Profiling of COTS Real-Time Development Environments: An Analysis of the LabVIEW Development Environment with the Real-Time Module

by

Mahdi Javer

A thesis submitted to
The Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
February 13, 2007

Your file Votre référence
ISBN: 978-0-494-26993-0
Our file Notre référence
ISBN: 978-0-494-26993-0

Abstract

As real-time embedded systems become more complex and development times shrink to meet market demand, developers are turning to commercial off the shelf (COTS) development tools. While these tools assist the developer by abstracting the details of implementation, the execution overheads introduced by the real-time operating system and the development environment are not known. This dissertation introduces the LabVIEW Real-Time Cost Model, which extends rate monotonic analysis to provide the developer with assistance in determining whether or not a set of tasks is schedulable. Required input from the developer includes the periods and execution times of the user tasks. Execution costs of the development environment are included in the model, allowing for more confidence that the tasks will be schedulable. The model's accuracy was verified using both simulated test as well as an industrial application used for control of a Captive Trajectory Simulation (CTS) system.

Acknowledgements

First of all, I would like to thank my supervisors Prof. Pearce and Prof. Ahmadi. Their positive support and feedback kept me motivated even during the times of frustration.

This research would not have been possible without the assistance of NRC-IAR and NI. I would like to acknowledge the effort put forth by Mathieu Gibeault and Norman Tang for allowing me to use the facilities at the NRC to conduct my research. Additionally, Nicolas Ricciardi and the support staff at NI have been instrumental in providing the necessary support to learn the intricacies of the LabVIEW Tool.

I also want to thank my parents for all their moral support. They always ensured I had a roof over my head and never went hungry.

I would like to thank all of my friends at Carleton (Laurence, Andrew, Matthew, and Megan) for encouraging me to skip class to play pond hockey, taking me out at the knees during hockey, or throwing empty beer glasses at me. You made sure that my time in school remains unforgettable.

Finally, I want to especially thank Farin Manji for her excellent work in editing this dissertation.

Table of Contents

List of Figures

List of Tables

List of Acronyms

CMU	Carnegie Mellon University
COTS	commercial off the shelf
COTS-RTP	COTS Real-Time Profiling
CTS	Captive Trajectory Simulation
DAQ	data acquisition
DMA	direct memory access
DOF	degree of freedom
DSP	digital signal processing
IAR	Institute for Aerospace Research
LCM	least common multiple
LRCM	LabVIEW Real-Time Cost Model
NI	National Instruments
NRC	National Research Council Canada
PXI	PCI eXtensions for Instrumentation
RMA	rate monotonic analysis
RTOS	real-time operating system
SEI	Software Engineering Institute

UI	user interface
VI	virtual instrument
WCET	worst case execution time

Chapter 1

Introduction

The design and development of real-time embedded systems is a very complex task. Not only do the applications have to be functionally correct, but they also need to be able to meet strict timing constraints. The introduction of timing, compounded with the interaction between the inputs and outputs of the system, results in a challenging problem, often difficult to solve. The difficulty does not stem from determining the timing requirements alone, but in determining with confidence that the application will actually meet all the timing requirements.

The recent advancements in technology of both hardware and software have done very little to deal with the issue of time in real-time embedded systems. Rather, most of the advancements have done more harm than good [1]. Hardware designers have added pipelining, on-board cache, branch predictors, and many other mechanisms in order to boost the performance of processors. However, this increased performance has come at the expense of predictability of run-times. The randomness of cache updates and back-tracking of execution, when incorrect branch predictions are made, have made it difficult to determine the execution times of applications.

Theoretical research in the area of real-time systems analysis dates back to the early part of the 70's for predicting the worst-case timing behaviours. However, this theory has not been fully implemented in practice. Programming languages widely in use, such as C, C++, and Java lack the necessary functions to represent timing or deadlines. Instead, they tend to focus on issues such as abstraction, code reuse, and memory management. The introduction of garbage collectors, that run at random intervals in order to clean up free memory, results in lack of predictability. While these features have benefited developers, the implementation of these features often hinders the ability to use theory in order to perform timing analysis.

Many real-time developers have stayed with lower level languages such as C and Assembly. Most often this is not done to optimize code, but rather to ensure that the code runs in a predictable manner. If the code does not run in a predictable manner, the worst case execution time (WCET) becomes difficult to determine, therefore making it difficult to ensure that timing deadlines can always be met.

As more applications become embedded, non-software developers are forced to play the role of real-time software developers; this is often done in the field of control systems. In the past, the design of control systems was typically performed using a hardware-only solution and was often restricted to the application at hand; any changes to the system would result in costly upgrades to controllers. As companies make the shift to software based solutions, which are often easier to upgrade, most of the work is still assigned to the person who is in charge of the original hardware design, who may have very little to no software experience.

The shift from hardware to software solutions coupled with a more condensed time period dedicated to marketing theses products has led to developers progressively using tools that provide assistance in dealing with timing issues. This has led to the

increased use of commercial off the shelf (COTS) products that can abstract some of the complexities and allow for rapid design and development. A sector leading this movement is the consumer industry; they place the latest technologies in the hands of demanding consumers. The automation and control sector is also rapidly enhancing the services they provide due to a demand created by companies wishing to upgrade their systems at minimal cost.

The use of COTS tools does not come without drawbacks. As underlying details of the implementation of real-time systems are hidden from the developer, it results in a loss of control over the system. There are also hidden costs introduced by the development environment and real-time operating system (RTOS). Lack of knowledge of these costs can lead to unexpected results especially when attempting to ensure that deadlines will be met. However, even with the numerous drawbacks of the COTS tools, the loss of competitiveness in the market is often a greater threat. Thus, certain levels of inadequacy are often accepted as long as the final product operates within tolerable measures of failure.

Development tools such as MATLAB/Simulink and LabVIEW are widely used in the design and development of real-time embedded control systems. They provide simple visual programming interfaces that have the ability to create an application simply by connecting functional blocks. They also have the added benefit of abstracting the implementation details from the user, allowing someone with little to no programming experience to quickly build an application from scratch. The toolkits included in these packages are often referred to as providing "real-time" support; however, the approaches used do not always guarantee that the resulting system meets timing constraints. Further research needs to be done to align these tools with the current state of research in real-time systems.

In addition to the development environment, the runtime platform also plays a role in the prediction of real-time behaviour. The individual contributions and interactions between the hardware, operating systems, middleware, and compiler optimizations can all affect the real-time behaviour of a system. Often, the details of the runtime platform do not provide the necessary information to perform the analysis of real-time behaviour. Details of the hardware can be determined from documentation; however, cache and branch predictors are dynamic and require very low-level analysis to accurately determine runtime behaviours. Additionally, third party software is often treated as a black box, as the inner details and interactions are unknown. Due to the overall lack of system knowledge by the developer, the analysis of practical engineering development using current research on real-time systems has been very slow in taking a foothold.

The long-term goal of the COTS Real-Time Profiling (COTS-RTP) research project is to advance the state of the art in practical engineering methods. This can be done by aligning industry-calibre tools with research theory. The main goal is to create a generic model that characterizes all the effects and overheads introduced by the development environment and runtime platforms; the model can be applied to an application as it evolves. As the application is updated or changed, the parameters of the model only need to be adjusted to suit the new system.

This research focuses on modelling the characteristics of the LabVIEW development environment with the LabVIEW Real-Time Module. The results include the creation of the LabVIEW Real-Time Cost Model (LRCM) which is used to assist a developer in determining if an application can meet its deadlines. Hidden costs of the development environment are taken into account in the LRCM. The model gives the developer more confidence that all the hard real-time deadlines of the application

can be met.

The LRCM is used to predict if a set of tasks can be scheduled to meet their deadlines. By providing the execution times and periods of the user tasks, the LRCM can predict if the tasks will be schedulable using rate monotonic criteria. Tests conducted on the LRCM shows that predictions made about the scheduling of user tasks on the system has a 3.84% average error versus the 21.82% average error when using RMA prediction alone.

LabVIEW is chosen as the system to model based on its broad use in industry for the design and development of control systems. In addition to the LabVIEW development environment, National Instruments (NI) [2] also provides a large variety of hardware to support embedded systems. The toolkits seamlessly integrate the hardware and software, making them a popular choice in a variety of applications. Various analog-to-digital and digital-to-analog hardware I/O boards are available and supported in the LabVIEW environment. The effective use of these boards typically involves performing cyclic I/O and conversion at regular intervals. This demand for regular servicing of hardware I/O boards often leads to the periodic handling and processing of data buffers. The periodic nature of the work aligns well with research theories such as the *rate monotonic* approach, making it a good fit for analysis.

Research, based on the use of the LabVIEW tool, is currently being conducted for a number of projects at the Institute for Aerospace Research (IAR) of the National Research Council Canada (NRC). The projects include the implementation of a Captive Trajectory Simulation (CTS) system, as well as an ongoing study dealing with the feasibility of using LabVIEW to control the operation and data collection in a high-speed wind tunnel. The CTS system includes a robotic platform with eight degrees of freedom installed inside the wind tunnel. Such a system requires a high

degree of reliability due to the safety requirements of the application. The robot control involves real-time joint control loops at the low-level and extensive computations at the high-level. A model, such as the LRCM, is a useful tool to increase confidence that all timing requirements can be met.

The following sections of this document are structured as follows. Chapter 2 provides a brief overview of background information pertaining to this research. It defines the concepts of real-time embedded systems and a real-time operating system. An overview of LabVIEW with the Real-Time Module and a description of the CTS system being developed at the NRC-IAR is also provided.

Current research being conducted in the analysis of real-time systems is reviewed in Chapter 3. This includes an introduction to rate monotonic theory and its underlying assumptions, as well as further research performed to extend the theory by relaxing these assumptions. Work done on applying theory to practical applications, as well as current research on the LabVIEW development environment are outlined.

In Chapter 4, an analysis of the state of the art research introduced in Chapter 3 is conducted in order to highlight issues raised when applying rate monotonic theory to practical applications. The Chapter also includes the statement of the thesis as well as the proposed solution. Finally, the scope of the research and contributions of the work are discussed.

Chapter 5 provides a guideline for creating a model based on rate monotonic analysis. This includes a breakdown of the characteristics of the development environment that must be identified before a model is created. A full description of the real-time platform being analyzed is followed by the presentation of the LabVIEW Real-Time Cost Model. The chapter concludes with measured values of overheads used by the LRCM.

The testing of the LRCM is covered in Chapter 6. This includes a description of all the tests conducted and their results. An analysis of the test results, including a discussion of the drawbacks of the current model is also covered.

Chapter 7 concludes with final thoughts regarding the research and provides direction for future advancements.

Chapter 2

Background Information

This chapter provides the framework to enable readers with varied experiences to understand the main principles behind this research. The terms and concepts used throughout the rest of this document are defined. Section 2.1 begins by providing the definition of a real-time embedded system, as well as a describing some of the challenges that are encountered when dealing with time. In Section 2.2, the concept of a real-time operating system is introduced. The components of the RTOS used to address the timing constraints of real-time applications are also described. Section 2.3 provided a brief introduction to LabVIEW and the LabVIEW Real-Time Module. Important features of the tools used in real-time development are also highlighted. The final section describes the CTS application that is currently under development at the NRC-IAR.

2.1 A Brief Introduction to Real-Time Embedded Systems

An embedded system is a computer system which exists as a component of a larger system. Applications running on the computer system are restricted to performing functions pertaining to the larger system. In the case of a robotic system, a computer system that is used solely for the control of the robot is called an embedded system. Other examples of embedded systems are the computers found in cell phones, PDA's, and digital music players.

A real-time system introduces the concept of time into an application. Not only must the program be functionally correct, but it must also produce results in a given time interval, in order to be deemed correct. The COTS-RTP project only considers applications that have "deadlines" that must be met, in order for the results to be deemed correct.

Real-time applications are typically broken down into a set of *tasks*. Each task provides a function that can be executed on the system. These tasks are often invoked at regular intervals. At each invocation, the task senses the state of the system, performs some computation, and may also send commands to change or display the state of the system [3]. Using the example of an automobile, one task may check the anti-lock braking system to see if action needs to be taken if the wheel gets locked, while another task may check to see if moisture has appeared on the windscreen and automatically turn on the wipers.

The main difference between a real-time task and a non-real-time task at the process level is that the real-time task has a deadline by which it must complete the task. In critical applications, results produced after the deadline are not only late, but

also wrong [4]. Applications of real-time systems include nuclear power plant control, flight control, space mission control, robotics, industrial automation, and automotive systems. A missed deadline in one of these applications can have catastrophic results.

A common misconception among many in both academia and industry is that real-time means real-fast. Although there are applications where speed of computation is of the essence, running an application as fast as possible still does not guarantee that its deadlines will be met. While the purpose of real-fast computation is to minimize the average response time of a set of tasks, in real-time computing the goal is to ensure that the stringent timing requirements of each task can be met [5]. A real-time system needs to be predictable in order to guarantee that the application deadlines can be met.

The concept of time in a real-time system is also dependent on the environment. In a household temperature control system, reaction to an increase in temperature by perhaps turning on an air conditioning unit may have a response time of a matter of seconds. If this temperature control system were part of a nuclear plant, the response time in seconds may not be sufficient. When developing control systems for automobiles versus airplanes, the response times may also be different due to the speeds at which they travel. Hence, in the design of a real-time embedded system, the physical system must also be taken into account.

Real-time systems are typically broken down into two categories: hard real-time and soft real-time. In a hard real-time system, any failure to meet a deadline is considered unacceptable. Recovery from such a failure is either very difficult or impossible. Typical application examples include controlling a nuclear power plant, or an airplane. Any failure in these applications may lead to the loss of life. In a soft real-time system, occasional failure is acceptable and the system is designed to re-

cover and continue execution. Late results may still be useful but their significance decreases over time. An example would be a household temperature control system where late sensing of a temperature value would not have a drastic effect, as long as subsequent readings occurred on time. The focus of the COTS-RTP project is to provide confidence that all the deadlines of a hard real-time system can be met.

In order to ensure that hard real-time systems can meet their deadlines, the system must be predictable. To tackle the predictability issue, developers often make use of a real-time operating system (RTOS). The next section introduces the concept of a RTOS and the features included to address predictability.

2.2 Real-Time Operating Systems

An operating system is a software-based interface between the application software and hardware of a computer. The core of the operating system is referred to as the *kernel*. The kernel provides the basic features of memory management, hardware interfacing, and task scheduling.

A RTOS is a class of operating system that is used in real-time applications [6]. The RTOS is designed to respond to an event as quickly as possible in a *deterministic* manner. In a deterministic system, given a particular input, the output will consistently provide the same result. Thus, a deterministic system allows for predictable results. In real-time systems where timing correctness is just as important as logical correctness, the response time of the RTOS needs to be known in order to determine the timing correctness of an application. Thus, response times are typically provided as worst case values by the RTOS vendor. It is important to note that using a RTOS does not guarantee an application will be real-time. The RTOS only provides facilities

that, if used properly, can guarantee that deadlines can be met deterministically.

When using a RTOS, application tasks are often implemented as *threads*. The thread is a basic unit of work managed by the operating system. Its execution is managed by making use of a specialized scheduling algorithm. The developer can make use of the scheduling algorithm when designing an application to ensure deterministic behaviour in the final system. The following sections provide a brief overview of how the scheduling of tasks (or threads) is commonly handled by the RTOS.

2.2.1 Priority Assignment

Most RTOSs allow the developer to assign a *priority* to a task or thread. The priority of a task is a ranking system that allows the RTOS to compare the importance of each task relative to another task running on the system simultaneously. This provides a mechanism for the RTOS to determine when and how resources will be allocated to each task, including allocation of the processor. Different methods of assigning priority range from random assignment to assignment based on some heuristic. Common methods used to assign priority include "rate monotonic" assignment where the priority is assigned based on the period of the task (further described in Section 3.1.1), as well as "earliest deadline first" which assigns highest priority to the task that has the earliest deadline.

The assignment of priority can be either static (i.e. fixed) or dynamic. In static priority assignment, the priority of the task is set either during design time, or when the application is first launched. The priority of the task is then fixed throughout the execution of the application. Dynamic priority assignment on the other hand, allows the priority of a task to be changed during run time. Rate monotonic assignment uses a static priority assignment scheme, while earliest deadline first uses a dynamic

priority assignment method.

2.2.2 Scheduling Methods

The determination of which task is allocated the processor at any given time is done by making use of a scheduling algorithm implemented by the RTOS *scheduler*. The scheduling of a task can occur either when the task is released, or during a *context switch*. A context switch occurs when the processor switches from running one task to running another, hence changing the context of the task running on the system altogether. The scheduling algorithm decides when each task will be allocated the processor and maintains order in the system. A number of different scheduling algorithms are commonly used by RTOS vendors and are described below.

