the QNX network drivers, thus removing the connection between CPU-1 and other computers (see UMOAC testbed in Figure 39 of Chapter 5). This results in a standard alone controller computer.

Table 35 lists the measurement results for the execution times and periods of individual controller tasks. When the network drivers are disabled, the mean execution times and the standard deviations of both execution time and interval of most tasks are improved.[4] As a consequence, the mean interval between the X and Y Servo tasks has been shortened from 594.9 μs to 490.0 μs. The standard deviation of this interval is also improved from 13.8 μs to 9.7 μs. Figure 81 illustrates the performance difference.

This experiment indicates that the strategy of locating and eliminating sources of timing disturbance can be quite effective. However, this strategy can be infeasible when the sources are essential system hardware or software modules. This leads to our next experiment in search for an alternative strategy.

6.5 Two-Axis Controller with Combined Task Structure

Although disabling QNX network drivers can improve the performance of real-time controllers, it would be a major setback for a distributed system. Without the network

4. The **Y Interpolator** is an exception. Its performance is slightly worse without network drivers. Because during its execution, other tasks are released and cause the disturbance in its execution.

		Sample Size	Mean (μs)	Std. Dev. (μs)	Min (μs)	Max (μs)
XServo	exec. time	323	202.1	5.3	191.6	219.7
	interval	323	9996.4	19.8	9958.1	10067.7
YServo	exec. time	323	201.1	6.8	191.4	224.6
	interval	323	9996.5	19.7	9955.4	10051.8
Interval between starts of XServo and YServo		323	490.0	9.7	454.5	511.5
X Interpolator	exec. time	323	90.6	3.0	77.6	110.4
	interval	323	9996.3	22.1	9898.8	10090.3
Y Interpolator	exec. time	323	104.3	35.0	75.7	184.7
	interval	323	9996.5	22.1	9938.2	10130.7
Feedback	exec. time	323	634.8	15.6	615.4	692.7
	interval	323	9996.4	20.9	9968.4	10051.2
Actuator	exec. time	323	131.4	8.6	122.3	170.5
	interval	323	9996.7	30.5	9990.0	10128.7

Table 35: Statistics for controller with separate task structure and without QNX network drivers running.

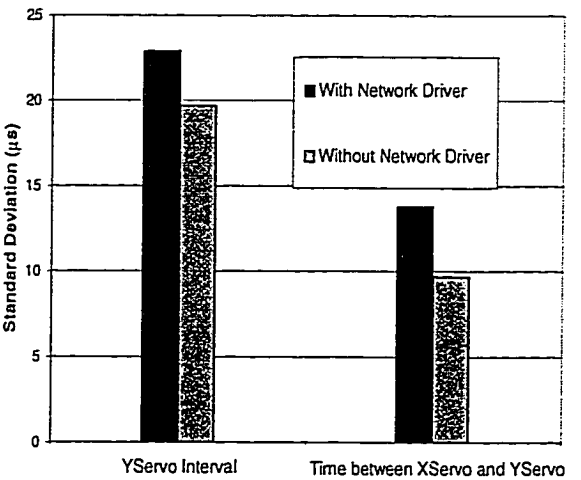


Figure 81: Performance with & without QNX network drivers.

drivers, each computer will become isolated. Although careful adjustment of network configuration and driver parameters helps reduce the problem, the network remains a major source of timing disturbance. While we do not want to sacrifice the networking capability, we need to investigate other methods to improve the performance of real-time controllers. A candidate strategy is to combine tasks with similar periods into a composite task. Note that a controller with composite tasks is different from a monolithic controller in that the components of a composite task are still modular, as in the controller with a separate task structure.

In the controller with a separate task structure, it can be difficult to prevent other tasks (e.g., the Actuator task) from running between the X and Y Servo tasks, because of the timer variation and other unpredictability associated with the RTOS. A software structure that can enforce the sequence of task execution would help. Because the mean and standard deviation of the interval between the X and Y Servo tasks should be as close to zero as possible, all tasks of the same[5] period are combined into a single task, in which the X Servo task is executed right before the Y Servo task. The composite task is compiled and linked as a single executable. It runs at a period of 10 *ms* and needs only one timer. For the tasks in a composite task, the inter-task communications among them can now be using global shared data objects instead of using shared memory since these tasks share the same address space. Figure 82 shows the new controller structure, which consists of only one composite control task now.

Table 36 lists the performance measurements of the controller with the combined task. Note that the QNX network drivers are *enabled* in this experiment. As expected, the mean interval between X and Y Servo tasks is shortened significantly from 594.9 μs to 76.9 μs (Figure 83). The standard deviation of the interval also improves (Figure 84).

---

5. If task periods are similar but not identical, it may be possible to use the shortest period as the common period for the combined task.

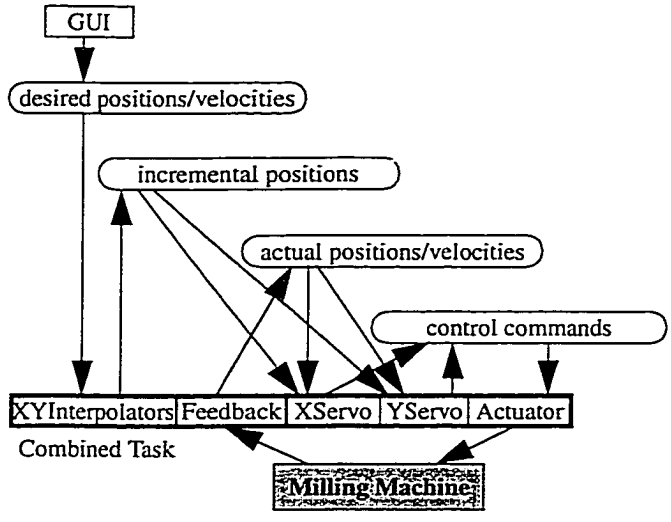


Figure 82: Prototype two-axis modular controller with combined task structure.

		Sample Size	Mean (μs)	Std. Dev. (μs)	Min (μs)	Max (μs)
Y Servo	exec. time	948	76.9	10.9	61.4	153.4
	interval	948	9995.8	104.5	9739.9	10278.8
Interval between starts of XServo and YServo		948	76.9	10.9	61.4	153.4
Combined Task	exec. time	948	998.7	88.2	896.2	1192.9
	interval	948	9996.6	33.9	9874.9	10156.6

Table 36: Statistics for controller with combined task structure and with QNX network drivers running.

However, the Y Servo task exhibits larger interval variation than that in the controller with the separate task structure, as shown in Figure 84. This is because some tasks (X and Y Interpolators, Feedback and X Servo) are executed before Y Servo in the combined task. Therefore, the variance of the Y Servo interval reflects the aggregate effect of the variances of the combined task interval and the execution times of those tasks before the Y Servo task. Even though the standard deviation of the Y Servo interval worsens, it is still relatively small—about 1% of the mean interval. On the other hand, the improvements are significant on the mean and standard deviation of the interval between X and Y Servo tasks.

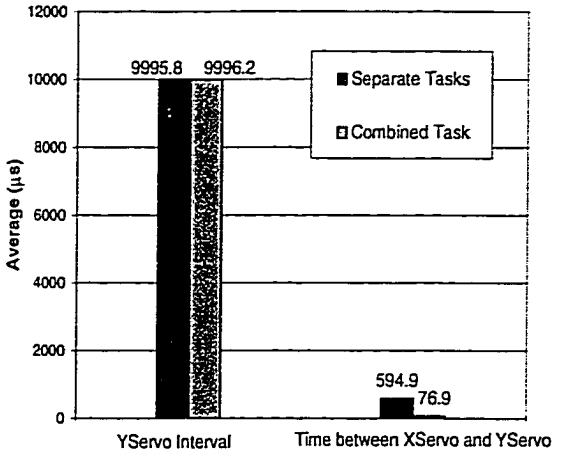


Figure 83: Mean intervals before and after combining tasks.

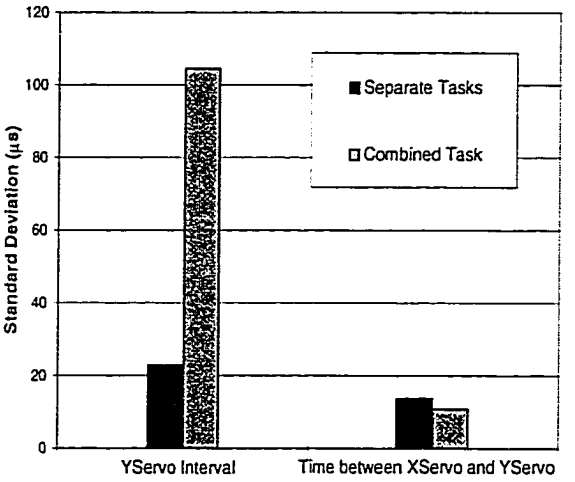


Figure 84: Standard deviations before and after combining tasks.

The modular controller with the combined task has less overhead. With the separate task structure, the mean of the total execution time was 1440.2 µs, while the

combined task structure lowers this mean to 998.7 μs. Therefore, combining the tasks reduces execution time by more than 30%. The combined task requires no context switch and shared memory inter-task communication for its constituent tasks. The savings in execution time mostly comes from the elimination of the context switch overhead. While our measurement indicates that each shared memory read or write takes about 1 μs, the context switch overhead is in the order of 10 μs [127]. Also, because all of the code is run in the same address space there are more opportunities for optimization by the application developer which may further improve the efficiency of the code.

Though not as flexible as the separate task structure, the combined task structure is easier to understand because of its sequential nature, especially considering that mechanical engineers are the main developers of control applications and many of them do not have experience in multi-tasking programming methods. Because the inter-task communication is removed, it is less complex. The sequential execution structure resembles monolithic control programs in that they all have a single thread of control. Unlike the monolithic programs, however, the new controller consists of well-defined modules and thus is easier to change, maintain and reuse. For example, the well-defined task components (i.e., Interpolator, Feedback, Servo and Actuator) in our composite controller task make it possible for us to apply our proposed MBST methodology for probabilistic deadline guarantees. We can use MBST to predict task performance if there are changes to individual task components. This is not possible with a monolithic controller.

In summary, open-architecture modular controllers with a separate task structure are very flexible, in terms of task changes and distribution. But they are vulnerable to timing variations and undesired task interleaving. In order to reduce such vulnerability, one strategy is to identify and eliminate timing disturbance sources while maintaining the separate task structure. This strategy can be very effective, but it is often infeasible because the disturbance can come from essential system hardware or software modules,

e.g., device drivers. The alternative strategy is to combine tasks with the same period into a composite task. While not as flexible as those with a separate task structure, the resulting controllers with the combined task structure offer deterministic task execution orders and smaller overhead. There is clearly a trade-off between feasibility and performance. The optimal point may be application dependent.

6.6 Related Work

Koren *et al.* [80] analyzed the effect of control architectures and communication networks on a manufacturing system's performance. In particular, they estimated the amount of real-time data and communication network bandwidth needed in open-architecture controllers. We instead characterize and measure controller performance in terms of deadline and interval constraints.

The study by Abdelzaher and Shin [3], which was conducted independently, indicates that grouping tasks by period has the potential of increasing CPU utilization while keeping the task set schedulable. This is consistent with our findings with real controller tasks.

We are not aware of any published work on empirical studies of the impact of software timer unpredictability on the performance of modular real-time controllers. This work is important in that it reveals practical issues associated with the implementation of real-time open-architecture controllers. The experiences gained and the lessons learned from this empirical study can be valuable to practitioners as well as researchers.

6.7 Summary

Traditional, monolithic controllers are being modularized to take advantage of newly-available hardware and software components and to reduce controller development and maintenance costs. Since the resulting modular open-architecture controllers rely on the

RTOS services, the behavior of the underlying RTOS can have a significant impact on the controller performance.

Given the presence of RTOS unpredictability, we identified and evaluated two strategies to improve the performance of modular controllers. The first strategy is to study the interaction among application and RTOS modules. Once the problem sources are discovered, they can be avoided or fine-tuned in order to minimize their impact. However, this strategy can be ineffective when the problem sources are essential system modules. In this case, a more desirable strategy is to study the software architecture of controllers. There are opportunities to improve the controller software architecture by optimizing data flow or interaction among controller modules. For example, we reorganized separate tasks of our prototype controller with the same period into a single task. Our experiment showed that the controller with the combined task structure exhibits significant performance improvements.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

7.1 Research Contributions

Our work has made following important contributions to real-time systems research:

- We identified and measured characteristics of system unpredictability in three commercial RTOSs: VxWork, QNX and pSOSystem.
- We investigated the effects of non-ideal characteristics of RTOSs on the schedulability of periodic real-time tasks under RM, EDF and FIFO scheduling algorithms. We found that RTOS unpredictability, such as the interval variation of POSIX timers, has a significant impact on the system's ability to meet task deadlines.
- To make hard deadline guarantees in the presence of RTOS unpredictability, we proposed RMTU to augment the original RM scheduling theory. We also designed systematic experiments to derive the model parameters.
- For non-hard real-time tasks requiring performance guarantees, we developed a practical framework for probabilistic deadline guarantees, whose components include PRTCM, completion-probability-cognizant and CPU-utilization-cognizant heuristics, a comparative study of scheduling algorithm performance, and MBST. Our performance evaluation of RM, EDF, FIFO and our new heuristics showed that RM performs well in terms of useful job ratio while UM_CP is superior in terms of task completion probability miss ratio.