Scheduling of tasks can be implemented in either a priority based method, or by assuming all tasks have the same priority. In the case where all tasks have the same priority, scheduling is often done in a round-robin fashion. The processor time is divided into equal time slices, and each task gets a slice of time. The task that has been allocated the processor will run until the time is expired, or until it completes its execution, whichever comes first. Once all tasks have had a chance to run, a time slice is once again allocated to each of the tasks that have not completed yet. This continues until all tasks have completed their execution.

Scheduling tasks based on priority can be further divided into two categories based on whether *preemption* is allowed by the RTOS. When preemption is available, a task of higher priority that arrives while a lower priority task is running is allowed to take the processor away from the lower priority task (i.e. preempt its execution). In the case of non-preemptive scheduling, when the processor is free, the task that currently has the highest priority is allocated the processor and is allowed to run

until completion. If a higher priority task arrives while the processor is running another task, the higher priority task must wait until the running task completes its execution before obtaining the processor. Allocation of the processor to the highest priority task, that is ready to run when the processor is free, continues until all tasks are complete.

In the case of preemptive priority based scheduling, the highest priority task that is ready to run will always be the one that is running on the system at any given time. While running the highest priority task, if a task of even higher priority arrives, the currently running task is preempted so that the newly arrived higher priority task can be allocated the processor. Once the highest priority task that is ready to run is completed, the next highest priority task is then allowed to run (unless preempted by a newly arrived higher priority task) until all tasks are complete. This scheduling mechanism will work for both static and dynamically allocated priorities, but in the case of dynamically allocated priorities, the scheduler must also be aware of changes in priority and not just newly arrived tasks.

In the next section, the LabVIEW development environment and the additional Real-Time Module are described. This includes details of the priority allocation methodology and scheduling algorithms implemented on the system.

2.3 The Basics of LabVIEW and the LabVIEW Real-Time Module

LabVIEW is a graphical development environment commonly used in industry for the development of control system and test platform applications. In addition to LabVIEW, National Instruments also provides hardware modules for data acquisition

and I/O. These modules can be seamlessly integrated into an application by using the supplied libraries.

Originally, LabVIEW was not designed with support for hard real-time applications. Applications were run on non-real-time operating systems which allowed other applications to be run at the same time, causing contention for the processor. With the introduction of the Real-Time Module, the applications are now able to run on hardware platforms (referred to as a *real-time target*) which run a RTOS. The LabVIEW Real-Time Module also introduces the functionality to allow the developer to create tasks that are released periodically. With the extended features of the Real-Time Module, developers can create real-time applications.

In the sections that follow, a brief introduction to the LabVIEW development environment will be provided. This includes a high-level description of the programming environment, as well as an overview of the features of the Real-Time Module to support development of real-time applications.

2.3.1 LabVIEW

Applications created in LabVIEW are referred to as a virtual instrument (VI). Each VI is made up of a user interface (UI) front panel and block diagram. A VI can also be called from another VI, and is referred to as a sub-VI. Included in the LabVIEW package are various VIs in the form of programming functions and device drivers. By using the supplied VIs, a developer can rapidly create applications with minimum effort.

LabVIEW uses a dataflow model for programming [7]. In this model, each VI is created by dragging tools or objects either onto the front panel or block diagram. Each object or tool on the front panel has a corresponding component in the block

diagram. Graphical representations of functions are then added to the block diagram to control the front panel objects. Wiring these blocks together creates a flow of data. When the VI is run, the flow of the program does not necessarily flow from left to right in a block diagram. A node (object or function) on the block diagram will only execute when all of the data arrives at its inputs. Once the node completes its execution, output from the node is passed to the input of the next node in the dataflow path. If two nodes are not connected by any wires (or data path), they create a separate dataflow, and can be run independently or in parallel. This is different from text-based programming languages like C and C++, which follow the control flow model where programs are run sequentially. Due to the lack of sequencing in dataflow programming, the final implementation of the compiled code is unknown to the developer. Figure 2.1 shows a sample VI with the front panel and block diagram windows. The different data paths as well as the blocks on the front panel with corresponding components on the block diagram are highlighted.

Each VI in an application runs in a separate thread. The only exception is a subroutine (a special kind of sub-VI) which runs within the calling VIs thread. The priority of a VI can be set to one of five different levels ranging from background priority (lowest) to time-critical priority (highest). The priority can only be set during development, and is fixed at run-time (i.e. static priority assignment). Scheduling of threads is done in a priority based preemptive manner based on the assigned priority of the VI running in the thread. If two threads have the same priority, they are scheduled in a round-robin fashion.

A VI can be created to run periodically by the use of the Wait or WaitUntil function within a while loop. Adding a Wait or WaitUntil function in the VI will cause it to yield the processor until the specified time has elapsed. It can then

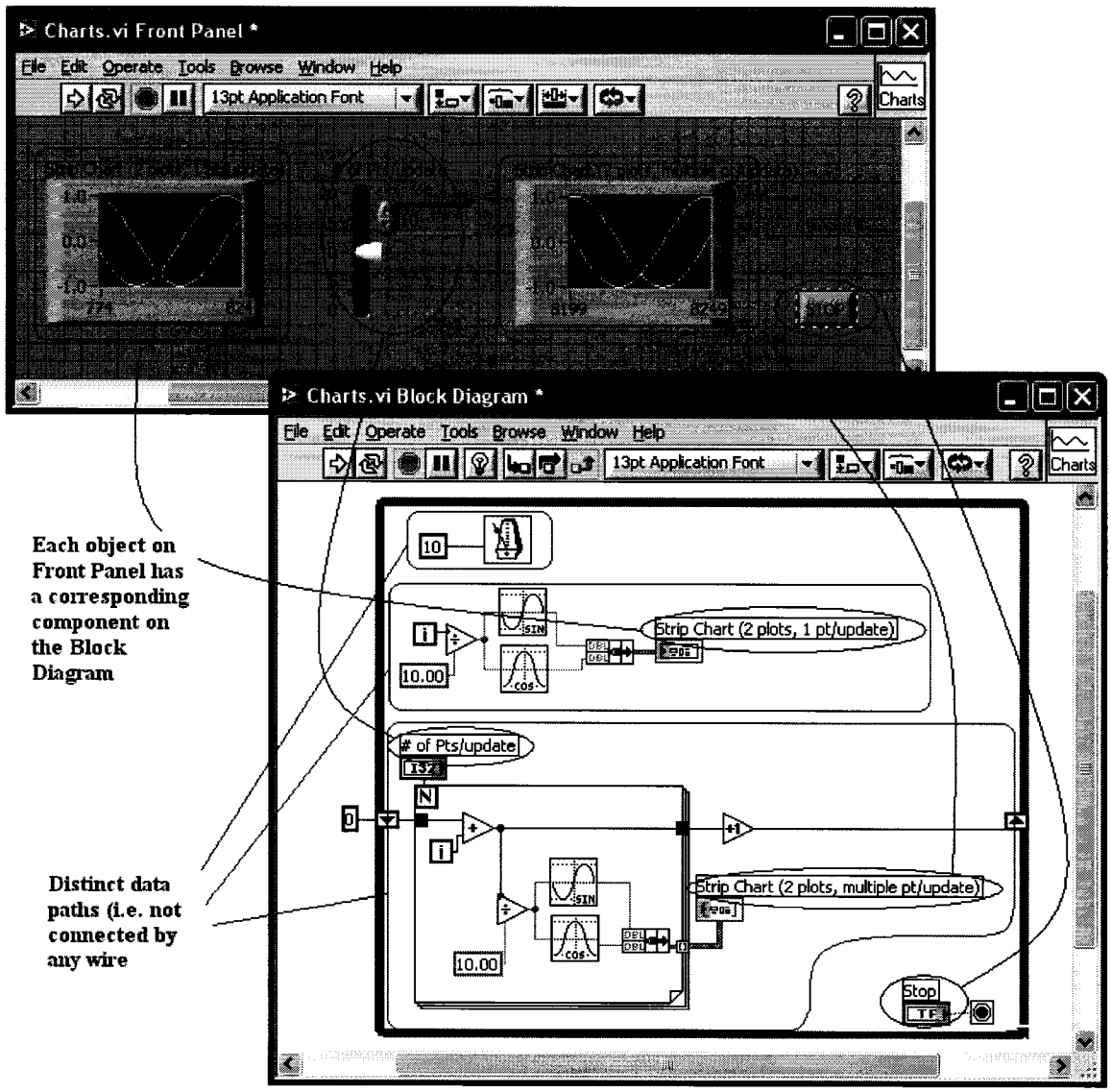

Figure 2.1: VI front panel and block diagram with multiple data paths.

continue the execution of the while loop. It is important to note that placement of the Wait or WaitUntil function in the block diagram can be done in sequence to the rest of the code, or in parallel. The placement of a Wait or WaitUntil function will influence when the wait actually takes place in the code, and will also influence the timing of the loop. Additional information on creating software timed loops can be found in NI's online help pages [8].

2.3.2 The LabVIEW Real-Time Module

The Real-Time Module was added to extend the capabilities of LabVIEW to meet the need for deterministic real-time performance [9]. The Real-Time Module makes use of a real-time target platform (a hardware platform available from NI) which runs the Phar Lap RTOS developed by Ardence [10]. Deterministic behaviour of the application is enforced by not allowing user interfaces to be created on the target platform. Instead, all user interactions are handled by creating a periodic thread to service interactions between the real-time target platform and a host system that runs the actual user interface. Applications can also be created without a UI by setting all objects on the front panel to hidden which results in the periodic UI thread ignoring the front panel objects. For applications that require some user interaction, the periodic thread on the real-time target platform allows for creation of deterministic applications that can handle user interaction which is non-deterministic by nature.

Figure 2.2 shows the relationship between the NI development software and real-time hardware versus other COTS development and hardware packages. In the more traditional development environments, each part of the development tool is broken up into its individual parts: compiler, linker, debugger, and analysis tools. In the LabVIEW model, each of these individual pieces are encapsulated into a single tool.

The entire process used to convert the application from the block diagram to a running application on the target platform is abstracted from the developer. Hence, the final implementation of the code is unknown to the developer.

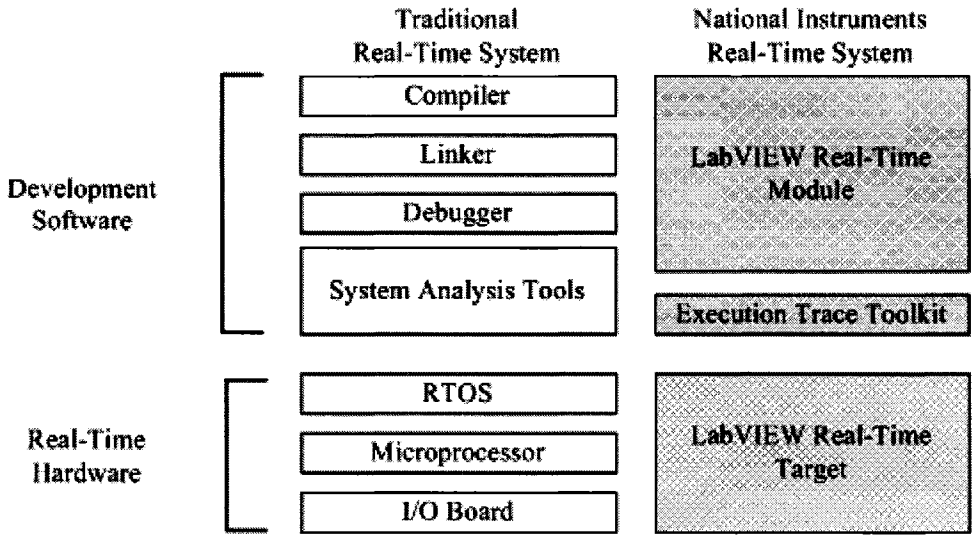

Figure 2.2: NI vs. traditional real-time development solutions.

In addition to the Real-Time Module, a toolkit that provides assistance in the analysis of timing characteristics is also offered. The LabVIEW Execution Trace Toolkit allows the developer to capture timing and execution data on VI and thread events running on the real-time target platform [11]. The information gathered by the Execution Trace Toolkit can be used to determine the timing characteristics of the application.

The Real-Time Module introduces the Timed Loop function which is used to create tasks that are executed deterministically. Each Timed Loop runs in its own thread in an application. They can be set up to either run at a periodic interval, or be triggered by an external hardware interrupt. Each Timed Loop can have an individual priority set at a value ranging from 1 (lowest) to 2,147,480,000 (highest).

The number of Timed Loops that can be created on the system is restricted to 128, thus the number of distinct priorities that can be assigned is effectively reduced to 128. The Timed Loop priorities fall between the Time Critical priority (highest) and High priority (second highest) levels of VIs. Although the Timed Loop is created within a VI, once execution begins, each Timed Loop runs in its own thread. The priority of the Timed Loop is assigned and will not be affected by the priority of the VI in which it was created. The scheduling of Timed Loop threads is done in a priority based pre-emptive manner similar to that of VI threads.

Additional features of the Timed Loop include options to offset the initial start time and handle tasks that start late (after their deadline). The developer can decide how to handle the late execution of a task by setting the mode of the loop. Two mode options, which can be individually enabled or disabled, are available for each Timed Loop. The first option is to discard missed periods. By selecting this option, an instance of a task that obtains the processor after its deadline has passed will not be executed. The second option allows for the selection of maintaining the original phase (i.e. continue releasing the thread based on the original schedule). Deselecting this option will cause the scheduler to create a new schedule so that new iterations of the loop will be released at multiples of the period starting from the current time. If all of the Timed Loops in an application need to be synchronized, the second option should be selected. The combination of the mode options allows for four distinct methods for handling both the initial start times as well as tasks that start late.

The periodic release of each task is achieved by making use of a timing source. Due to LabVIEWs close interaction with data acquisition (DAQ) hardware modules, both internal CPU timing and external timing from a DAQ module are available. When using the internal CPU timing, individual timing sources can be created for

each Timed Loop, or they can share a timing source. By sharing a timing source, the release time of all of the tasks can be better synchronized. The internal CPU timing source on the real-time target platform allows the period, and the offset of the Timed Loops to be set in units of microseconds.

The main objective of using a Timed Loop is not to execute code faster; it is to offer better determinism in real-time applications. By properly setting the priority, period, offset, and mode of the loop, an application can be created such that all tasks will meet their deadlines. All Timed Loops can also be synchronized to start at the same time by using the SynchronizeTimedLoopStarts VI. This VI ensures that the Timed Loops start at the same time and use the same timing source. The offset parameter of the Timed Loop can be set to prevent all of the loops from starting at the same time, while maintaining their schedules relative to one another. There is also a StopTimedLoop VI which can be used to stop a loop, allowing for a clean exit from an application.

Attributes of the Timed Loop such as priority, period, and mode, can be changed during run-time from within the loop. All changes made will take effect on the next iteration of the loop. Additionally, run-time parameters relating to time can be read from within the loops. These parameters include the actual start time of the loop, the end time of the previous release of the loop, and a "Finished Late" flag which indicates if the previous iteration of the loop finished its execution after its deadline. Further information on the setup and use of Timed Loops can be found in [12].

2.4 An Overview of the Captive Trajectory Simulation (CTS) System

The motivation for selecting LabVIEW as the development platform for analysis stems from its use in industry for control applications. With the drastic increase in computer speeds today, the aerospace industry is experiencing a shift from implementing custom real-time platforms to using COTS solutions. This shift is occurring in air-borne vehicles as well as in the ground test facilities used in the development and testing of systems. Due to the nature of these applications, real-time performance is of great importance in the final product.

An example of such an application is currently being developed at the Institute for Aerospace Research of the National Research Council Canada's supersonic test facilities [13]. The LabVIEW development environment with Real-Time Module is being used to control a redundant robotic manipulator intended to manipulate instrumented aerodynamic models inside a wind tunnel with transonic flow conditions.

A Captive Trajectory Simulation system is used to emulate the trajectory of a *store* (any object released from an aircraft) to investigate its safe clearance from an aircraft [14, 15]. The CTS consists of an eight degree-of-freedom (DOF) robotic manipulator with aerodynamically shaped links suitable for operation inside a wind tunnel. A model of a store is attached to the tip of the robot and follows a desired trajectory expressed by both position and orientation. Aerodynamic forces on the store are measured using a sensitive 6 DOF balance (force sensor). The force and moment information are then fed to the store motion simulation program in order to compute the new position, velocity and acceleration. The trajectory is taken as the desired path to the robot controller. The overall CTS control software requires that extensive

computation be completed by several tasks before the execution of each control period; hence the project can benefit from the results of this research. Figure 2.3 shows a model of what the CTS system would look like inside the wind tunnel. A model aircraft is bolted to a mounting bracket on the side of the tunnel, while the store is attached to the tip of the robotic manipulator and it can be positioned according to the test being conducted.

Due to the stringent safety requirements imposed by the application, a higher degree of reliability is required compared to other industrial robots. Control of the robot is broken up into various tasks including trajectory generation, collision avoidance, inverse kinematics, and the control of each individual joint. Due to the redundant joints in the robot, the inverse kinematics (determining the positions of each joint when given the position and orientation of the end-effector) can become computationally expensive. The robot will also employ advanced control algorithms to synchronize the motion of all of the joints in order to minimize deviations of the tip from the desired path. This will require the control algorithm to run at a rate in the hundreds of Hertz and be to deterministic in nature. Ensuring the control algorithm can meet these requirements will be vital to the success of this project.

Due to the high cost of the precision parts used in the robot manipulator, the need for proper design and analysis of the system is important to ensure that no damage is done to either the robot or wind tunnel. The safety of the system depends on the control software working correctly. A model such as the LRCM would play a useful role in determining if all task deadlines can be met, and this would result in the safe deployment of the system.

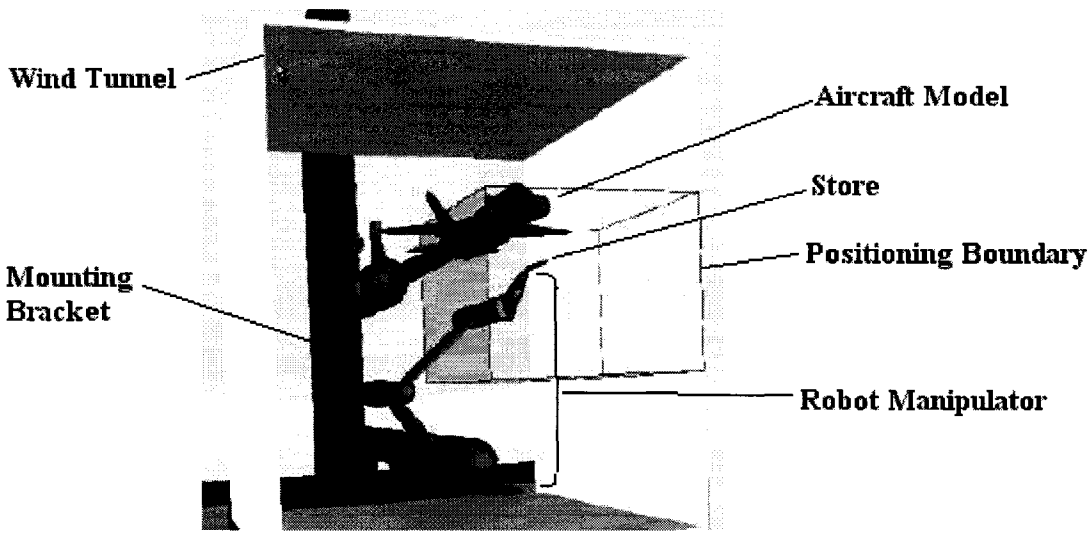

Figure 2.3: CTS platform installed in the wind tunnel.

Chapter 3

State of the Art

The purpose of the COTS-RTP project is to better align scheduling theory with real-time software development using COTS tools. This chapter provides information about current research being conducted in the area of real-time embedded systems. In Section 3.1, research in the area of fixed priority scheduling is covered. This includes theory on rate monotonic analysis (RMA), which is further extended by time demand analysis. Current work on the relaxation of the RMA assumption that platform overheads are negligible is also provided as a basis for further examination conducted by the COTS-RTP project.

Section 3.2 outlines a technique for the estimation of system overheads. It provides a more realistic approach to measuring the context switch costs introduced by the RTOS when the inner workings are not known. This leads to Section 3.3 which covers work performed on applying theoretical research to practical systems. An analysis of a number of different industries, where work is being performed to bridge the gap between theory and practice, is provided. Finally, recent work done on the LabVIEW tool to determine real-time performance is discussed in Section 3.4.

3.1 Fixed Priority Scheduling

In fixed priority scheduling, the priority of each task is set once and not changed during execution. Rate Monotonic theory was introduced by Liu and Layland [16] for fixed priority real-time systems analysis. Their paper includes the introduction of Rate Monotonic Analysis (RMA) to predict whether an application will meet its deadlines. The theory is based on several underlying assumptions, and contributions have been made by others to relax these assumptions. This section summarizes the RMA theory introduced by Liu and Layland. Further work on fixed priority scheduling by using timed demand analysis is also introduced. Finally, work done by others to relax the negligible platform overhead assumption made by RMA is provided.

3.1.1 Rate Monotonic Analysis

The Rate Monotonic Analysis (RMA) theory for worst-case execution time (WCET) is based on a simple periodic tasking model for real-time applications. A task is a software block that must be scheduled to occur with a regular period in the execution of an application. Each task ($task_i$) is characterized by its period (p_i), the processor execution time required to complete the task (c_i), and the deadline of the task relative to its periodic release (d_i). In the RMA scheme, priority assignment is static, and is based on the period of the task. Highest priority is given to the most frequent work, and decreasing priorities are assigned to tasks with increasing periods. The resulting priority assignment is "Rate Monotonic" due to the monotonic relationship between each task's priority and its periodic frequency.

In applying RMA theory to an application, tasks are often implemented as *threads*

under the management of a RTOS. The threads are assigned static rate monotonic priorities and the RTOS schedules the threads to execute at regular timed intervals, i.e. it releases each task periodically. Scheduling is done in a priority based preemptive manner (as described in Section 2.2.2). Thus, the highest priority task that is ready to run will always be the one occupying the processor at any given time.

The original RMA theory is based on the assumptions described below. While the assumptions may seem simplistic, research since then has further developed the theory to relax the assumptions.

Assumption 1: All tasks are periodic. This may not seem practical for some applications, but it does fit nicely with many control applications.

Assumption 2: Task deadline = period (i.e. $d_i = p_i$). The model assumes that each task must be completed before the next release of the task.

Assumption 3: Tasks are independent. Independent tasks do not require synchronization or inter-task communication.

Assumption 4: Each task's processing requirement is constant. This assumption allows the processor utilization of each task to be represented by a single value.

Assumption 5: Platform (hardware, compiler and middleware) overheads are negligible. This assumption is needed to obtain a platform independent theory, and is a major impediment in practical applications of RMA.

RMA uses processor utilization as the metric for guaranteeing whether an application will meet its timing constraints under worst case loading. $Task_i$'s processor utilization is the percentage of time that the processor spends executing $task_i$ (i.e. c_i/p_i). Liu and Layland proved that a system of n tasks will meet its deadlines under worst case loading when:

$$\sum_{i=1}^{n} \frac{c_i}{p_i} = n(2^{1/n} - 1) \tag{3.1}$$

The left side of the relation is the total processor utilization of all of the tasks. The relation is pessimistic, i.e. it is sufficient, but not necessary, and therefore there may be conditions where the relation does not hold but the system will still meet its deadlines. Time demand analysis based on recurrence relations have been proposed to address such cases [17]. The next section provides further explanation of time demand analysis.

3.1.2 Time-Demand Analysis

Time demand analysis is an iterative technique that can be used to converge on the response time of tasks to decide whether the tasks meet their deadlines under worst case loading. The worst case loading of a system occurs when $task_i$ is started at a *critical instant.* The critical instant is defined as a point in time where a release of $task_i$ will have the longest response time. This is shown by Liu and Layland to occur when $task_i$ is released at the same time as all other higher priority tasks (denoted as time t_0).

To perform the analysis, each task is considered individually starting with the highest priority task and going in order of decreasing priority. In order to determine if $task_i$ is schedulable (after determining that the tasks with higher priorities are schedulable), an instance of $task_i$ that is released at a critical instant t_0 must be shown to meet its deadline. This requires considering the total processor demand by $task_i$ and all other higher priority tasks by:

$$w_i(t) = c_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil c_k \text{ , for } 0 < t \leq p_i \tag{3.2}$$

$Task_i$ will meet its deadline if at some time (t) between the critical instant $(t_0 = 0)$ and the task's deadline (p_i), the supply of processor time (t) is greater than or equal to the demand for processor time $(w_i(t))$ (i.e. $w_i(t) \leq t$, for some time t where $t_0 < t \leq p_i$).

3.1.3 Relaxation of the Assumption that Platform Overheads are Negligible

The assumption that platform overheads are negligible is fairly simplistic. To assume that their is no cost associated with task scheduling and preemption by the RTOS can lead to neglecting what may be a large taxation of the processor. Katcher *et al.* [18] relaxed this assumption by introducing the scheduling costs of a system. The work performed was a first step in bridging the gap between real-time scheduling theory and implementation realities.

Four different scheduling implementations are presented and analyzed by Katcher *et al.* to determine the overhead and blocking costs for each implementation. The metric used for evaluating the scheduling algorithms is known as the breakdown utilization, and is defined as follows. The execution time (c_i) in a given task set is multiplied by a scaling factor α while keeping the period (p_i) fixed. The task set is scaled to the point at which any increase in α will cause at least one task to miss its deadline. The utilization of the task set given by Equation 3.3 is known as the breakdown utilization. Analytical measurements using the rate monotonic priority

assignment show that the schedulable utilization of the scheduling algorithms when compared to the ideal (i.e. when no overheads are considered) degrades as the number of tasks is increased. Even with a small number of tasks (5 or 10), the breakdown utilization is reduced by 20% from the ideal. Selection of the scheduling algorithm is also found to play a role in determining the amount of overhead introduced.

$$\text{Breakdown Utilization} = \sum_i \frac{\alpha c_i}{p_i} \tag{3.3}$$

All four of the scheduling implementations under analysis by Katcher *et al.* assume a perfectly preemptable kernel. The scheduling implementations are divided into two categories: event-driven and timer driven. Event-driven implementations require a mechanism to generate interrupts that signal task periods (frequently referred to as task arrivals). This is further subdivided into two implementations: integrated interrupt event-driven scheduling and nonintegrated interrupt event driven scheduling. In the integrated system, the priority of the hardware interrupt matches the priority of the software tasks, while in the nonintegrated system, the hardware and software priorities do not match. Thus the nonintegrated system results in an additional overhead to schedule the task.

The timer-driven schedulers use periodic timer interrupts from a programmable timer to allow the scheduler to run. An internally maintained time value is used to evaluate task periods and decide when to invoke them. The two timer-driven scheduling implementations are: timer-driven scheduling and timer-driven scheduling with counter. The timer-driven scheduler with counter uses is an optimization of the timer-driven scheduler. Instead of invoking the scheduler at every timer interrupt,

a simple counter is used to keep track of the number of timer ticks before the next task's arrival. Each timer interrupt decrements the counter, and the scheduler is only invoked when the counter expires.

The overheads and blocking terms of each scheduling algorithm are broken down into basic units. Analysis of a scheduling implementation determines where each overhead unit occurs. The summation of basic units is then used to calculate the overhead when a task is released, or complete, or due to a periodic timer used in scheduling. To highlight this, an example implementation of the nonintegrated interrupt event-driven scheduling theory is provided below.

In the nonintegrated interrupt event-driven scheduling mechanism, all tasks are initiated by external interrupts. However, the priority of the software task associated with the interrupt is not immediately known to the RTOS. Thus, every time a task arrives, the current task is interrupted to determine if the new task is of higher or lower priority. If the arriving task has higher priority, it will preempt the current task. If it is of lower or equal priority, it will not preempt the current task, but will still interrupt it to perform scheduling.

Figure 3.1 shows an example of nonintegrated interrupt driven scheduling. The processor is initially idle until the first interrupt occurs. This interrupt signals the arrival of $task_2$. Since no other tasks are running on the system, $task_2$ is allocated the processor. The overhead involved in handling the interrupt and scheduling $task_2$ is represented by $C_{preempt}$. While $task_2$ is executing, an interrupt is generated by $task_1$ signalling that it is ready to run. As $task_1$ is higher priority than $task_2$, the processor is now allocated to $task_1$. Once again, a $C_{preempt}$ overhead is incurred. During the execution of $task_1$, $task_3$ becomes ready to run and generates an interrupt. Since $task_3$ is lower priority than the current running task, $task_1$, the processor is returned

to $task_1$. However, the interrupt causes $task_1$ to be blocked while the priority of $task_3$ is determined and the scheduling of $task_3$ takes place (denoted by $C_{nonpreempt}$). Once $task_1$ completes, a trap is issued to end the task and start the next highest priority task that is ready to run. The overhead of exiting a task is denoted as C_{exit}. $Task_2$ continues to run until it is interrupted once again by $task_1$. Once the second instance of $task_1$ is completed, $task_2$ continues running to completion. The processor is finally allocated to $task_3$. A final instance of $task_1$ interrupts $task_3$ before all of the tasks complete their execution.

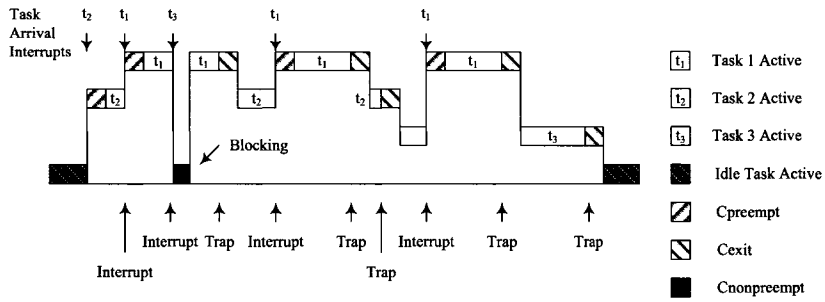

Figure 3.1: Nonintegrated interrupt driven scheduling [18].

The overheads shown in Figure 3.1 are broken down into basic units as follows:

$$\begin{aligned} C_{preempt} &= C_{int} + C_{sched} + C_{store} + C_{load} \\ C_{nonpreempt} &= C_{int} + C_{sched} + C_{resume} \\ C_{exit} &= C_{trap} + C_{load} \end{aligned} \tag{3.4}$$

where,

- $C_{preempt}$: total overhead to process an interrupt when a new task preempts the currently running task

- $C_{nonpreempt}$: total overhead to process an interrupt when preemption of a task does not take place
- C_{exit}: total overhead to clean up after a task has completed
- C_{int}: time to handle the interrupt
- C_{sched}: time to execute the scheduling code
- C_{resume}: time to return to the previously active task
- C_{store}: time to save the state of the active task
- C_{load}: time to load the new active task
- C_{trap}: time to handle the trap generated by normal completion of a task

Each of the values on the right hand side of Equation 3.4 are measured values, and used to calculate the overheads introduced by the scheduler. Using the calculated overhead values, a set of periodic tasks are considered to be schedulable under the nonintegrated interrupt event-driven implementation using the rate-monotonic scheduling algorithm if the following holds:

$$\forall i, 1 \leq i \leq n, \min_{0<t\leq D_i} (\sum_{j=1}^{i} \frac{C_j + C_{preempt} + C_{exit}}{t} \left\lceil \frac{t}{p_j} \right\rceil) + \frac{(n-i)C_{nonpreempt}}{t} \leq 1 \quad (3.5)$$

In applying Equation 3.5, each task from 1 to n is considered individually. The first part of the equation:

$$\min_{0<t\leq D_i}(\sum_{j=1}^{i}\frac{C_j + C_{preempt} + C_{exit}}{t}\left\lceil\frac{t}{p_j}\right\rceil) \tag{3.6}$$

adds up the processor utilization of $task_i$ and all higher priority tasks that have been released by time t. Finding the minimum value for the summation gives us the earliest point at which the demanded time of the processor satisfies the time demanded by the tasks, if it exists, in the range $0 < t \leq D_i$.

As lower priority tasks may also interrupt $task_i$, the term:

$$(n-i)C_{nonpreempt}$$

$$\frac{(n-i)C_{nonpreempt}}{t} \tag{3.7}$$

considers the worst case blocking by the remaining $(n-i)$ tasks. By using RMA priority assignment, the periods of $task_{i+1}$ to $task_n$ are greater than $task_i$. Thus, they can only block $task_i$ once during the period p_i. Equation 3.7 considers the worst case where all of the $(n-i)$ remaining tasks block $task_i$.

The measurement of the overheads as broken down by Katcher *et al.* requires an in depth knowledge of the underlying workings of the scheduler. As most COTS development tools are proprietary, this information may not always be available. In the next section, research on estimating overheads without in-depth knowledge of the scheduler is introduced.