Furthermore, our experiments with prototype milling machine controllers in the UMOAC testbed demonstrated the validity of MBST.

- To provide guidelines for real-time application development in the presence of RTOS unpredictability, we identified strategies to minimize its impact—tuning the underlying computer system and optimizing the software architecture of the application itself. Our measurement data of prototype open-architecture milling machine controllers showed that, while both strategies are effective, the latter produces better results.

Our research results reported in this thesis represent the first step in developing practical solutions to real-time computing issues in real-world applications, in particular, open-architecture manufacturing control systems.

7.2 Future Directions

Several areas of our research can be extended. Because of the difficulties associated with providing hard or probabilistic deadline guarantees in the presence of RTOS unpredictability, we have focused our research on the uniprocessor case. The techniques developed apply to real-time tasks after they are assigned to the processor. As real-time applications become more and more distributed, it would be very helpful to study the effects of RTOS and communication network unpredictability on real-time applications. In order to provide deadline guarantees, we would need to develop principles for task assignment or load sharing in the presence of system unpredictability.

Because some of our real-time deadline guarantee techniques are empirical, their results are system-dependent. In the context of open control systems, reconfiguration of the mechanical machine hardware necessitates the reconfiguration of the computing and control environments as well. This inadvertently affects the real-time performance of the system. Performance assessment methodologies and automated tools would be of great

value in order to re-calibrate the parameters of our empirical models for deadline guarantees.

System integration would also be a very important problem. It would be helpful to develop a knowledge base that encapsulates our empirical deadline guarantee models and methodologies, as well as other real-time computing principles and application hardware/ software component models. This knowledge base will then be consulted in determining how to integrate different components.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Robert Abbott, and Hector Garcia-Molina, "Scheduling Real-time Transactions," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pp. 71-81.

[2] Robert K. Abbott, and Hector Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," *ACM Transactions on Database Systems*, Vol. 17, No. 3, September 1992, pp. 513-560.

[3] Tarek F. Abdelzaher, and Kang G. Shin, "*Period-Based Partitioning and Allocation of Large Real-Time Applications*," 1996.

[4] D. Agrawal, and V. Krishnaswamy, "Using Multiversion Data for Non-interfering Execution of Write-only Transactions," *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, May 1991, pp. 98-107.

[5] James Albus, "RCS: A Reference Model Architecture for Intelligent Machine Systems," *Proceedings of the International Workshop on Open-Architecture Controllers for Automation*, Ann Arbor, Michigan, April 1994.

[6] Y. Altintas, and W.K. Munasinghe, "A hierarchical open-architecture CNC system for machine tools," *Annals of the CIRP, Vol. 43, No. 1*, 1994, pp. 349-354.

[7] B. Anderson, "Next Generation Workstation/Machine Controller (NGC)," *Proc. IPC'92*, April 1992, pp. xix-xxvi.

[8] Hagit Attiya, *et al.*, "Bounds on the time to reach agreement in the presence of timing uncertainty," *Journal of the Association for Computing Machinery*, Vol. 41, No. 1, January 1994, pp. 122-152.

[9] A. Attoui, and M. Schneider, "An Object Oriented Model for Parallel and Reactive Systems," *Proceedings of Real-Time Systems Symposium*, December 1991, pp. 84-93.

[10] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Software Engineering Journal*, September 1993, pp. 284-292.

[11] N.C. Audsley, I.J. Bate, and A. Burns, "Putting Fixed Priority Scheduling Theory into Engineering Practice for Safety Critical Applications," P*roceedings of the 1996 IEEE Real-Time Technology and Applications Symposium*, June 1996, pp. 2-10.

[12] S. Baruah, D. Chen, and A. Mok, "Jitter Concerns in Periodic Task Systems," *Proceedings of Real-Time Systems Symposium*, December 1997, pp. 68-77.

[13] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the Competitiveness of On-Line Real-Time Task Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1991.

[14] G. Bernat, and A. Burns, "Combining (n m)-Hard Deadlines and Dual Priority Scheduling," *Proceedings of Real-Time Systems Symposium*, 1997, pp. 46-57.

[15] Carlton Bickford, Marie S. Teo, Gary Wallace, John A. Stankovic, and Krithi Ramamritham, "A Robotic Assembly Application on the Spring Real-Time Sys-

tem," *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symposium*, June 1996, pp. 19-28.

[16] Thomas E. Bihari, "Current Issues in the Development of Real-Time Control Software," *Real-Time Systems Newsletter*, Vol. 5, No. 1, 1989, pp. 1-5.

[17] Thomas E. Bihari, and Prabha Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples," *IEEE Computers*, December 1992, pp. 25-32.

[18] Sushil Birla, "A Conceptual Framework for Modeling Manufacturing Automation," *Directed Study Report*, Department of Electrical Engineering and Computer Science, University of Michigan, September 1993.

[19] Sushil Birla, "Modeling Sensors in Manufacturing Automation," Department of Electrical Engineering and Computer Science, University of Michigan, 1995.

[20] Sushil Birla, and Kang Shin, "Intelligent Control of Manufacturing Automation: Making it Affordable and Maintainable," *Proceedings of the 27th CIRP International Seminar on Manufacturing Systems*, May 1995, pp. 59-68.

[21] S. Biyabani, J. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1988.

[22] Grady Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.

[23] R. A. Bowman and J. A. Muckstadt, "Stochastic analysis of cyclic schedules," *Operations Research, 41*, September-October 1993, pp. 947-958.

[24] A. P. Buchmann, *et al.*, "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *Proc. Fifth Data Engineering Conf.*, 1989, pages 470-480.

[25] G. Buttazzo, M. Spuri, and F. Sensini, "Value vs. Deadline Scheduling in Overload Conditions," *Proceedings of Real-Time Systems Symposium*, 1995, pp. 90-99.