3.2 Overhead Estimation

When analyzing with COTS tools, it is not always possible to break down each of the overheads into basic units. As the implementation details of the application are hidden, determining the actual costs may not be possible. In this situation, manual measurements can be made to estimate the overheads.

Stewart provides an overhead estimation technique for measuring the cost of performing a context switch from one task to another [19]. Figure 3.2 shows a set of three tasks and the overheads involved in performing a context switch (denoted as Δ_{thr}). Measurement of this overhead can be performed by creating a simple application with periodic tasks where each task toggles a bit on an output port. Using a logic analyzer, the timing values of the bit transitions can be captured. Figure 3.3 shows an example of a trace captured by a logic analyzer with two tasks. The tasks are designed so that the first bit transition is from low to high and the final bit transition is from high to low. Channel 1 captures the output of the lower priority task running on the system. When a higher priority task arrives (output on channel 2), the lower priority task is interrupted and must wait until the higher priority task completes before continuing with its execution. Using the two tasks in Figure 3.3, the RTOS overhead can be approximated as:

$$2\Delta_{thr} = t_1 - t_2 - t_3, \tag{3.8}$$

where,

- Δ_{thr} is the context switch overhead of the task,

- t_1 is the total time the lower priority task is interrupted,
- t_2 is the execution time of the higher priority task, and
- t_3 is the duration of time required to transition a single pulse on the output port.

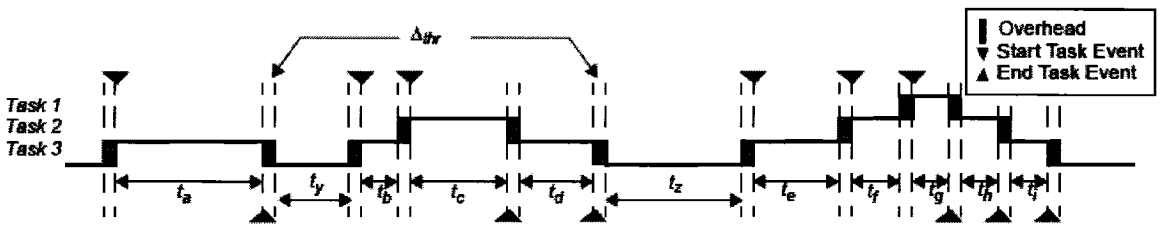

Figure 3.2: Effects of context switch overheads [19].

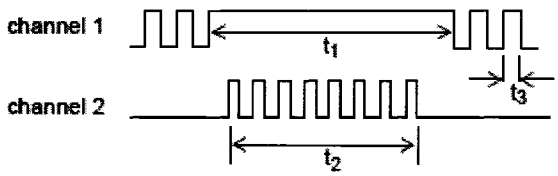

Figure 3.3: Measuring context switch overhead [19].

The reason for subtracting t_3 from $t_1 - t_2$ is that during the measured duration of t_1, the code for producing one pulse (t_3) is also executed. If the context switch occurs during a low cycle, t_3 should be measured as the length of a 0 pulse rather than a 1 pulse. The calculation gives us $2\Delta_{thr}$ since two context switches take place; the first switch occurs when the higher priority task switches in to use the processor, and a second occurs when the higher priority task is completed and execution is switched back to the lower priority task. Using these measured values, the analysis of real-time systems can now include the costs of the system.

3.3 Applying Theory in Practice

The concept of applying real-time theory to physical systems is not a new one. In the realm of hard real-time systems, a measure of confidence that the system will perform within its timing constraints is of great importance. Simulation and testing are useful tools for ensuring the correct behaviour of an application; however they do not always deal with time adequately. Applications of scheduling theory are being discussed in a variety of fields including control systems, multimedia and aerospace.

In the following section, research performed to extend RMA theory in order to apply it to practical systems is outlined. Section 3.3.2 provides examples of research in which RMA is extended in order to account for the execution costs of practical systems. Finally, examples of tools created to assist in the analysis of time in real-time systems are provided in Section 3.3.3.

3.3.1 Extending Theory for Practical Systems

The application of RMA theory for analysis of practical systems was initially slow to take hold. A research initiative to apply Liu and Layland's theory began in 1982 with the collaboration between Carnegie Mellon University (CMU) and IBM. The work was further extended when the Software Engineering Institute (SEI) joined forces with CMU in 1988 in a US Department of Defence initiative to further extend the methods for a broader range of systems [20]. A direct result of the work was the creation of a handbook by Klein *et al.* [21] providing guidelines to apply RMA theory to practical applications. The handbook provides the developer with a collection of methods (based on RMA) that can be applied in different real-time situations. An application of the theory in the area of industrial computing was highlighted by Klein

in [22].

3.3.2 Practical Implementations of Theoretical Research

The work performed by Katcher *et al.* described in Section 3.1.3 comes from research conducted at CMU. The research directly led to the further extensions by Kettler *et al.* [23], where the scheduling implementations of Katcher *et al.* are applied to digital signal processing (DSP) RTOSs. This involves the creation of individual models to represent three different DSP RTOSs used in multimedia applications while including the implementation costs of each scheduler in the model. Each RTOS under consideration implements a different scheduling algorithm, similar to the ones described in Section 2.2.2. The models proved to be effective when comparing and evaluating the performance of different RTOSs, even when different scheduling algorithms were used. The results further showed how the models could be used to accurately tune or optimize the real-time performance of a given architecture.

The extensions of RMA theory produced by the collaboration between CMU with SEI allowed for theory to be applied to real-time data acquisition applications where direct memory access (DMA) is performed by tasks. Analysis of an application using DMA is performed by del Val and Viña in [24]. The application consists of a real-time system used to inspect industrial containers to find early defects in their welds. In this application, RMA theory is extended to allow for blocking and asynchronous I/O in the system. del Val and Viñ found the benefits of using RMA to be better predictability, better understanding of operating system overheads, and allowing for full understanding of system overheads. This is largely due to the fact that performing the analysis forces developers to mathematically characterize the application. The drawbacks come from having to estimate the blocking costs and DMA overheads, and

the fact that estimates on execution times can only be made in later stages of the development. The pessimistic nature of the scheduling analysis is also seen to be a drawback in certain applications.

Burns and Wellings consider the implications of using scheduling theory on an attitude and orbital control system for the Olympus satellite [25]. In this case, analysis using scheduling theory is applied to an application developed in Ada. The deadline monotonic scheduling algorithm is used in this application. This is a static priority assignment algorithm which assigns highest priority to the task with the shortest deadline and decreasing priorities to tasks as their deadlines increase. Practical issues of scheduling theory are addressed by extending the model to include scheduler costs by using the work of Katcher *et al.*, as well as including measurements of context switch overheads, and blocking costs due to resource sharing. When applying the model, the task under consideration and all higher priority tasks commence at the same time (i.e. a critical instant). However, the worst case behaviour is actually seen when the tasks are started offset to each other. This is due to the nature of the context switch overhead. When starting all of the tasks at the same time, the minimum number of context switches actually takes place, thus a lower cost is incurred. An analysis of a set of tasks that did not share a critical instant (i.e. start offset to each other in order to give a worst case execution time) was not performed due to the complexity required in the analysis. However, based on comparisons with a previous version of the application that uses a cyclic executive, the new model based solution was found to be more effectively engineered when modifications to the application were required.

Rate monotonic theory is applied to the scheduling of a set of real-time tasks for control of an autonomous robot by George and Kanayama [26]. The use of RMA

theory allows for separation of the analysis of logical correctness of the application from the analysis of timing correctness. The change is made from a cyclic executive by creating a scheduler on top of the operating system in user space. Applying rate monotonic theory is shown to simplify modification of the tasks, allowing for addition, deletion, or modification of a task without disturbing the timing correctness.

3.3.3 Generic Analysis Tools

Stewart and Arora developed a prototype tool called AFTER (Assist in Fine Tuning Embedded Real-time systems) to assist in analyzing and fine-tuning timing properties [27]. This generic tool is seen as a first step towards meeting the goal of fully automated analysis. The application is first implemented in the developers' language of choice. As long as the software is analyzable (defined starting and stopping points, periods with defined timing specifications, and defined deadlines), developers can make use of the AFTER tool for analysis and interactive prediction of real-time scheduling. The real-time behaviour of the system is captured by using a profiling mechanism as the application executes. This can be done by making use of a logic analyzer if no profiling tools are available. The data is then uploaded to the system running the AFTER tool, which correlates the measured data with the specifications of the system and displays the results to the user. If problems are detected, a prediction mode of the tool allows the developer to compare the results of switching between different scheduling algorithms, or optimizing one or more tasks. The prediction generated by AFTER is determined by using the measurements of the context switch overheads that occur at the beginning and end of each task, the measurements of each of the tasks' execution times, and the proposed modification made by the user. The effects of the proposed changes are immediately seen, thus, only changes that predict an

improvement should be made. Stewart and Arora broke up the design of the AFTER tool into functional units. The functions of each unit, including issues and solutions determined during development of the tool, are provided as a guideline for others creating an automated analysis tool.

A higher level approach to achieving timing confidence is introduced by Huang *et al.* in [28]. Current modeling languages have difficulty dealing with time, and timing is often platform specific. Principles for coping with these difficulties that should be incorporated by an "adequate" design approach are proposed. This includes an expressive modeling language as well as automatic and correctness-preserving transformation tool to implement the final system. The modeling language should include: adequate expressive power to deal with real-time issues such as timeliness, determinism, and concurrency; platform independent semantics to make simulation and verification unambiguous; operational semantics which lends itself naturally to executability; and modularity support in order to manage the complexity of a system. A platform independent approach called POOSL (Parallel Object-Oriented Specification Language) is introduced to assist in modeling time and to address all of the issues raised. Initial experiments with the use of POOSL confirmed the advantages of their approach over other tools on the market (which did not contain all of the design approaches described). As real-time development turns to the use of higher level development languages, the need for such tools becomes more evident.

3.4 Analysis of the LabVIEW Development Environment

Most of the available work on LabVIEW has been more application oriented and has been developed either at NI or by LabVIEW developers. The implementation of multirate applications using the features of the LabVIEW Real-Time Module is outlined in a white paper by NI [12]. While the paper does an effective job describing how to implement a multirate application, no insight on real-time analysis is provided. Mention of setting the task priorities based on increasing periods is made; however, no specific reference to RMA or any theoretical work on scheduling is given.

Work on timing analysis of VIs has been performed by Winiecki and Bilski [29, 30]. They present a methodology to measure the time required to execute an operation in LabVIEW. First, a basic operation is repeated multiple times, and a measurement of the time required to perform the repeated operation is obtained. Then the execution time of the operation is determined by dividing the measured time by the number of repetitions. Finally, the measurement and calculation to determine the execution time is repeated a pre-determined number of times, and the final result is obtained by taking an average of all of the acquired times. The research made use of LabVIEW version 6.1 without the use of the Real-Time Module, and running on Windows 9x or NT.

The work is further extended in [31] to determine the real-time mode (hard or soft real-time) that can be achieved on the system. Instrumentation applications that perform data acquisition and software analysis of the data (such as a spectrum analyzer) before the next data arrives are the target applications for this research. In this situation, the DAQ card obtains the next set of data while the software is

processing the previous set. Measurements of the execution times of the software processing gives values for the disturbances (i.e. longer execution times) and their frequencies of occurrence. The experiments are conducted on desktop computers running LabVIEW version 6.1 on Windows 2000. Using the measured disturbances, the most frequent instances can be eliminated by adjusting the margin for disturbances on the DAQ card. This is essentially done by increasing the period at which the DAQ card acquires the next set of data so that the data will arrive after the software has completed processing the previous data. Bilski and Winiecki found that this can result in achieving soft real-time deadlines. In order to achieve hard real-time deadlines, alternatives to running the standard LabVIEW package on a desktop are recommended. This includes programming multithreaded applications in LabVIEW or C, or by creating low level programs on a real-time kernel.

Chapter 4

Proposed Research

The research presented in the previous two chapters provides the required background for the analysis of real-time systems. In this chapter, an analysis of the state of the art research introduced in Chapter 3 is conducted in Section 4.1 to show areas that can be further explored. An introduction to the problem being addressed is provided in Section 4.2. Finally, the thesis of this research is stated in Section 4.3, and the scope and contributions of the work are outlined in Section 4.5 and Section 4.6 respectively.

4.1 Current Research Limitations

This section provides an analysis of the current research being conducted on real-time systems analysis introduced in Chapter 3. The limitations of the current research are identified when attempting to analyze practical applications.

4.1.1 RMA Scheduling

RMA theory was first introduced by Liu and Layland and includes a relation (Equation 3.1) to determine if a set of tasks can be scheduled under worst case loading. The theory is further extended by time demand analysis to deal with situations where the relation fails but the set of tasks may still be schedulable. While this work does provide a good basis for analysis, a well defined method to extend the theory to practical applications is not given. The assumptions made are also very simplistic, and may not be achievable in practical applications.

Further work by Katcher *et al.* on nonintegrated interrupt event-driven scheduling provides a solution to deal with the costs of a scheduler and the context switch between tasks when pre-emption occurs. The generic models can be applied to various different RTOS implementations of schedulers. However, having to break down the costs of the overheads into basic units is impractical, and may be impossible if detailed knowledge on the inner workings of the RTOS is not known.

4.1.2 Overhead Esitimation

As more developers turn towards the use of COTS real-time development environments, the details of the implementation of the final application are no longer available for analysis. This makes it more difficult to apply theoretical algorithms to analyze these environments. The overhead estimation technique introduced by Stewart outlines a method to measure the operating system costs without knowledge of the inner workings of a system. However, the technique assumes that overheads only occur at the beginning and at the end of a task's execution. While this assumption may be practical in certain situations, different sets of tools often employ different methods

for context switching and scheduling. The measurements of overheads at the beginning and at the end of a task may not be adequate to include all of the costs of the system.

4.1.3 Practical Implementations

A great deal of work to bridge the gap between theory and practice came from the collaboration between CMU and SEI. The goal was to extend the research introduced by Liu and Layland by creating models to relax some of the underlying assumptions made by the original theory. A direct result of this was the creation of a handbook to assist the developer in applying RMA analysis to practical systems. However, the research was further reaching as it opened up the discussion on how to apply research theory to practical applications.

The research stemming from the collaboration between CMU and SEI has allowed for the analysis of applications in a number of industries. This includes the real-time DAQ application analyzed by del Val and Viña. Kettler *et al.* applied the work of Katcher *et al.* to compare the implementation costs of several different multimedia RTOSs. Burns and Wellings use the relaxation of assumptions to analyze an application used for a satellite system. The benefits of applying RMA are shown by George and Kanayama in their work with autonomous robots. While the need to apply theory to real systems is demonstrated in each of these areas, the research endeavours have been restricted to the specific application being analyzed. A generic model for analysis of any application created on similar systems has yet to emerge.

4.1.4 Generic Analysis Tools

A more generic tool to automate the analysis of a real-time system is created by Stewart and Arora in developing the AFTER tool. As in the overhead estimation technique, this tool assumes that overheads costs only occur at the beginning and the end of each task. While a guideline for creating an automated analysis tool is provided, other system overheads are not included in the analysis. Thus only a small subset of systems can be analyzed using the AFTER tool.

A higher level approach to analysis by the introduction of time in modeling languages is performed by Huang *et al.* This involves a platform independent approach to assist in modeling time. The advantage of using their approach is shown to be a more accurate representation of time with fewer introduced errors. However, no analysis is provided to ensure timing requirements can be met. Additionally, overheads introduced by the system are not discussed.

4.1.5 LabVIEW Development Environment

Real-time analysis on the LabVIEW tool from NI has yet to be performed. Winiecki and Bilski provide insight on determining execution times by using an averaging method on non-real-time platforms. This work was extended when the results were applied to instrumentation applications to assist in determining if real-time deadlines can be met. However, their work was performed on the LabVIEW development environment running on a desktop system. Further research is required to include the analysis of the Real-Time Module running on a real-time target platform.

4.2 Furthering Research - The COTS-RTP Project

Despite the decades of research, the final step in the creation of a tool to assist in applying theory to practical applications has yet to take place. Currently, no fully functional COTS tool for the analysis of real-time embedded systems has come forward. A tool that can provide a priori analysis of a set of tasks to determine if they are schedulable would be of great benefit. Such a tool would be able to include the characteristics of the development environment and RTOS so that measurements of the overheads can be included in the analysis. Determination of the worst case execution times of tasks coupled with the periods of the tasks can then be used to predict if a schedulable solution is possible. This ideal solution is part of the long term goals of the COTS Real-Time Profiling research project.