[26] Ken Chen and Paul Muhlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function," *The Journal of Real-Time Systems*, Vol. 10, No. 3, May 1996, pp. 293-312.

[27] Albert Mo Kim Cheng, "Scheduling transactions in real-time database systems," *38th Annual IEEE Computer Society International Computer Conference - COMPCON SPRING '93*, 1993, pp. 222-231.

[28] S. Cheng, J. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proceedings of Real-Time Systems Symposium*, December 1986.

[29] Jan Chomicki, "Real-time integrity constraints," *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1992, pp. 274-282.

[30] Manhoi Choy, Mei-Po Kwan, and Hong Va Leong, "On real-time distributed geographical database systems," *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS-27)*, 1994, pp. 337-346.

[31] Jen-yao Chung, Jane W.S. Liu, and Kwei-jay Lin, "Scheduling Periodic Jobs That Allow Imprecise Results," *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990, pp. 1156-1174.

[32] F. Civello, A. Copsey, and P. Terelak, "Object-oriented development of real-time monitoring and control systems: a case study," *IEE Colloquium on the Design, Implementation and Use of Object-Oriented Systems*, London, UK. January 1994, pp. 3/1-3/3.

[33] E.G. Coffman, Jr. and Zhen Liu, "On the optimal stochastic scheduling of out-forests," *Operations Research*, Vol. 40, pp. S67-S75, Jan-Feb., 1992.

[34] James Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[35] Michael F. Coulas, Glenn H. Macewen, and Genevieve Marquis, "RNet: A Hard Real-Time Distributed Programming System," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987, pp. 917-932.

[36] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jauhari, "The HiPAC Project: Combining Active Databases and Timing Constraints," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pp. 51-70.

[37] Lisa DiPippo, and Victor Wolfe, "Object-based Semantic Real-time Concurrency Control," *Proceedings of Real-Time Systems Symposium*, December 1993, pp. 87-96.

[38] Stuart R. Faulk, and David L. Parnas, "On Synchronization in Hard-Real-Time Systems," *Communications of the ACM*, Vol. 31, No. 3, March 1988, pp. 274-287.

[39] Esther Frostig, "A stochastic scheduling problem with intree precedence constraints," *Operations Research*, Vol. 36, pp. 937-43, Nov-Dec., 1988.

[40] Hector Garcia-Molina, and Kenneth Salem, "Main Memory Database Systems: An Overview," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992, pp. 509-516.

[41] Michael R. Garey, and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.

[42] Roman Ginis, Victor Fay Wolfe, and J.J. Prichard, "The Design of an Open System with Distributed Real-Time Requirements," *Proceedings of the 1996 IEEE Real-Time Technology and Applications Symposium*, June 1996, pp. 82-90.

[43] Marc H. Graham, "How to get serializability for real-time transactions without having to pay for it," *Proceedings of Real-Time Systems Symposium*, December 1993, pp. 56-65.

[44] Marc H. Graham, "Real Time Data Management," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software*, May 1993, pp. 51-56.

[45] Marc H. Graham, "Issues in Real-Time Data Management," *The Journal of Real-Time Systems*, 4, 1992, pp. 185-202.

[46] Vincenzo Grassi, Lorenzo Donatiello, and Salvatore Tucci, "On the optimal checkpointing of critical tasks and transaction-oriented systems," IEEE Transactions on Software Engineering, Vol. 18, No. 1, January 1992, pp. 72-77.

[47] Ching-Chih Han, and Kwei-Jay Lin, "Scheduling Distance-Constrained Real-Time Tasks," *Proceedings Real-Time Systems Symposium*, 1992, pp. 300-308.

[48] Ching-Chih Han, and Kwei-Jay Lin, "Scheduling Real-Time Computations with Separation Constraints," *Information Processing Letters*, 42, 1992, pages 61-66.Hector Garcia-Molina and Bruce Lindsay, "Research Directions for Distributed Databases," *SIGMOD Record*, Vol. 19, No. 4, December 1990, pp. 98-103.

[49] Seungjae Han, Kang G. Shin, and Jaehyun Park, "A Non-intrusive Distributed Monitoring Support in Fault Injection Experiments," *4th IEEE International Workshop on Evaluation Techniques for Dependable Systems*, October 1995.

[50] Jayant R. Haritsa, "Approximate Analysis of Real-Time Database Systems," *Proceedings of the 10th International Conference on Data Engineering*, 1994, pp. 10-19.

[51] Jayant R. Haritsa, Miron Livny, and Michael J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of Real-Time Systems Symposium*, December 1991, pp. 232-242.

[52] Jayant R. Haritsa, Michael J. Carey, and Miron Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *The Journal of Real-Time Systems*, 4, 1992, pp. 203-241.

[53] Tony Haynes, "The NGC/LEC Project," *Proceedings of the International Workshop on Open-Architecture Controllers for Automation*, Ann Arbor, Michigan, April 1994.

[54] P.S. Heidmann, "A Statistical Model for Designers of Rate Monotonic Systems," *RMA Users Forum*, Software Engineering Institute, PA, 1994.

[55] Jiawei Hong, Xiaonan Tan, and Don Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System," *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1736-1744.

[56] John E. Hopcroft, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[57] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeo K. Taneja, "Processing Aggregate Relational Queries with Hard Time Constraints," *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989, pp. 68-77.

[58] Jing Huang, and Le Gruenwald, "Logging techniques in real-time database systems," *Proceedings of the Energy-Sources Technology Conference*, 1994, pp. 247-257.

[59] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proceedings of the 17th International Conference on Very Large Data Bases*, Very Large Data Base Endowment, September 1991, pp. 35-46.

[60] Jiandong Huang, *et al.*, "On Using Priority Inheritance In Real-Time Databases," *Proceedings of Real-Time Systems Symposium*, December 1991, pp. 210-221.

[61] Jiandong Huang, *et al.*, "Priority Inheritance In Soft Real-Time Databases," *The Journal of Real-Time Systems*, 4, 1992, pp. 243-268.

[62] IEEE, *IEEE Guide to POSIX Open System Environment (IEEE 1003.0)*, 1995.

[63] Integrated Systems Inc., *pSOSystem/68K User's Manual*, 1992.

[64] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer, "An Object-Oriented Real-Time Programming Language," *IEEE Computer*, October 1992, pp. 66-73.

[65] Farnam Jahanian, and Aloysius K.-L. Mok, "A Graph-Theoretic Approach for Timing Analysis and its Implementation," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987, pp. 961-975.

[66] Kevin Jeffay, and Donald L. Stone, "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems," *Proceedings of Real-Time Systems Symposium*, December 1993, pp. 212-221.

[67] E.D. Jensen, C.D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating System," *Proceedings of Real-Time Systems Symposium*, December 1985, pp. 112-122.

[68] M. Joseph, and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, Vol. 29, No. 5, 1986, pp. 390-395.

[69] Sanjay Kamat, Nicholas Malcolm, and Wei Zhao, "Performance Evaluation of a Bandwidth Allocation Scheme for Guaranteeing Synchronous Messages with Arbitrary Deadlines in an FDDI Network," *Proceedings of Real-Time Systems Symposium*, December 1993, pp. 34-43.

[70] Ryoji Kataoka, Tetsuji Satoh, and Kenji Suzuki, "Survey of concurrency control for real-time database systems," *IEICE Transactions on Information and Systems*, Vol. E76-D, No. 2, February 1993, pp. 145-153.

[71] Kevin B. Kenny, and Kwei-Jay Lin, "Building Flexible Real-Time Systems Using the Flex Language," *IEEE Computer*, May 1991, pp. 70-78.

[72] Kevin A. Kettler, Daniel I. Katcher, and Jay K. Strosnider, "A Modeling Methodology for Real-Time/Multimedia Operating Systems," *Proc. Real-Time Technology and Applications Symposium*, pp. 15-26, May 1995

[73] Jinhwan Kim, and Heonshik Shin, "Priority-driven concurrency control based on data conflict state in distributed real-time databases," *Microprocessing and Microprogramming*, Vol. 38, No. 1-5, September 1993, pp. 491-498.

[74] Woosaeng Kim, and Jaideep Srivastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling," *Proceedings of Real-Time Systems Symposium*, December 1991, pp. 222-231.

[75] Young-Kuk Kim, and Sang H. Son, "Predictability and Consistency in Real-Time Database Systems," in S. Son ed., *Advances in Real-Time Systems*, Prentice Hall, 1995, pp. 509-531.

[76] Young-Kuk Kim, "Predictability and Consistency in Real-Time Transaction Processing," *PhD Dissertation*, University of Virginia, May 1995.

[77] Mark H. Klein, *A Practitioner's handbook for real-time analysis guide to rate monotonic analysis for real-time systems*, Kluwer Academic Publishers, Boston, July 1993.

[78] Mark H. Klein, John P. Lehoczky, and Ragunathan Rajkumar, "Rate-Monotonic Analysis for Real-Time Industrial Computing," *IEEE Computer*, January 1994, pp. 24-32.

[79] G. Koren, and D. Shasha, "Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips," *Proceedings of Real-Time Systems Symposium*, 1995, pp. 110-117.

[80] Yoram Koren, Zbigniew J. Pasek, A. Galip Ulsoy, and Paul K. Wright, "Timing and Performance of Open Architecture Controllers," to appear in *Proceedings of 1996 ASME International Mechanical Engineering Congress and Exposition*, Atlanta, Georgia, November 1996.

[81] Henry F. Korth, Nandit Soparkar, and Abraham Silberschatz, "Triggered Real-Time Databases with Consistency Constraints," *Proceedings of the 16th VLDB Conference*, 1990, pp. 71-82.

[82] Tei-Wei Kuo, and Aloysius K. Mok, "Using Data Similarity to Achieve Synchronization for Free," *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, 1994, pp. 112-116.

[83] Tei-Wei Kuo, and Aloysius K. Mok, "SSP: a Semantics-Based Protocol for Real-Time Data Access," *Proceedings of Real-Time Systems Symposium*, December 1993, pp. 76-86.

[84] James F. Kurose, and Renu Chipalkatti, "Load Sharing in Soft Real-Time Distributed Computer Systems," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987, pp. 993-1000.

[85] Kam-yiu Lam, and Sheung-lun Hung, "Static Two Phase Locking Protocols for Concurrency Control in Distributed Real-time Database Systems," *Proceedings of the First International Workshop on Real-Time Computing Systems and Applications*, Seoul, Korea, December 1994, pp. 185-189.

[86] Juhnyoung Lee, Myung-Joon Lee, and Cheol Su Lim, "Challenges in Real-Time Object-Oriented Databases," *Proceedings of the First International Workshop on Real-Time Computing Systems and Applications*, Seoul, Korea, December 1994, pp. 190-195.

[87] Juhnyoung Lee, and Sang H. Son, "Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems," *Proceedings of Real-Time Systems Symposium*, December 1993, pp. 66-75.

[88] J.P. Lehoczky, *et al.*, "Fixed Priority Scheduling Theory for Hard Real-Time Systems," A.M. van Tilborg and G.M. Koob, eds., *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kluwer Academic Publishers, Boston, 1991, pp. 1-30.

[89] J.P. Lehoczky, "Real-Time Resource Management Techniques," J.J. Marciniak, ed., *Encyclopedia of Software Engineering*, John Wiley and Sons, New York, 1994, pp. 1011-1020.

[90] Alberto Leon-Garcia, *Probability and Random Processes for Electrical Engineering*, Addison-Wesley, 1989.

[91] Kwei-Jay Lin, "Consistency Issues in Real-Time Database Systems," *Proceedings of the 22nd Annual Hawaii International Conference on System Science*, 1989, pp. 654-661.

[92] Kwei-Jay Lin and Ming-Ju Lin, "Enhancing Availability in Distributed Real-Time Databases," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pp. 34-43.

[93] C.L. Liu, and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, January 1973, pp. 46-61.

[94] Jane W.S. Liu, *et al.*, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, May 1991, pp. 58-68.

[95] C.D. Locks, "Best-Effort Decision Making for Real-Time Scheduling," *Ph.D. dissertation*, Department of Computer Science, Carnegie Mellon University, 1986.

[96] Victor B. Lortz, "An Object-Oriented Real-Time Database System for Multiprocessors," *Ph.D. dissertation*, Department of Electrical Engineering and Computer Science, University of Michigan, April 1994.

[97] Victor B. Lortz, and Kang G. Shin, "Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization," *Proceedings of OOPSLA'94*, pp. 453-467.

[98] Victor B. Lortz, and Kang G. Shin, "Semaphore Queue Priority Assignment for Real-Time Multiprocessor Synchronization," *IEEE Trans. on Software Engineering*, Vol. 21, No. 10, October 1995, pp. 834-844.