4.3 The Thesis

The focus of this research is to show that it is possible to model industry-calibre real-time development tools using research theory in order to obtain a more accurate a priori prediction that hard real-time deadlines can be met when compared to RMA alone. This involves creating a model, based on current theoretical research in real-time systems, to allow a priori analysis of a set of tasks to determine if they are schedulable. Testing of the model is conducted to determine if it can accurately predict if a set of tasks are schedulable.

4.4 Scope of The Research

The proposed solution consists of analysis of the LabVIEW (version 7.1) development environment extended by the LabVIEW Real-Time Module, and executing on a supported real-time target platform. Overheads introduced due to scheduling, context switching among tasks, and by LabVIEW tasks created by the development environment are measured and included in the model. User-created tasks are restricted to those created by the Timed Loop feature introduced by the Real-Time Module, which provides support for the periodic release of tasks at deterministic rates. To ensure determinism, no user interaction will be considered.

The resulting LabVIEW Real-Time Cost Model (LRCM) is generic such that applications created in LabVIEW using the Timed Loops can be analyzed to determine if all tasks' deadlines can be met. Testing to ensure correctness of the model uses both simulated data and a practical industrial application (i.e. the CTS system).

The LRCM is the first step in bridging the gap between theory and real-time development using COTS tools. As a starting point, the LRCM includes measurements based on average case execution times. The applications being tested are also restricted to user tasks with no inter-task communication. Successful creation of the LRCM assists in setting the direction for further study and analysis of COTS development environments.

4.5 Research Contributions

Several contributions have been made in this research to advance the theory itself and to allow for the analysis of practical applications. The contributions of this research are:

- Providing guidelines for determining what information needs to be obtained from a development environment before a model can be created (Section 5.1).
- An in-depth analysis of the overheads of LabVIEW with the Real-Time Module as running on a specific target (Section 5.5).
- Creating the LRCM that can be used by LabVIEW developers to fit an application to a theoretical algorithm and gain confidence that the application will meet its deadlines (Section 5.4.7).
- Showing that a model can be created without knowing the inner workings of a development environment.
- Showing that high-level development environments can be modeled using current theoretical methods in order to assist in determining if a set of tasks are schedulable.
- Applying theoretical research on time demand analysis, nonintegrated interrupt event-driven scheduling, and overhead estimation techniques to produce a practical solution to scheduling analysis.

4.6 Document Outline

The remainder of this document is broken down as follows. The LRCM is presented in Chapter 5 and begins with a guideline for the characteristics that need to present before creating a model of a real-time development environment based on RMA theory. This is followed by a description of the target system being modeled as part of this research. Next, a list of assumptions under which the LRCM operates is provided.

The characteristics of LabVIEW with the Real-Time Module are determined and theoretical research is then applied to these characteristics resulting in the LRCM. Finally, measurement methods used to determine the overheads, as well as the final measured values are presented.

Chapter 6 provides a thorough discussion on the testing done to validate the model. The test setup as well as each test case is described, followed by a discussion of the test results. The Chapter concludes with an analysis of the LRCM.

Chapter 5

Profiling the COTS Development Environment

In this chapter, the methodology used to profile the LabVIEW development environment with Real-Time Module is described. It begins with an outline of the characteristics that must be present in order to align the model with RMA theory. The system being analyzed is described in Section 5.2, followed by a summary of the assumptions made in the LabVIEW Real-Time Cost Model in Section 5.3. Section 5.4 presents the LRCM with corresponding description of the terms included. Finally a break-down of the measurement techniques used to determine the overheads, as well as the values of these overheads is included in Section 5.5.

5.1 Model Guidelines

To ensure a correct profile of a COTS development environment a basic understanding of the system must be sought. Each vendor typically has a slightly different method of

implementing tasks to ensure real-time performance. Once the methodology used in the tool has been established, a model based on theoretical research can be developed to assist in predicting if a set of tasks will be schedulable. This section outlines each of the characteristics that need to be present before a model, based on RMA theory, can be created.

5.1.1 Definition of a Task

When applying RMA theory to an application, tasks are typically implemented as threads. Management of the threads is done by the real-time operating system. The context and parameters of each thread are stored to allow the operating system to release them periodically and schedule them based on priority.

Scheduling analysis requires that each task be considered individually. Before applying a model, the definition of a thread must be established. Each development environment may have its own method used to create threads; in some instances, calling a subroutine can create another thread. As each thread must be scheduled to run on the system, all the threads that are created must be taken into account. Once all methods of creating a thread has been established, the model can created to properly handle the release of each task.

5.1.2 Assignment of Priority

Once the definition of a task has been established, the next step is to determine how priorities are assigned. The following questions must be answered in order to determine a correct model for the COTS tool.

1. How are priorities assigned?

2. Are the priorities static or can they be changed dynamically?

3. Can more than one task have the same priority?

4. How do the development environment priority levels relate to RTOS priority levels?

The first question relates to how the development environment deals with priorities. Some use a numbering scheme, while others may be based on names. It is important to determine the hierarchy of the priorities, i.e. ordering from highest priority to lowest priority level because the ordering needs to be known for RMA analysis.

Question two assists in determining what basic model to start with to ensure proper alignment with research theory. If priorities can be changed dynamically, a fixed priority algorithm may not be the best one to use as the basis for a model. Also, it is important to determine if the priority is set by the developer, or if the development environment assigns a priority to each task based on some heuristic. For RMA, a fixed priority method, where the priority can be set based on an increasing period of a task, needs to be present.

The third question becomes important when scheduling the tasks. In RMA theory each task needs to have a distinct priority. If this is not possible, a different approach will need to be considered.

Finally, if the development environment is not closely linked with the RTOS, different methods of assigning priority may be possible by both the COTS tool and the RTOS. This is a potential for conflict, especially if the COTS tool has more priority levels than the ones available in the RTOS. Common methods of dealing with this include a mapping of the priority levels onto the RTOS priority levels, or

by using a separate scheduler which is independent of the RTOS. Priority hierarchy needs to be established before RMA analysis can be conducted.

5.1.3 Task Deadline

For task deadlines, one of three possibilities is available: the deadline is less than the period of the task; the deadline is equal to the task period; or the deadline is greater than the period of the task. Depending on the criteria that can be set in the development environment, one or more of the options may be available for use. One of the basic assumptions of RMA analysis is that the deadline equals the period of the task, thus the second option must be available for RMA analysis to be performed. Extensions have been proposed to relax this assumption and may be used to model each case; however, this is beyond the scope of the research.

5.1.4 Scheduling of Tasks

The scheduling of tasks can only be established after determining the method used to set and adjust priorities. If task priorities can only be set statically at design time, then scheduling can only be done using a fixed priority scheduler such as RMA. However, if the priorities can be changed dynamically during the execution of the applications, a more complex scheduling algorithm may be required.

Another critical factor to consider is if tasks can be pre-empted. If no pre-emption is allowed, the task being run at any moment may not be the highest priority ready to run task. The scheduling method used to handle tasks of equal priorities (if allowed) must also be determined. When creating a model based on RMA, task pre-emption must be present and each task must have a distinct priority.

5.1.5 Kernel Overheads

The relaxation of the assumption that platform overheads are negligible requires that the overheads introduced by the kernel be established and measured. In this research the two kernel overheads that are being considered are the context switch overhead, and the scheduling overheads.

While the context switch overhead only occurs when the execution of a task begins and ends, establishing the scheduler overhead can be more difficult. If the periods of the tasks cannot be changed dynamically, the scheduler overhead may be a start-up cost where the schedule is determined at the beginning of the program execution. If this is not the case, a more thorough analysis must be done. It is fairly common to find that the scheduler is invoked either when a task is released or when it has completed execution. However, it is important to keep in mind that if two or more tasks are released at the same time, the scheduler might only be run once to schedule all of the tasks released. The priority of the scheduler must also be established. For instance, if the scheduler can pre-empt a higher priority task when a lower priority task is released, there is an additional blocking overhead for higher priority tasks that must be taken into account.

5.1.6 Overheads Introduced by the Development Environment

When using a COTS tool, there may be additional tasks created to service some of the utilities used by the development tool. This could include timer routines to ensure all threads are running on time, or hardware service routines to safely handle any hardware interrupts. Additionally, there may be overheads introduced to do all

the necessary background processing to implement the threading model.

If the overheads introduced are only at the beginning of the execution of an application, they can be considered as a start-up cost and kept out of the model. However, if one or more periodic tasks are created to handle the overheads, they must be included in the analysis to ensure that proper time is allocated to service the tasks.

5.2 System Description

The COTS development environment being considered in this research consists of the LabVIEW version 7.1 environment with the additional LabVIEW Real-Time Module available from NI. With the addition of the Real-Time Module, applications created in LabVIEW can be downloaded onto a real-time target for execution. The real-time target is a headless (i.e. no user interface) system that runs the Phar Lap RTOS which is developed by Ardence.

The target hardware platform being used consists of the PXI-8186 system available from NI. It is a 2.2GHz Pentium 4 Mobile processor with 512 KB of on board cache and 1 GB of DDR RAM. The PXI-8186 is housed in a PXI (PCI eXtensions for Instrumentation) chassis in which additional hardware I/O modules can be added. Figure 5.1 shows an example PXI chassis with the CPU and hardware modules. The CPU is on the left side of PXI chassis, and hardware modules are added to the slots to the right of the CPU. For the purposes of this research, no additional hardware modules were used. The hardware system being tested is a part of the platform being used to control the CTS system at the NRC-IAR.

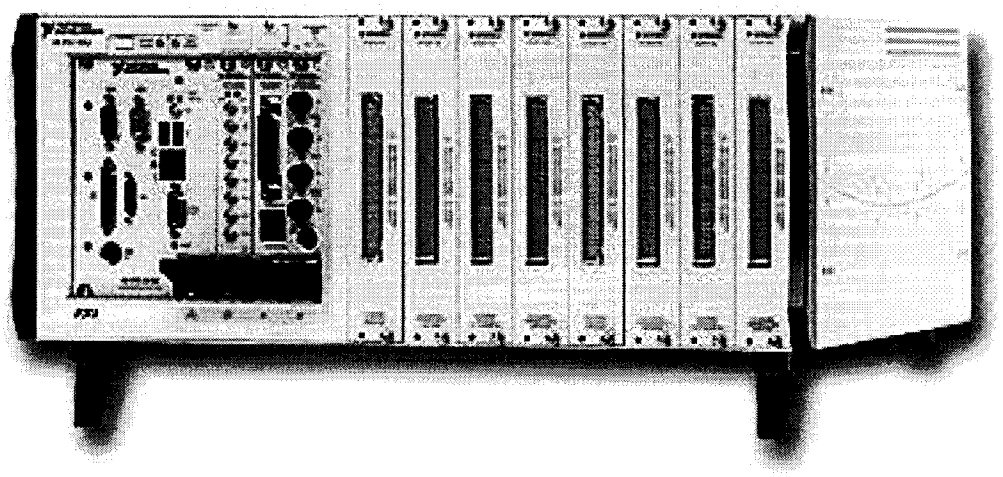

Figure 5.1: Target platform hardware.

5.3 Model Assumptions

The goal of this phase of the COTS-RTP project is to analyze LabVIEW with the Real-Time Module. RMA theory is being used to model the characteristics of the development environment with the exception of the assumption that platform overheads are negligible. The kernel overheads being considered in this research include the context switch and scheduler overheads. In addition to the costs introduced by the kernel, any additional overheads introduced by LabVIEW will also be included in the LRCM.

The following is a list of assumptions made by the LRCM:

1. Each task is periodic; the deadline is the same as the period.
2. Each task is independent, i.e. no inter-task communication.
3. Processing requirements of each task are constant.
4. Each task has a distinct fixed priority assigned as per RMA policy

5. Interactions between LabVIEW and the RTOS are not considered.

6. Effects of caching are minimal and not considered.

7. Only one timing source is used for all Timed Loops.

In order to achieve assumption three, no hardware interaction was performed using DAQ modules. Additionally, user interaction was not permitted as it does not allow for deterministic execution of the tasks. The execution time of each task is based on an average measurement and is not necessarily the worst-case execution time. Use of the average execution time is due to the high level abstraction of the graphical development environment. This results in the lack of knowledge as to how the final implementation is run on the hardware platform, making worst-case execution time calculations very difficult.

The development environment and target hardware platform are treated as a black box; thus, knowledge of the interactions between the RTOS and development environment are unknown. System overheads being considered include the context switch time, the cost of the scheduler, and the introduction of background tasks by the development environment. Each of the overheads are determined using information gained by testing of the development environment or from documentation provided by NI.

5.4 The Model

In Section 5.1, a guideline for characteristics that need to be present when developing a model based on RMA is provided. In this section, the guidelines are applied to the LabVIEW development environment with the Real-Time Module. This is followed

by the presentation of the LRCM which was created to characterize the costs of the systems and allow the developer to analyze a set of tasks and determine if they are schedulable.

5.4.1 A LabVIEW Task

In LabVIEW, tasks are implemented as threads. As LabVIEW was originally designed for non hard real-time applications and later expanded to create real-time applications with the addition of the Real-Time Module, two distinct methods of creating threads are available. The first method to create a thread is by creating a VI. The second method to create a thread is by creating a Timed Loop.

The LRCM only focuses on tasks created by the Timed Loop feature as the VI's do not provide the features to create periodic tasks that can be run deterministically. Thus, further analysis of VI threads will not be considered beyond this point.

5.4.2 Priority Assignment Structure in LabVIEW

Timed Loops use an unsigned integer value for priority. However, as LabVIEW only allows 128 Timed Loops to be created in an application, only 128 distinct priority levels can be assigned. Priority levels can be shared among Timed Loops, but for the purposes of this research, it is assumed that distinct priorities are assigned to each task.

The priority level of a Timed Loop can be set during development or changed at run-time from within the loop itself. Any change made at run-time is not fully dynamic as it will not take effect immediately, but at the next iteration of the loop. Since this change is not immediate, a dynamic scheduling algorithm cannot be im-

plemented. Due to this restriction, RMA, which is a static scheduling algorithm, is used as a basis for the model. Thus, it is assumed that the priority of each task is fixed for the application.

The number of priority levels on the Phar Lap RTOS is less than 128. However, scheduling is done by a LabVIEW implemented scheduler above the RTOS layer. Thus the RTOS is unaware of the LabVIEW priorities. The interaction between LabVIEW threads and Phar Lap threads are not known.

5.4.3 Task Deadlines in LabVIEW

The Timed Loop feature in the Real-Time Module has built in structures to deal with periodic tasks. In version 7.1 of LabVIEW, the deadline of a task created using a Timed Loop is the same as the start time of the next period. If a task finishes late, the "Finished Late" flag within the Timed Loop is set and can be read in the next iteration of the loop.

The handling of late tasks is done based on the mode options set for the Timed Loop (as described in Section 2.3.2). In order to gain feedback on when a task fails, the option to continue executing a task, even if it was late, was used. This ensured the "Finished Late" flag was set and could be used to determine failure of the application.

5.4.4 The LabVIEW Scheduling Methodology

Scheduling in LabVIEW is done in a priority driven pre-emptive manner. Thus the highest priority thread that is ready to run will always have control of the CPU. If two threads share the same priority, scheduling of the two tasks will be done via time-slicing in a round robin fashion. In order to simplify the model and align it

with RMA theory, it is assumed that the developer will assign a distinct priority to each task based on RMA priority assignment, and the priority of each task will be fixed during the execution of the application. This priority assignment is easily implemented using the LabVIEW Timed Loops.

5.4.5 Kernel Overheads

The two kernel overheads being considered are the context switch and the scheduler. As the context switch overhead cannot be explicitly seen, estimates are made using the techniques outlined by Stewart (recall Section 3.2). The context switch overhead is incurred at both the beginning and at the end of a task.

The Execution Trace Toolkit is used to determine when the LabVIEW scheduler executes. The scheduler task is seen to run at the time the task is released. If a lower priority task is released while a higher priority task is currently running, the scheduler will interrupt the higher priority task. Once scheduling has been completed, the higher priority task will continue to execute. If multiple tasks are released at the same time, the scheduler will run only once before the tasks begin to execute. The scheduling overhead is assumed to be constant.

5.4.6 LabVIEW Development Environment Overheads

When running an application created in LabVIEW, additional threads that are not part of the application can be seen to be running on the system. Further analysis and discussion with support personnel at NI resulted in identifying three threads that were created by LabVIEW for each application. The first task, called the ETS Timer Thread, is used to maintain the system timer used by LabVIEW. The second and

third tasks are used to poll the Ethernet link between the real-time module and a host system that handles all interaction with the user. Although the tests were designed to meet the assumption that no user interaction takes place, monitoring of the Ethernet is still present and periodically polls the connection to check for incoming messages. These tasks are included in the model as described below.