[99] S. Marche, "Measuring the Stability of Data Models," *European Journal of Information Systems*, Vol. 2, No. 1, 1993, pp. 37-47.

[100] Martin Marietta Astronautics Group, *Next Generation Workstation/Machine Controller Specification for an Open System Architecture Standard*, NGC-0001-13-000-SYS edition, March 1992.

[101] John Marsh, *et al.*, "Object-oriented management of real-time data in integrated avionics architectures," *1993 IEEE National Aerospace and Electronics Conference*, 1993, pp. 529-534.

[102] M. Marucheck, and J.K. Strosnider, "An Evaluation of the Graceful Degradation Properties of Real-Time Schedulers," *The Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, 1995.

[103] Clifford W. Mercer, and Hideyuki Tokuda, "The ARTS Real-Time Object Model," *Proceedings of the 11th Real-Time Systems Symposium*, 1990, pp. 2-10.

[104] A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," *Ph.D. dissertation*, Massachusetts Institute of Technology, May 1983.

[105] Magnus Morin, *et al.*, "Real-Time Hierarchical Control," *IEEE Software*, September 1992, pp. 51-57.

[106] Jogesh K. Muppala, Steven P. Woolet, and Kishor S. Trivedi, "Real-Time-Systems Performance in the Presence of Failures," *IEEE Computer*, May 1991, pp. 37-47.

[107] Hidenori Nakazato and Kwei-Jay Lin, "A Design Methodology for Real-Time Database Systems," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software*, May 1993, pp. 64-68.

[108] Hidenori Nakazato, and Kwei-Jay Lin, "Concurrency control algorithms for real-time systems," *Microprocessing and Microprogramming*, Vol. 38, No. 1-5, September 1993, pp. 647-654.

[109] Swaminathan Natarajan, and Wei Zhao, "Issues in Building Dynamic Real-Time Systems," *IEEE Software*, September 1992, pp. 16-21.

[110] *Next Generation Workstation/Machine Controller (NGC) Requirements Definition Document (RDD)*, 1989.

[111] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1992.

[112] Open-Architecture Controls Team, Developer's Guide for Open-Architecture Control of the Robotool, Dept. of Electrical Engineering & Computer Science and Dept. of Mechanical Engineering & Applied Mechanics, U. of Michigan, 1995.

[113] Gultekin Ozsoyoglu, *et al.*, "Processing Real-Time, Non-Aggregate Queries with Time-Constraints in CASE-DB," *Proceedings of the 8th International Conference on Data Engineering*, January 1992, pp. 410-417.

[114] Gultekin Ozsoyoglu, and Richard Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 4, August 1995, pp. 513-532.

[115] M. Tamer Ozsu, and Patrick Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, 1991.

[116] Chang Yun Park, and Alan C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *IEEE Computer*, May 1991, pp. 48-57.

[117] Jaehyun Park, *et al.*, "An Open Architecture Real-Time Controller for Machining Processes," P*roceedings of the 27th CIRP International Seminar on Manufacturing Systems*, May 1995, pp. 27-34.

[118] Joan Peckham, Victor F. Wolfe, JJ Prichard, and Lisa C. DiPippo, "RTSORAC: Design of a Real-Time Object-Oriented Database System," *Technical Report 94-231*, University of Rhode Island, 1994.

[119] Dartzen Peng, and Kang G. Shin, "Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control," *IEEE Transactions on Computers*, Vol. C-36, No. 4, April 1987, pp. 500-516.

[120] James L. Peterson, and Abraham Silberschatz, *Operating System Concepts*, Second Edition, Addison-Wesley, 1985.

[121] Wade D. Peterson, *The VMEbus Handbook*, expanded third edition, VFEA International Trade Association, 1993.

[122] Michael Pinedo, "Stochastic scheduling with release dates and due dates," *Operations Research, 31*, May-June 1983, pp. 559-572.

[123] A. Pizzarello, and F. Golshani, "In-Memory Databases: An industry perspective," *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, 1992, pp. 96-101.

[124] Gustav Pospischil, *et al.*, "Developing Real-Time Tasks with Predictable Timing," *IEEE Software*, September 1992, pp. 35-44.

[125] G. Pritschow, and C. Daniel, "Open Control System - A Future-Oriented Concept," *Proceedings of the 27th CIRP International Seminar on Manufacturing Systems*, May 1995, pp. 5-17.

[126] Calton Pu, and Krithi Ramamritham, "A Formal Characterization of Epsilon Serializability," 1992.

[127] QNX Software Systems Ltd., *QNX Documentations*, http://www.qnx.com.

[128] Young-Gook Ra, and Elke A. Rundensteiner, "A Transparent Object-Oriented Schema Change Approach Using View Evolution," *Proceedings of the 11th International Conference on Data Engineering*, March 1995, pp. 165-172.

[129] R. Rajkumar, L. Sha, and J.P. Lehoczky, "Real-time synchronization protocols for multiprocessors," *Proc. Real-Time Systems Symposium*, Dec. 1988, pp. 259-269.

[130] Krithi Ramamritham, "Real-Time Databases," *Distributed and Parallel Databases*, 1, 1993, pp. 199-226.

[131] Krithi Ramamritham, and Panos K. Chrysanthis, "In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties," *Distributed Object Management*, Ozsu, Dayal, and Valduriez Ed., Morgan Kaufmann Publishers, 1992.

[132] Krithi Ramamritham, and Nandit Soparkar, "Report on DART'96: Databases: Active and Real-Time (Concepts meet Practice)," 1996.

[133] Krithi Ramamritham, John A. Stankovic, and Wei Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, August 1989, pp. 1110-1123.

[134] Karsten Schwan, Prabha Gopinath, and Win Bo, "CHAOS-Kernel Support for Objects in the Real-Time Domain," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987, pp. 904-916.

[135] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin, "On Task Schedulability in Real-Time Control Systems," 1996.

[136] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pp. 82-98.

[137] L. Sha, R. Rajkumar, and J.P. Lehocsky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990, pp. 1175-1185.

[138] Wei Kuan Shih, Jane W.S. Liu, and C.L. Liu, "Modified Rate-Monotonic Algorithm for Scheduling Periodic Jobs with Deferred Deadlines," *IEEE Transactions on Software Engineering*, Vol. 19, No. 12, December 1993.