Each of the LabVIEW-created tasks runs at a priority below any of the application threads. Missing the deadlines of these tasks is permitted occasionally, but the effect of repeatedly missing them is not known. When applying the LRCM, the priorities of these tasks are modeled according to their periods of execution. If application tasks have longer periods than the LabVIEW tasks, the LabVIEW tasks will be assigned a higher priority than the application tasks. This ensures that the processor time required for the LabVIEW tasks is considered, avoiding any long term impact caused by missing their deadlines

Additional overheads are also introduced by the Timed Loops. As the Timed Loop has features to internally handle the periodic execution of the loop and maintain variables to monitor the start/end time and finished late flags, additional costs are incurred in the first iteration of the loop to set up the infrastructure needed to carry out these duties. The creation of a timing source for the Timed Loops also invokes another scheduler thread to deal with setting up the timing source. As these costs only occur at the start of an application, they are not included in the analysis model but rather as start-up costs of the system.

5.4.7 The LabVIEW Real-Time Cost Model

As LabVIEW does not allow for dynamic priority changes, the LRCM is based on fixed priority scheduling algorithms. The periodic task release by the Timed Loops

aligns itself easily with RMA theory. This section presents the extensions made to deal with the cost of the scheduler, the context switch overheads, and the tasks introduced by the LabVIEW development environment.

Initial work on the LRCM presented in [32] is based on time demand analysis theory. The time demand analysis equations were modified to include context switch overheads by using the overhead estimation technique as described by Stewart. Since then, the model has been further extended to include the research on nonintegrated interrupt event-driven scheduling theory as introduced by Katcher *et al.* to account for the cost of the scheduler.

Figure 5.2 shows an example of three tasks with context switch and scheduler overheads. Priorities are assigned based on increasing periods as in RMA giving $task_1$ the highest priority and $task_3$ the lowest priority. The execution time for $task_i$ is given by c_i.

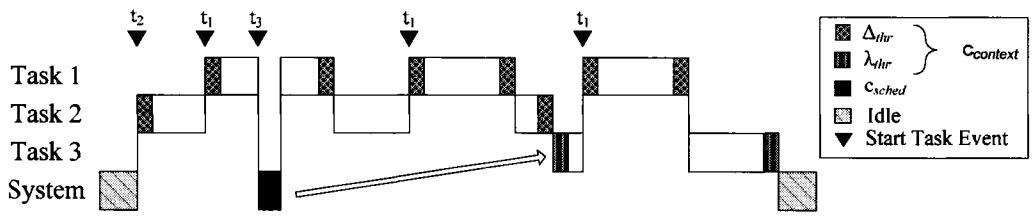

Figure 5.2: LabVIEW Real-Time Cost Model Example.

The context switch overhead ($c_{context}$) is represented by two different values, Δ_{thr} and λ_{thr}. The first value, Δ_{thr}, is used when a new task arrives and pre-empts the current task. Thus, it includes the cost of the scheduler in the context switch. On the other hand, λ_{thr} is used when a context switch occurs to start a task that has already been released, but is obtaining the processor to run for the first time. This does not include the scheduler cost as the cost of scheduling this task (c_{sched}) has

already been accounted for when the task was released (as shown by the large arrow). In the example in Figure 5.2, $task_1$ and $task_2$ use the Δ_{thr} overhead since they begin running as soon as they are released. However, $task_3$ uses the λ_{thr} overhead as it does not get the processor when it is released, but the scheduling has been done at the time of release. For any given $task_i$, it will take an additional $2c_{context}$ to complete its execution when considering the context switch overhead. Based on the application being analyzed, the correct selection of the context switch overhead for each task is made.

The LRCM is a combination of time demand analysis with the nonintegrated interrupt event-driven scheduling theory. Figure 5.2 is similar to the one described by Katcher *et al.* (refer to Section 3.1.3), but instead of determining the overheads using basic units, the context switch overheads are determined using the overhead estimation technique by Stewart (introduced in Section 3.2). Priorities are assigned to all the tasks based on RMA theory, where the task with the shortest period has the highest priority.

Updating the processor utilization function to include the context switch overheads gives:

$$U_{LRCM} = \sum_{i=1}^{n} \frac{c_i + 2c_{context}}{p_i} \tag{5.1}$$

As long as the utilization is less than or equal to 1, the analysis can continue.

Since the LabVIEW scheduler can interrupt a higher priority task when a lower priority task is released, any given $task_i$ can be blocked by all lower priority tasks for maximum of $(n - i) * c_{sched}$ as shown in the nonintegrated interrupt event-driven

scheduling theory. Updating the time-demand analysis equation to include both the context switch overhead estimate and scheduler overhead results in the LRCM:

$$w_i(t) = (c_i + 2c_{context}) + (n-i)c_{sched} + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context}) \text{ , for } 0 < t \leq p_i \quad (5.2)$$

where,

- $(c_i + 2c_{context})$ represents the execution time and associated context switch overhead of the current task being considered $(task_i)$.
- $(n - i)c_{sched}$ accounts for the blocking caused by the scheduling of all lower priority tasks that are released during the execution of $task_i$.
- $\sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context})$ is the summation of all of the execution times and context switch overheads of higher priority tasks than $task_i$ that are released in the interval 0 to time t.

Using Equation 5.2, each task created in the application as well as the three tasks created by the LabVIEW development environment are considered individually. Priorities are assigned in a rate monotonic fashion where the task with the shortest period is assigned the highest priority. As in time demand analysis, tasks are considered in decreasing order of priority, starting with the highest priority task. An instance of $task_i$ that is released at critical instant t_0 is schedulable if the total processor time w_i demanded by $task_i$ and all other higher priority tasks is less than or equal to the supply of available time t at any point before its deadline. Mathematically speaking:

$$w_i(t) \leq t \text{ for some } t \leq p_i \tag{5.3}$$

Once $task_i$ is deemed to be schedulable, analysis of each task continues until $task_n$. Finally, if all n tasks satisfy the time demand criteria then the set of tasks is schedulable.

When applying the LRCM, the lower context switch overhead (λ_{thr}) can be applied to the LabVIEW created tasks as they do not invoke the scheduler. The fact that the scheduler is not being run will also allow for the scheduler blocking cost (c_{sched}) to be omitted for these tasks.

5.5 The LabVIEW Real-Time Cost Model Measurements

The LRCM requires measurements of the context switch and scheduler overheads as well as characterization of each of the tasks introduced by the LabVIEW development environment. In this section, each of the methods used to estimate the overheads is outlined and values of the measurements are presented.

5.5.1 Context Switch Time - With Scheduler

The method used to measure context switch time is based on the overhead estimation technique described by Stewart (refer to Section 3.2). Rather than using a logic analyzer to collect the data, the raw timing data is stored in an array that is saved to disk at the end of the analysis. Timing data is captured from LabVIEW with the

use of the included TickCount VI as part of the Real-Time Module. This VI provides a time-stamp in units of milliseconds, microseconds, or ticks with 8, 16, or 32bits of resolution.

The toggling of bits is simulated by storing an integer representing 0 or 1 and the associated time stamp (output of the TickCount in ticks) into an array. In between the toggling, a busy wait is implemented using a multiplication command within a For Loop. The busy wait gives a more controlled toggling, which is beneficial when debugging the test program. The pseudo-code below gives an outline of how the code is implemented in LabVIEW.

```
1.  begin task
2.    loop n times
3.      get tick count
4.      store value in an array
5.      busy wait
6.      get tick count
7.      store value in an array
8.      busy wait
9.    end loop
10.   get tick count
11.   store value in an array
12. end task
```

When calculating $2\Delta_{thr}$ using the overhead estimation technique, the cost of the final storage of the tick count value into the array (line 11) is included in the overhead

measurement. This is because the TickCount VI has executed before the storage is done and thus not part of the measured t_2. The time for storage into the array can be easily measured by running a small program where a TickCount is taken immediately before and after an array storage is made. Subtracting these two TickCount values gives the time for storage into an array. The cost of this storage must then be subtracted from the calculation outlined in the overhead estimation technique (Equation 3.8) in order to get the correct context switch estimate. Thus, the formula for obtaining $2\Delta_{thr}$ now becomes:

$$2\Delta_{thr} = t_1 - t_2 - t_3 - (\textit{array storage time}) \tag{5.4}$$

Table 5.1 summarizes the values obtained for the context switch overhead ($2\Delta_{thr}$) using two tasks. The average value of $2\Delta_{thr} = 15.10\mu s$ is used by the LRCM for analysis.

min (μs)	*max* (μs)	*mean* (μs)	*std. deviation* (μs)
13.37	21.28	15.10	0.653

Table 5.1: Context switch overhead - with scheduler.

5.5.2 Context Switch Time - Without Scheduler

When more than one task is released at the same time, scheduling of all of the tasks will take place only once. Scheduling also takes place when a lower priority task is released while a higher priority task is running, resulting in blocking of the higher priority task. In both these situations, the context switch overhead (λ_{thr}) is lower

than the context switch overhead when scheduling occurs during the context switch (Δ_{thr}). Thus a measurement of λ_{thr} needs to be made to account for the lower context switch time.

The measurement of λ_{thr} is done by scheduling multiple tasks to be released at the same time. The end time of the first task is then subtracted from the start time of the second task, and so on. As the scheduling is done at the beginning, before the tasks began execution, the result of the calculation provides an estimate of how long the context switch will take if the scheduler is not run. As previously mentioned in the context switch measurement when the scheduler does run, the clock tick captured at the end of the task happens before the storage of the value into an array can take place. The estimate of $2\lambda_{thr}$ between $task_i$ and $task_{i-1}$ is thus obtained by the following formula:

$$2\lambda_{thr} = (start\ time)_i - (end\ time)_{i-1} - (array\ storage\ time) \tag{5.5}$$

Measurements of the context switch overhead are summarized in Table 5.2. During the measurements, it was discovered that the overhead between the highest priority task and the next highest priority task (denoted by λ_{thr1}) is larger than the overhead between any other tasks (denoted by λ_{thr2}). Further discussion with NI support found that this is due to an additional hidden scheduling cost that takes place at the end of the highest priority task. The scheduling that occurs at the beginning of the highest priority task only handles the scheduling for the task itself, and other tasks are scheduled after the first task has completed. The average value for $2\lambda_{thr1}$ is $7.18\mu s$ while the average value for $2\lambda_{thr2}$ is $6.13\mu s$. This represents a difference of over $1\mu s$.

Thus, each of these overheads must be applied to the corresponding situation when applying the LRCM to account for the difference in overheads.

	min (μs)	*max* (μs)	*mean* (μs)	*std. deviation* (μs)
$2\lambda_{thr1}$	7.05	10.57	7.18	0.315
$2\lambda_{thr2}$	5.99	11.94	6.13	0.286

Table 5.2: Context switch overhead - without scheduler.

5.5.3 LabVIEW Tasks

The LabVIEW development environment introduces three tasks to handle the background work required by an application created in LabVIEW. The first task is the ETS Timer Thread. This thread is used by LabVIEW to implement the timing features required to run the application. The second and third tasks introduced by LabVIEW are shown as Unnamed Thread1 and Unnamed Thread2 in the Execution Trace Toolkit logs. According to NI, these tasks poll the Ethernet connection to the Host system periodically to ensure that communication is handled properly.

Measurement of the period and execution times of the three tasks is done manually using the Execution Trace Toolkit. Multiple execution times are gathered one at a time by zooming into the trace log and setting up measurement bars to determine the execution time. The average of these measurements was can be found in Table 5.3. The average execution times of $7.33\mu s$ at a period of $1002\mu s$ for the ETS Timer, of $10.71\mu s$ at a period of $2004\mu s$ for the Unnamed Thread1, and $2.03\mu s$ at a period of $2004\mu s$ for the Unnamed Thread2 are used in the LRCM during evaluation. The measured periods of these tasks is close to the approximate values of $1000\mu s$ for the ETS Time Thread and $2000\mu s$ for the Unnamed Threads as stated by NI technical

support. Differences may be due to the overhead introduced by the Execution Trace Toolkit, or due to errors in aligning the measurement bars in order to determine the actual values.

task	*min (μs)*	*max (μs)*	*mean (μs)*	*std. deviation (μs)*	*period (μs)*
ETS Timer	6.97	8.30	7.33	0.242	1002
Unnamed Thread1	10.49	13.11	10.71	0.384	2004
Unnamed Thread2	1.93	2.87	2.03	0.175	2004

Table 5.3: LabVIEW created tasks overheads.

5.5.4 Scheduler Blocking Overhead

The scheduler in LabVIEW is fired every time a task is released. If a lower priority task is released while a higher priority task is currently running, the scheduler will interrupt the higher priority task. Once the newly released task has been scheduled, the higher priority task will continue its execution.

Measuring the time of the scheduler blocking a higher priority task is done by determining the execution time of a high priority task when it is not interrupted by the release of a lower priority task. This value is then subtracted from the execution time when a higher priority task is blocked by the scheduler due to the release of a lower priority task. Setting up the two scenarios is performed by using the offset feature of the Timed Loop. The offset value determines when the first release of a task is performed relative to the start of the application. By adjusting the offset value of the lower priority task, it can be made to interrupt the higher priority task. The

average scheduler blocking overhead is determined to be $5.09\mu s$ as shown in Table 5.4.

min (μs)	*max* (μs)	*mean* (μs)	*std. deviation* (μs)
1.63	23.51	5.09	0.484

Table 5.4: Scheduler blocking overhead.

5.5.5 LRCM Model Including Overheads

An example of a set of tasks with the measured overhead estimates can be seen in Figure 5.3. $Task_1$ and $task_2$ are released at the same time. This results in the scheduling of both of these tasks at the same time, which is included in Δ_{thr}, the context switch time for $task_1$. As $task_2$ immediately follows $task_1$, the λ_{thr1} overhead is used to account for the context switch time. The arrival of $task_3$ occurs while $task_1$ is running. Thus $task_1$ is interrupted and blocked while the scheduling for $task_3$ takes place. As $task_3$ finally gets the processor after $task_2$ is complete, the lower context switch time of λ_{thr2} is used as the scheduling for $task_3$ has already taken place (c_{sched}). λ_{thr2} can be used instead of λ_{thr1} since $task_3$ does not run immediately after the highest priority task ($task_1$).

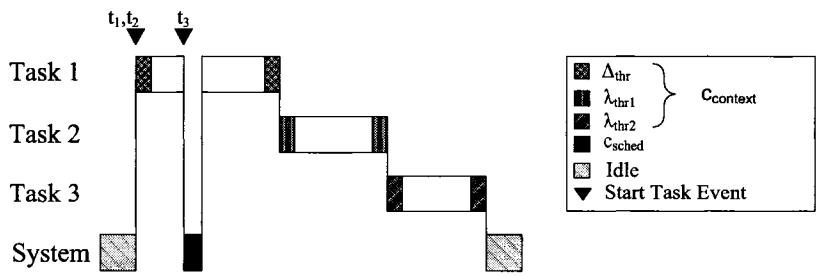

Figure 5.3: LabVIEW Real-Time Cost Model with overheads.

The LRCM described using Equation 5.1 and Equation 5.2 can be used to predict if the set of tasks will meet their deadlines. The only difference is that now $c_{context}$ can take on one of three different values (Δ_{thr}, λ_{thr1}, or λ_{thr2}), depending on the situation encountered.

Chapter 6

Testing of the LabVIEW Real-Time Cost Model

The LabVIEW Real-Time Cost Model presented in Chapter 5 was developed to assist in predicting if an application will meet its timing deadlines. This chapter covers the testing done to determine the accuracy of the model. Section 6.1 provides high-level details of the test setup. It is followed by the description and results of each test case in Section 6.2. In Section 6.3, an analysis of the test results is performed. The chapter concludes with an analysis of the model, including an overview of the strengths and weaknesses of the model and its ability to predict if a set of tasks is schedulable.

6.1 Test Setup

In this section, a description of the method used to test the LRCM and compare it to RMA analysis is provided. Section 6.1.1 provides an overview of the test cases used to analyze the LRCM. The implementation details of the tests cases are described in

Section 6.1.2. Finally, the criteria used to measure failure and compare the LRCM to RMA analysis is detailed in Section 6.1.3.

6.1.1 Overview

Testing of the LRCM is performed using several user-created tasks at the same time. Each task is first run independently on the system for 1000 runs. The average execution time is calculated for use in the analysis. Each run of the task executes the same code (i.e. there are no conditional statements to change the flow of execution). So it is expected that each run of the task should produce the same execution time. However, there is a possibility that other unknown events may occur during a particular run of the task (such as a hardware interrupt). This results in the measurement of the execution time of the task being larger as it also includes the execution time of the interrupt when measured. By averaging the execution over 1000 runs, a more reliable measure of the execution time of just the task itself can be obtained.