[139] Kang G. Shin, "Real-Time Communications in a Computer-Controlled Workcell," *IEEE Transactions on Robotics and Automation*, Vol. 7, No. 1, February 1991, pp. 105-113.

[140] Kang G. Shin, and C.M. Krishna, "New Performance Measures for Real-Time Digital Computer Controls and Their Applications," *Control and Dynamic Systems*, Academic Press, 1990.

[141] Kang G. Shin, C.M. Krishna, and Yann-hang Lee, "A Unified Method for Evaluating Real-Time Computer Controllers and Its Application," *IEEE Transactions on Automatic Control*, Vol. AC-30, No. 4, April 1985, pp. 357-366.

[142] Kang G. Shin, and Parameswaran Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *IEEE Proceedings*, Vol. 82, No. 1, January 1994, pp. 6-24.

[143] LihChyun Shu and Michal Young, "Real-Time Concurrency Control with Analytic Worst-Case Latency Guarantees," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software*, May 1993, pp. 69-73.

[144] B.W. Silverman, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, 1986.

[145] Mukesh Singhal, "Issues and Approaches to Design of Real-Time Database Systems," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pp. 19-33.

[146] Dag Sjøberg, "Quantifying Schema Evolution," *Information and Software Technology*, Vol. 35, No. 1, January 1993, pp. 35-54.

[147] S.H. Son, "Scheduling real-time transactions using priority," *Information and Software Technology*, Vol. 34, No. 6, June 1992, pp. 409-415.

[148] Sang H. Son, *et al.*, "Integration of a database system with real-time kernel for time-critical applications," *Proceedings of the Second International Conference on Systems Integration*, 1992, pp. 172-180.

[149] Sang H. Son, Juhnyoung Lee, and Yi Lin, "Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control," *The Journal of Real-Time Systems*, 4, 1992, pp. 269-276.

[150] Sang H. Son, Juhnyoung Lee, and Savita Shamsunder, "Real-time transaction processing: pessimistic, optimistic, and hybrid approaches," *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, 1992, pp. 222.

[151] Sang H. Son, Young-Kuk Kim, and Robert C. Beckinger, "MRDB: A multi-user real-time database testbed," *Proceedings of the Hawaii International Conference on System Sciences*, 1994, pp. 543-552.

[152] Nandit Soparkar, Henry F. Korth, and Avi Silberschatz, "Time-Constrained Transaction Scheduling," *Technical Report*, TR-92-46, Department of Computer Sciences, University of Texas at Austin, December 1992.

[153] John A. Stankovic, "Misconceptions About Real-Time Computing," *IEEE Computer*, October 1988, pp. 10-19.

[154] John A. Stankovic, "Decentralized Decision Making for Task Reallocation in a Hard Real-Time System," *IEEE Transactions on Computers*, Vol. 38, No. 3, March 1989, pp. 341-355.

[155] John A. Stankovic, and Krithi Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, May 1991, pp. 62-72.

[156] John A. Stankovic and Wei Zhao, "On Real-Time Transactions," *SIGMOD Record*, Vol. 17, No. 1, March 1988, pp. 4-18.

[157] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-Based Robotic Applications," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1282-1295, November/ December 1992.

[158] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects," *Technical Report CMU-RI-TR-93-11*, Carnegie Mellon University, July 1993.

[159] Bjarne Stroustrup, *The C++ Programming Language*, second edition, Addison-Wesley, 1991.

[160] M. Szafarczyk, "Open Architecture Controllers and Automatic Supervision in Manufacturing," *Proceedings of the 27th CIRP International Seminar on Manufacturing Systems*, May 1995, pp. 45-50.

[161] P. Thambidurai, and K.S. Trivedi, "Transient Overloads in Fault-Tolerant Real-Time Systems," *Proceedings of Real-Time Systems Symposium*, December 1989.

[162] L.M. Thompson, "Using pSOS+ for Embedded Real-Time Computing," *COMPCON 1990*, pp. 282-288.

[163] T.S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu, "Probabilistic Performance Guarantees for Real-Time Tasks with Varying Computation Times," *Proceedings of Real-Time Technology and Applications Symposium*, 1995, pp. 164-173.

[164] K.W. Tindell, A. Burns and A. Wellings, "An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks," *Real-Time Systems*, 6, 1994, pp. 133-151.

[165] Hideyuki Tokuda, and Clifford W. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, 23(3), July 1989, pp. 29-53.

[166] Frank Shou-Cheng Tseng, Arbee L.P. Chen, and Wei-Pang Yang, "A Probabilistic Approach to Query Processing in Heterogeneous Database Systems," *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, February 1992, pp. 176-183.

[167] Ozgur Ulusoy, "Current Research on Real-Time Databases," *SIGMOD Record*, Vol. 21, No. 4, December 1992, pp. 16-21.

[168] Steve Vestal, "On the Accuracy of Predicting Rate Monotonic Scheduling Performance," *Tri-Ada*, December 1990.

[169] VFEA International Trade Association, *The VMEbus Specification.*

[170] P.J. Weinberger, and Debasis Mitra, "Probabilistic models of database locking: solutions, computational algorithms, and asymptotics," *Journal of the Association for Computing Machinery*, 31, October 1984, pp. 855-878.

[171] Wind River Systems, *VxWorks Reference Manual 5.1*, 1993.

[172] V. Wolfe, L.C. DiPippo, and P.J. Fortier, "The Design of Real-Time Extensions to the Open Object-Oriented Database System," *TR94-236*, Department of Computer Science, University of Rhode Island, 1994.

[173] Victor F. Wolfe, Lisa B. Cingiser, J. Peckham, and J. Prichard, "A Model For Real-Time Object-Oriented Databases," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software*, May 1993, pp. 57-63.

[174] Paul Wright, "Open-Architecture Manufacturing," *Proceedings of the International Workshop on Open-Architecture Controllers for Automation*, Ann Arbor, Michigan, April 1994.

[175] Jia Xu, and David L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990, pp. 360-369.

[176] Yong I. Yoon, and Song C. Moon, "Reliable transaction processing for real-time distributed database systems," *Microprocessing and Microprogramming*, Vol. 34, No. 1-5, February 1992, pp. 63-66.

[177] Philip S. Yu, *et al.*, "On Real-Time Databases: Concurrency Control and Scheduling," *Proceedings of the IEEE*, Vol. 82, No. 1, January 1994, pp. 140-157.

[178] Wei Zhao, Krithi Ramamritham, and John A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987, pp. 949-960.