Another reason for using the average is to minimize the effect of the state of the cache when the task is run. If the cache contains all of the data required for the run of the task, the execution time of the task will be exceptionally fast compared to the situation where none of the data is in the cache and every instruction results in a cache miss. Since the LRCM does not take into account the overheads caused by a cache miss, averaging the measured execution time values smoothes out the gains of a cache hit versus the expense incurred due to a cache misses.

Using the measured execution times, a prediction can be made to determine if the set of tasks will meet their deadlines. Beginning with Equation 5.1 of the LRCM, the predicted processor utilization is determined. If this is less then 100%, analysis can continue using the updated time demand analysis equation of the LRCM (Equa-

tion 5.2). If the analysis predicts that all task deadlines can be met, the set of tasks are said to be schedulable.

The first ten tests are performed using simulated loading of the system. This is achieved by performing a multiplication operation within a For Loop. To increase the execution time of a task, the number of loops is increased. Each test begins with the situation where the application is predicted to pass. The load on the processor is then gradually increased by adjusting the execution time of the highest priority task. From the initial point of predicted pass, the load is increased until the application fails to meet its deadlines.

The tasks are all started at the same time to ensure worst case loading (i.e. a critical instant as defined in Section 3.1.2). Each time the load is increased, the tasks are allowed to run for at least the period of the lowest priority task. Running the test for the period of the lowest priority task ensures that all tasks have been released at least once during the test. The lowest priority task can be either a user task or a LabVIEW created task. Consistency in testing is maintained by increasing the load at intervals of the least common multiple (LCM) of the periods of the user tasks. This ensured that each time the load is increased, the relative starting point of the set of user tasks is maintained, thus maintaining the critical instant.

The final test case consists of an application based on development currently being done on the CTS system at the NRC-IAR. As the execution times of these tasks are dependent on the application itself, system loading can not be increased by adjusting execution times as in the simulated loading tests. Rather, the system load is increased by adjusting the period of the tasks until failure to meet its deadlines occurs. Further specific details of each test case will be covered in Section 6.2.

6.1.2 Implementation Details

Each test case is run without any user interface. This is done to prevent any overheads introduced by having to display data to the user. As each application created by LabVIEW automatically has a UI (i.e. the front panel), all data objects that were created as part of the block diagram are set to hidden on the front panel. This prevents LabVIEW from creating additional tasks to service the UI, which would add to the complexity of the analysis.

All user tasks in the application are started at the same time by making use of the SynchronizeTimedLoopStarts VI. By passing in the names of each loop, this VI starts all of the user tasks at the same time. As the Timed Loops share a single timing source, the first release of each of the tasks occurs at the same time.

When an application is first executed, LabVIEW requires processing time to set up the Timed Loops and create timing mechanisms to ensure proper timed release of the tasks. This start-up overhead causes the first few iterations of each task to finish late. As the tests are set up to complete their execution even if they are running late, the late task essentially steals time from future releases of the task. This causes all future tasks to finish late, unless there is some slack time in the system which can be used to bring it back to schedule. Thus, the start-up overheads impact the entire run of an application.

The impact of the start-up cost is avoided by using a case structure within each task. A built-in iteration counter in the Timed Loop allows for the developer to determine how many times the task has been released since start-up from within the loop. By using this value as an input to a case structure, the first few iterations of the Timed Loop are set up so that execution of the code will be by-passed. This essentially runs a no load task for the first few iterations. The actual application

starts after a few runs, allowing for the overheads to be complete before it begins doing useful work. The load for each user task is introduced at a point where the iteration number aligns with the LCM of the tasks. This ensures the relative starting point of each task remains consistent with the test set-up. The analysis of the test results also ignores the first iterations when analyzing the data, thus avoiding the start-up overheads.

Within each task, measurement and storage of the start and end times as well as the Finished Late flag is performed. The time is captured by making use of the TickCount VI to get the time stamp at the start and the end of the task. Each of the values are then stored in an array for analysis at the end of the test. Most of the overhead introduced to capture the time and store it in an array is inherently included as part of the measured execution time of the task. However, the storage of the task end time occurs after the end time stamp is taken. Thus, the additional storage cost must be accounted for in some manner. Figure 6.1 shows this additional cost as γ_{timing}. This figure is similar to Figure 5.2, the LRCM example, except that the additional γ_{timing} must be added to each user task. Updating the model to include γ_{timing} gives:

$$U_{LRCM} = \sum_{i=1}^{n} \frac{c_i + 2c_{context} + \gamma_{timing}}{p_i} \tag{6.1}$$

$$\begin{aligned} w_i(t) =& (c_i + 2c_{context} + \gamma_{timing}) + (n-i)c_{sched} + \\ & \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context} + \gamma_{timing}) \text{ , for } 0 < t \leq p_i \end{aligned} \tag{6.2}$$

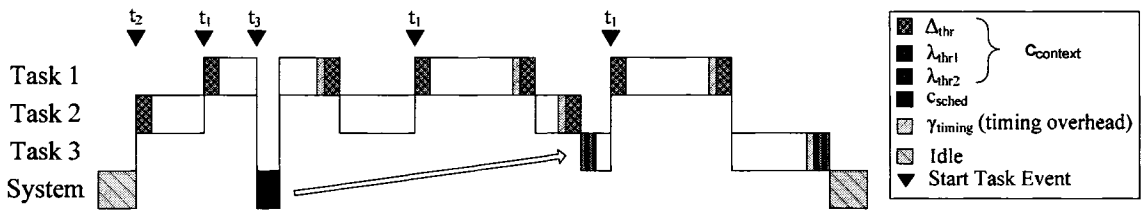

Figure 6.1: LabVIEW Real-Time Cost Model with timing overhead.

Note that γ_{timing} only needs to be included for user-created tasks, and not LabVIEW created tasks. The cost of measuring the start and end time of a task would not be part of a final application; however, the purpose of these tests is to show the model functions correctly. Including the timing overhead ensures that the overhead of the timing mechanism is correctly accounted for when testing the LRCM. Thus for each of the test cases, Equation 6.1 and Equation 6.2 are used instead of Equation 5.1 and Equation 5.2 respectively to ensure that the costs of gathering the test data are accounted for. The average value of γ_{timing} was measured to be $1.79\mu s$.

6.1.3 Measure of Failure

The application is said to fail in meeting its deadlines when the "Finished Late" flag of the lowest priority user task is set (as the lowest priority task would be the first one to fail). Although the LabVIEW tasks are given a priority based on the period of execution in the model, they are actually implemented at the lowest priority in the application. Thus, the LabVIEW tasks will fail to meet their deadlines first before any user tasks fails. Unfortunately, there is no simple method available to determine if one of the tasks created by LabVIEW has failed to meet its deadlines (they do not have a Finished Late flag of their own). The only way to determine if a LabVIEW task has failed to meet its deadline is by making use of the Execution Trace Toolkit.

However, this toolkit introduces its own overheads, which are unknown. In order to ensure testing is not impacted by these overheads, the toolkit is not used. Thus, failure of the lowest priority application task is the only metric available to determine failure.

In order to determine the accuracy of the model, the predicted failures (RMA and LRCM predictions) are compared to the measured failure. Comparison of the LRCM to standard RMA is also performed. As a basis for comparison, the processor utilization is determined using:

$$U = \sum_{user\_tasks} \frac{c_i}{p_i} \tag{6.3}$$

The processor utilization is calculated for each of the RMA predicted failure, the LRCM predicted failure, as well as the measured failure. By determining the processor utilization using Equation 6.3, a direct comparison between the predicted and measured failures can be done.

The percent error in the predicted LRCM and RMA models versus the measured failure is determined by:

$$\%error = \left| \frac{U_{meas\_failure} - U_{pred\_failure}}{U_{meas\_failure}} \right| * 100 \tag{6.4}$$

An example calculation of the RMA and LRCM predictions, as well as the processor utilization and % error is performed for test case 2, and can be found in Appendix A.

6.2 Test Cases

The following sections describe each test in detail. The test cases are presented in increasing complexity of the set of tasks. The final test case is based on development of the CTS system by the NRC-IAR. After the description of the test case, results of both the predicted analysis using the model and the actual running of the application are presented. The analysis of the results is presented in Section 6.3.

6.2.1 Test Case 1: Two user tasks, $task_1$ interrupts $task_2$

The first test consists of two user tasks with the same period. Each task is designed to start with approximately equal processor utilization. $Task_1$ has an offset such that it will always interrupt $task_2$ during execution. Table 6.1 summarizes the starting execution times and periods of both the user and development environment tasks for test case 1. Note that in this test, a critical instant as defined in Section 3.1.2 does not hold as $task_1$ starts offset to $task_2$.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	30.47	7.55	1.79	100	30	100
2	30.26	7.55	1.79	100	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.1: Summary of tasks used in test case 1.

The LRCM is run using the Δ_{thr} context switch overhead for each of the user tasks as shown in Figure 6.2. This is due to the fact that the scheduler has to be run

each time one of the tasks is released.

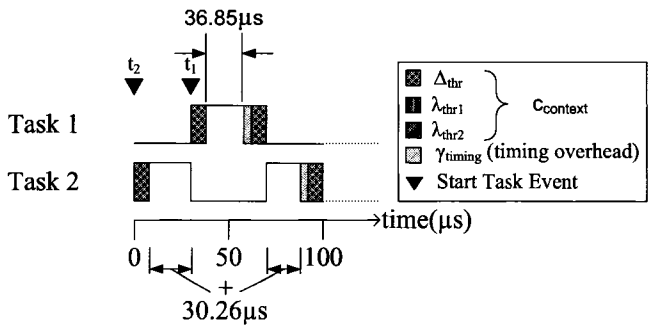

Figure 6.2: Test Case 1: Two user tasks, $task_1$ interrupts $task_2$.

In Table 6.2, the predicted RMA and LRCM failure as well as the measured failure are shown. As was mentioned in Section 6.1.1 the execution time of $task_2$ is held constant while $task_1$ is slowly increased. The predicted LRCM failure occurs when the execution time of $task_1$ is $33.33\mu s$ with a processor utilization of 63.59%. The measured failure occurs at $c_1 = 36.85\mu s$ with a processor utilization of 67.11%. The RMA predicted failure occurred at a much higher processor utilization of 82.84% which is achieved when $c_1 = 52.58\mu s$. As can be seen from Table 6.2, the % error of the RMA prediction is significantly higher than the LRCM error (over 17% higher).

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	52.58	82.84	23.44
LRCM predicted	33.33	63.59	5.25
measured	36.85	67.11	-

Table 6.2: Test Case 1: predicted and measured failure points.

6.2.2 Test Case 2: Two user tasks, both released at the same time

In the second test case, the same tasks are used as in the first test case, but this time the starting offset is removed from $task_1$. The removal of the offset causes both user tasks to start at the same time. This creates a critical instant t_0. Both user tasks are designed to start at approximately equal processor utilization. A summary of the execution times and periods of the tasks in test case 2 is shown in Table 6.3.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	39.48	7.55	1.79	100	0	100
2	30.26	3.58	1.79	100	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.3: Summary of tasks used in test case 2.

As $task_2$ is always released at the same time as $task_1$, the scheduler only runs once. This allows for use of λ_{thr1} for the context switch overhead for $task_2$ when performing analysis using the LRCM (see Figure 6.3). With the lower context switch overhead, higher execution times of the tasks are possible when compared to test case 1 as can be seen in Table 6.4.

The calculation of the values in Table 6.4 can be found in Appendix A. Table 6.4 presents the processor utilization as well as the predicted failure by the LRCM and RMA, and the actual failure of the user tasks. The predicted failure by the LRCM occurs when the execution time of $task_1$ is $41.07\mu s$ and with a processor utilization

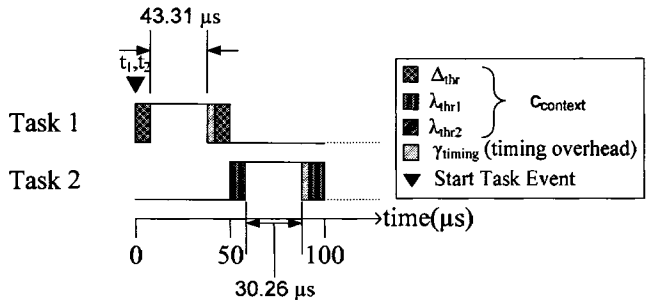

Figure 6.3: Test Case 2: Two user tasks, both released at the same time.

of 71.33%. The measured failure occurs at $c_1 = 43.31\mu s$ with a processor utilization of 73.57%. The RMA predicted failure is the same as in test case 1. However, since the context switch overheads are lower than test case 1, the % error of the RMA vs. the measured failure is much lower in this case (10.84% lower).

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	52.58	82.84	12.60
LRCM predicted	41.07	71.33	3.04
measured	43.31	73.57	-

Table 6.4: Test Case 2: predicted and measured failure points.

6.2.3 Test Case 3: Two user tasks, different periods

The two user tasks in test case 3 have different periods. $Task_1$ has a period of $100\mu s$ and $task_2$ has a period of $300\mu s$. Both the tasks are designed to have execution times so that they almost fully utilize the processor when including the overheads. No offsets are used in this test, so both tasks start at the same time t_0, i.e. a critical instant. A summary of the execution times and periods of both user and LabVIEW tasks is shown in Table 6.5.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	30.47	7.55	1.79	100	0	100
2	134.55	3.58	1.79	300	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.5: Summary of tasks used in test case 3.

As multiple releases of $task_1$ occur during the execution of $task_2$ (as seen in Figure 6.4), this test case verifies that the LRCM will correctly predict if a set of tasks is schedulable when multiple interrupts occur. The Δ_{thr} overhead is used for analysis of $task_1$, while the λ_{thr1} value is used for $task_2$ because the scheduling cost has already been accounted for.

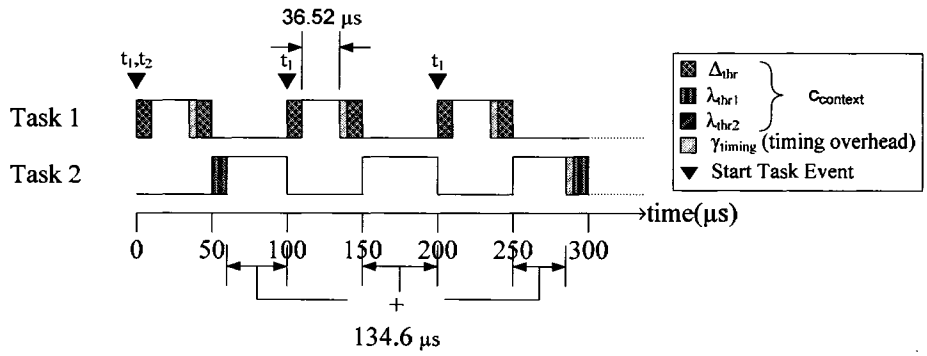

Figure 6.4: Test Case 3: Two user tasks, different periods.

The processor utilization and the predicted as well as measured failure of the tasks can be seen in Table 6.6. In this test, the LRCM predicted failure is calculated to occur at 77.22% utilization with $c_1 = 32.37\mu s$. Measured failure occurs at 81.37%

utilization with $c_1 = 36.52\mu s$. As this test case has two user tasks, as in test case 1 and 2, the RMA predicted failure still occurs at a processor utilization of 82.84% with the resulting $task_1$ execution time of $37.99\mu s$. In this test case, the % error of the RMA prediction when compared to the measured failure is 1.81%, which is lower than the LRCM % error of 5.10%.

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	37.99	82.84	1.81
LRCM predicted	32.37	77.22	5.10
measured	36.52	81.37	-

Table 6.6: Test Case 3: predicted and measured failure points.

6.2.4 Test Case 4: Two user tasks, periods are non-integer multiples

Similar to test case 3, test case 4 also has different periods for both user tasks. However, in this case the period of $task_2$ is no longer an integer multiple of the period of $task_1$. $Task_1$'s execution time begins at $5.37\mu s$ at a period of $50\mu s$. $Task_2$ has an execution time of $40.38\mu s$ with a period of $125\mu s$. Table 6.7 provides a summary of the task execution times and periods for test case 4.

As can be seen in Figure 6.5, the non-integer multiple of the task periods results in $task_2$ being released in the middle of $task_1$ causing the scheduler to interrupt $task_1$. $Task_1$ still has time to meet its deadlines, but it finishes later than it normally would. This causes the start of $task_2$ to be pushed back as well, but due to the extra slack in the timeline, $task_2$ still has time to finish executing before its period expires. $Task_1$ uses the Δ_{thr} context switch overhead, while $task_2$ uses the λ_{thr1} context switch

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	5.37	7.55	1.79	50	0	100
2	40.38	3.58	1.79	125	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.7: Summary of tasks used in test case 4.

overhead since the scheduler is not run at the time of the context switch.