[179] Lei Zhou, Elke A. Rundensteiner, and Kang G. Shin, "Schema Evolution for Real-Time Object-Oriented Databases," *Technical Report CSE-TR-199-94*, Department of Electrical Engineering and Computer Science, University of Michigan, March 1994.

[180] Lei Zhou, Elke A. Rundensteiner, and Kang G. Shin, "OODB Support for Real-Time Open-Architecture Controllers," *Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)*, April 1995, pp. 206-213.

[181] Lei Zhou, Michael J. Washburn, Kang G. Shin, and Elke A. Rundensteiner, "Performance Evaluation of Modular Real-Time Controllers," *Proceedings of 1996 ASME International Mechanical Engineering Congress and Exposition: Dynamic Systems and Control Division*, DSC-Vol. 58, November 1996, pp. 299-306.

[182] Lei Zhou, Elke A. Rundensteiner, and Kang G. Shin, "Schema Evolution of an Object-Oriented Real-Time Database System for Manufacturing Automation,"

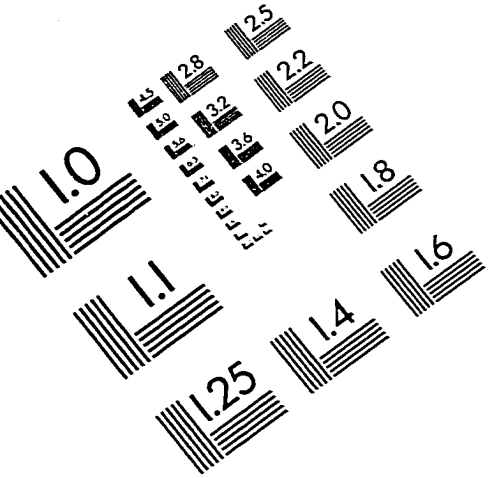
IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 6, November/December 1997, pp. 956-977.

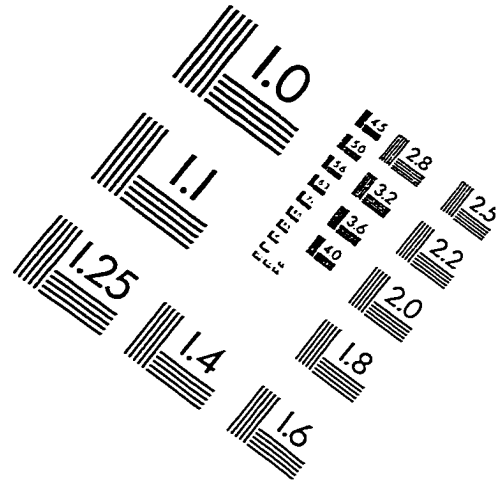
[183] Lei Zhou, Kang G. Shin, Elke A. Rundensteiner, and Nandit Soparkar, "Probabilistic Real-Time Data Access with Interval Constraints," *Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications (RTDB'96)*, March 1996, pp. 15-22.

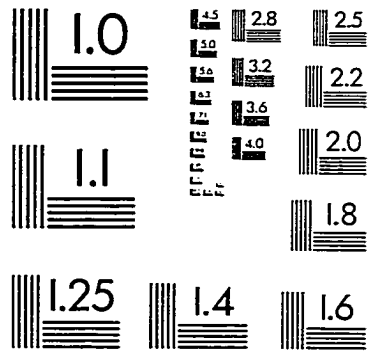
[184] Lei Zhou, Kang G. Shin, Elke A. Rundensteiner, and Nandit Soparkar, "Probabilistic Real-Time Data Access with Deadline and Interval Constraints," in Sang H. Son, Kwei-Jay Lin and Azer Bestavros ed., *Real-Time Databases Systems: Issues and Applications*, Kluwer Academic Publishers, 1997.

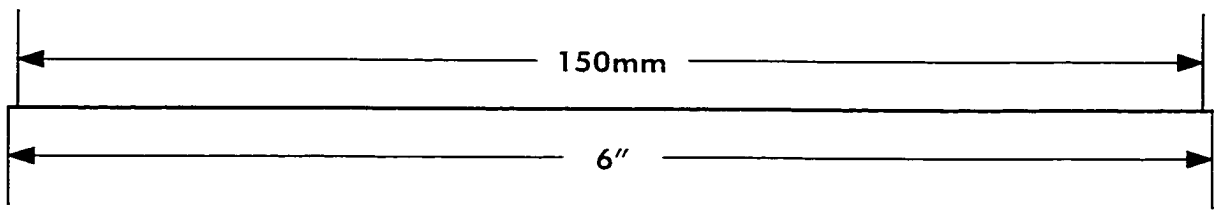
[185] Lei Zhou, Kang G. Shin, and Elke A. Rundensteiner, "Rate-Monotonic Scheduling in the Presence of Timing Unpredictability," *Proceedings of Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

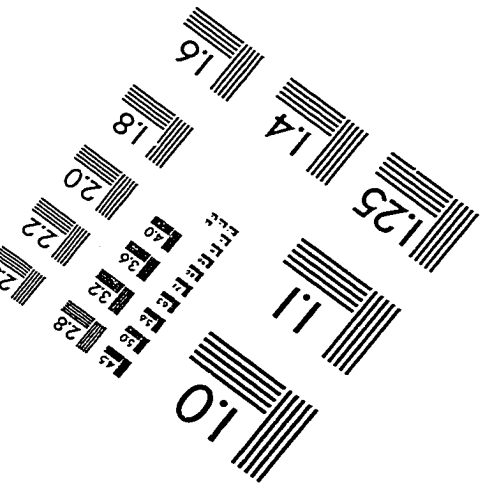
IMAGE EVALUATION TEST TARGET (QA-3)

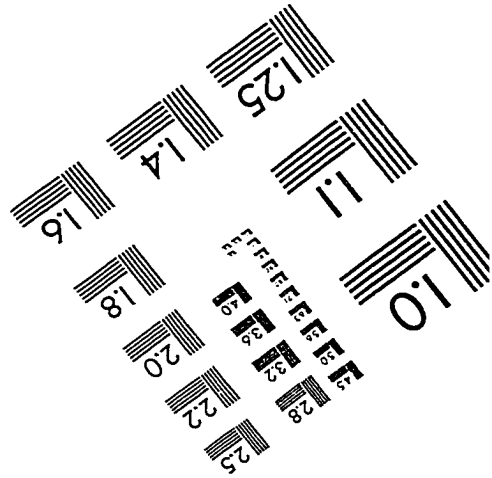












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989