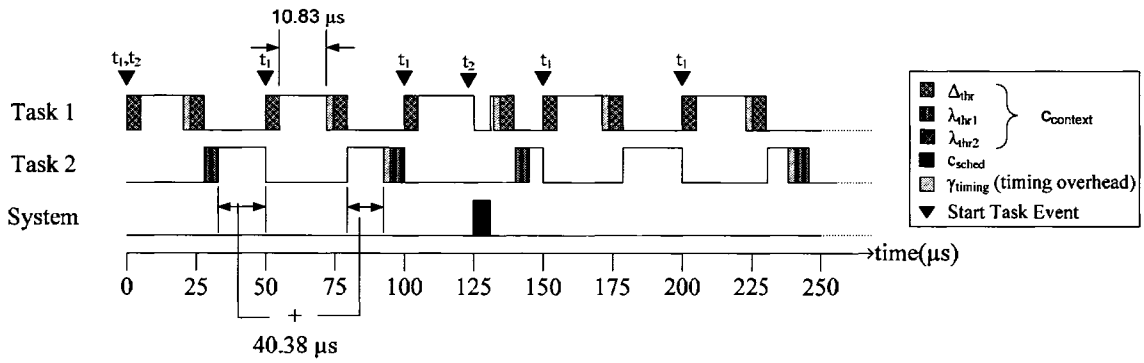

Figure 6.5: Test Case 4: Two user tasks, periods are non-integer multiples of each other.

Table 6.8 show the results of the analysis. As $task_1$'s execution time is increased, the application fails to meet its deadlines. However, as there is still some slack at the end of a hyper-period (the next point at which both tasks are released at the same time), only alternate releases of $task_2$ fail initially. Further increasing the load will cause all occurrences of $task_2$ to fail. The predicted failure of the users tasks by the LRCM occurs at 49.39% utilization when $c_1 = 8.54\mu s$. RMA predicts failure to occur at 82.84% utilization when $c_1 = 25.27\mu s$. The measured failure of the user

tasks occurs at 53.59% utilization with $c_1 = 10.83\mu s$.

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	25.27	82.84	53.55
LRCM predicted	8.54	49.39	8.46
measured	10.83	53.95	-

Table 6.8: Test Case 4: predicted and measured failure points.

6.2.5 Test Case 5: Two user tasks, ETS Timer higher priority

This test case is similar to test case 3 where the two tasks have different periods that are integer multiples of each other. The difference in this test is that now the periods are much longer. By lengthening the period, one of the LabVIEW created tasks, ETS Timer, now has a shorter period than the user-created $task_2$. This gives the ETS Timer thread a higher priority when applying the LRCM as can be seen in Table 6.9. Increasing the period of the user tasks also results in the overheads being a smaller percentage of a task's period. Thus, when calculating the processor utilization using Equation 6.2 for LRCM analysis, the overheads have less impact.

In this test case, $task_1$ is set to start with a execution time of $366.19\mu s$ and a period of $600\mu s$, while $task_2$ has an execution time of $365.57\mu s$ with a much longer period of $1200\mu s$.

From Figure 6.6 the point where the ETS timer thread should theoretically run can be seen, interrupting the execution of $task_2$. This would not be the case in reality as the LabVIEW created tasks have lower priority than any user task. By setting the ETS Timer thread to a higher priority than $task_2$ when performing analysis using the LRCM ensures that enough processor time is provided to allow the LabVIEW task

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	366.19	7.55	1.79	600	0	100
ETS Timer	7.33	3.58	0	1002	0	51
2	365.57	3.58	1.79	1200	0	50
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.9: Summary of tasks used in test case 5.

to complete before its deadline (even though it may finish late in reality). Note that the λ_{thr} context switch overhead is used for the ETS Timer thread as well as $task_2$.

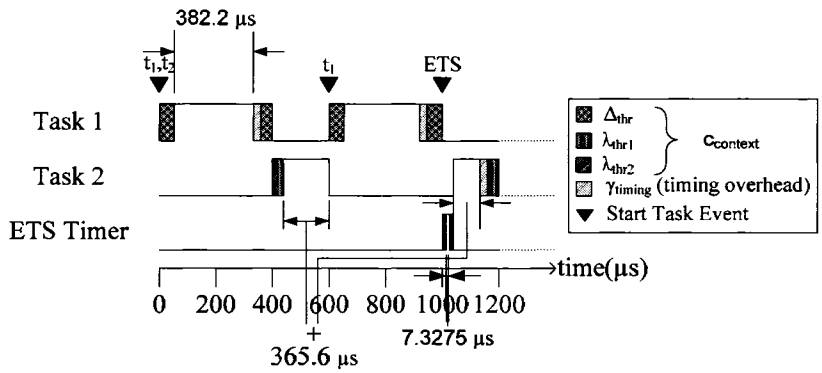

Figure 6.6: Test Case 5: ETS Timer thread has higher priority than $task_2$.

The results of test case 5 can be seen in Table 6.10. The RMA predicted failure of the tasks occurs when $c_1 = 314.27\mu s$ with a processor utilization of 82.84%. The LRCM predicts failure at $c_1 = 368.11\mu s$ with a processor utilization of 91.82%. The measured failure occurs at 94.16% utilization with $c_1 = 382.18\mu s$.

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	314.27	82.84	12.02
LRCM predicted	368.11	91.82	2.49
measured	382.18	94.16	-

Table 6.10: Test Case 5: predicted and measured failure points.

6.2.6 Test Case 6: Three user tasks, $task_1$ interrupts $task_2$, $task_2$ interrupts $task_3$

Test case 6 is the beginning of the tests that involve three user tasks. In this case, a simple three task test is conducted, with each task having the same period. As in test case 1, offsets are once again used to ensure that the higher priority tasks always interrupt the lower priority tasks. As the tasks are not released at the same time, a critical instant as shown by Liu and Layland (refer to Section 3.1.2) does not occur. Each task has a period of $100\mu s$. A summary of the execution times and periods of the user and LabVIEW tasks can be found in Table 6.11.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	8.54	7.55	1.79	100	35	100
2	15.23	7.55	1.79	100	20	75
3	20.23	7.55	1.79	100	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.11: Summary of tasks used in test case 6.

Figure 6.7 shows an example of the execution of the three user tasks. As can be

seen, $task_2$ interrupts the execution of $task_3$, and $task_1$ interrupts the execution of $task_2$. Since each of the tasks is released one at a time, the scheduler must be invoked for each thread. Thus, the Δ_{thr} value must be used for the context switch overhead in all cases when applying the LRCM.

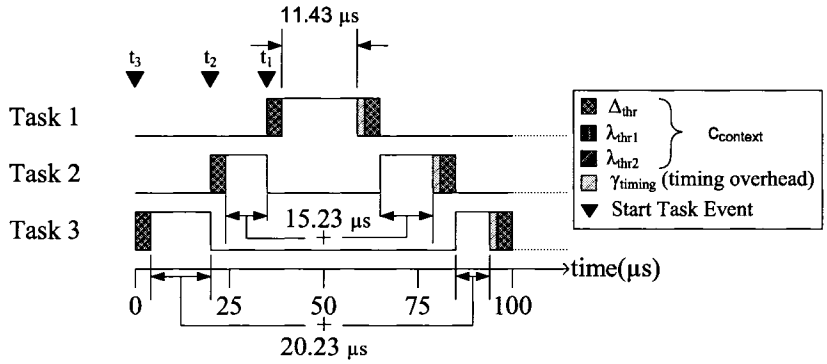

Figure 6.7: Test Case 6: Three user tasks, $task_1$ interrupts $task_2$ and $task_2$ interrupts $task_3$.

In Table 6.12, the processor utilization for the predicted and measured scheduling failures is shown. As can be seen from the table, the RMA predicted failure occurs at a utilization of 77.98%, as there are three user tasks. The LRCM predicted failure is a 45.93% utilization, with $c_1 = 10.48\mu s$. The failure of the user tasks is measured to occur at a utilization of 46.89% when $c_1 = 11.43\mu s$.

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	42.52	77.98	66.30
LRCM predicted	10.48	45.93	2.03
measured	11.43	46.89	-

Table 6.12: Test Case 6: predicted and measured failure points.

6.2.7 Test Case 7: Three user tasks, released at the same time

Similar to test case 6, in test case 7 the same three tasks are used. In this case the offsets are removed so that all of the tasks begin execution at the same time. The launching of all the tasks at the same time creates a critical instant. All three tasks have the same period of $100\mu s$, and start with the same execution time of $20.23\mu s$. A summary of the tasks used in test case 7 is given in Table 6.13.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	20.23	7.55	1.79	100	0	100
2	20.23	3.58	1.79	100	0	75
3	20.23	3.06	1.79	100	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.13: Summary of tasks used in test case 7.

The visual representation of the user tasks can be seen in Figure 6.8. The context switch overheads used in the LRCM to predict failure is different for each task. $Task_1$ uses the Δ_{thr} value since it incurs the cost of scheduling. Since both $task_2$ and $task_3$ are scheduled when they are launched at time zero, the λ_{thr} values can be used. For $task_2$, the λ_{thr1} value is used while for $task_3$ the λ_{thr2} value is used. The reason for different overheads for $task_2$ and $task_3$ is because $task_2$ follows the highest priority task ($task_1$) and must use λ_{thr1}, while $task_3$ does not incur the added overhead of following the highest priority task and can thus use λ_{thr2}.

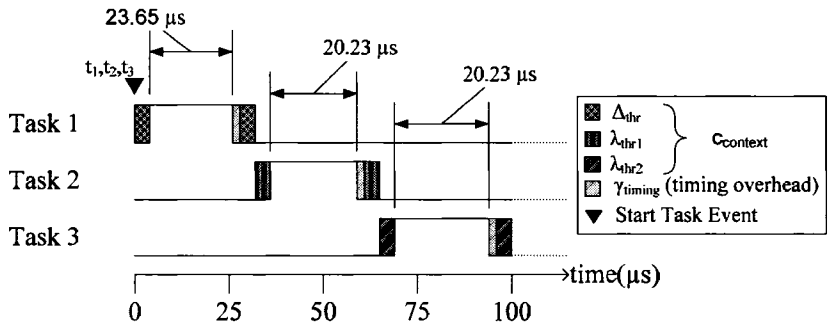

Figure 6.8: Test Case 7: Three tasks, all released at the same time.

The RMA predicted failure occurs at $c_1 = 37.52\mu s$ with a processor utilization of 77.98% while the LRCM predicted failure occurs at $c_1 = 22.37\mu s$ with the processor utilization at 62.82%. The measured failure occurs at $c_1 = 23.65\mu s$ with a processor utilization of 64.11%. This gives a 21.64% error for RMA vs. a 2.00% error for the LRCM. The summary of the results can be found in Table 6.14.

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	37.52	77.98	21.64
LRCM predicted	22.37	62.82	2.00
measured	23.65	64.11	-

Table 6.14: Test Case 7: predicted and measured failure points.

6.2.8 Test Case 8: Three user tasks, each with different periods

In test case 8, the periods of the three tasks are changed so that they are no longer equal. In this scenario, the higher priority tasks will be released more than once in each period of the lower priority tasks. $Task_1$ has a period of $125\mu s$ and starts with

an execution time of $20.49\mu s$. The period of $task_2$ is $250\mu s$ and it has an execution time of $88.26\mu s$. $Task_3$'s period is $500\mu s$ with an execution time of $88.26\mu s$. The execution times and periods of each of the tasks' is summarized in Table 6.15.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	20.49	7.55	1.79	125	0	100
2	88.26	3.58	1.79	250	0	75
3	88.26	3.06	1.79	500	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.15: Summary of tasks used in test case 8.

A visual representation of the user tasks over one period of the lowest priority tasks can be seen in Figure 6.9. The periods of $task_2$ and $task_3$ are an integer multiple of $task_1$'s period. This results in $task_2$ and $task_3$ always being released at the same time as one of the instances of $task_1$. Because of this, scheduling of the $task_2$ and $task_3$ always occurs at the same time as $task_1$. This allows for use of λ_{thr1} and λ_{thr2} context switch overheads for $task_2$ and $task_3$ respectively when predicting failure to meet deadlines using the LRCM.

The results of the test execution can be seen in Table 6.16. The predicted failure occurs at 77.98% utilization with $c_1 = 31.28\mu s$ for RMA and 78.77% utilization with $c_1 = 32.27\mu s$ when using the LRCM. Measured failure occurs when $c_1 = 34.49\mu s$ with a processor utilization of 80.55%. Thus, the RMA error is 3.19% while the LRCM error is 2.20%.

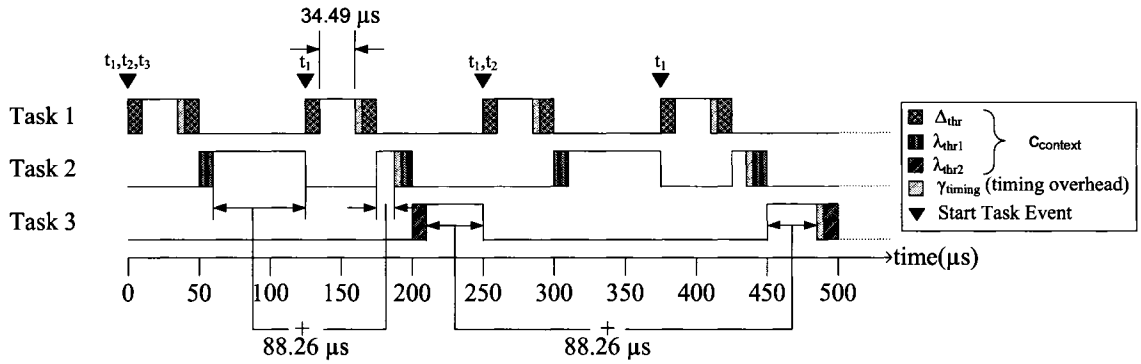

Figure 6.9: Test Case 8: Three user tasks, each have different periods.

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	31.28	77.98	3.19
LRCM predicted	32.27	78.77	2.20
measured	34.49	80.55	-

Table 6.16: Test Case 8: predicted and measured failure points.

6.2.9 Test Case 9: Three user tasks, periods are non-integer multiples

The three tasks used in test case 9 have periods which are non-integer multiples of each other. This causes lower priority tasks to be released while higher priority tasks are executing, as their periods are no longer aligned. In this situation, it can be seen how the blocking term in the LRCM comes into play as lower priority tasks will now interrupt higher priority tasks in order to be scheduled.

Table 6.17 provides a summary of each of the tasks' parameters in this test case. $Task_1$ and $task_2$ have periods of $100\mu s$ and $150\mu s$ respectively. The execution time of $task_1$ begins at $25.81\mu s$, while $task_2$'s execution time is held steady at $20.23\mu s$. $Task_3$'s period is $350\mu s$ and has an execution time of $88.26\mu s$.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	25.81	7.55	1.79	100	0	100
2	20.23	3.58	1.79	150	0	75
3	88.26	3.06	1.79	350	0	50
ETS Timer	7.33	3.58	0	1002	0	49
Unnamed Thread1	10.71	3.58	0	2004	0	48
Unnamed Thread2	2.03	3.58	0	2004	0	47

Table 6.17: Summary of tasks used in test case 9.

Table 6.18 gives a summary of the results of test case 9. The RMA predicted failure occurs at a processor utilization of 77.98%, while the LRCM predicts failure to occur at 68.98%. The measured failure occurs at 69.61% processor utilization resulting in RMA having an error of 12.01% and the LRCM having an error of 0.91%.

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	39.28	77.98	12.01
LRCM predicted	30.28	68.98	0.91
measured	30.91	69.61	-

Table 6.18: Test Case 9: predicted and measured failure points.

6.2.10 Test Case 10: Three user tasks, LabVIEW tasks have higher priority than user tasks

Test case 10 adds more complexity to the system by increasing the periods of the user tasks. By using RMA priority assignment, the priorities of the LabVIEW created tasks are no longer lower priority than the user-created tasks. $Task_1$ and $task_2$ have periods of $1500\mu s$ and $2000\mu s$ respectively, which is lower than the period of the

ETS timer thread with a period of $1002\mu s$. Thus, the ETS Timer thread has the highest priority since it has the shortest period. The two Unnamed Threads, both with periods of $2004\mu s$, also have greater priority than $task_3$ which has a period of $3000\mu s$. The execution times of $task_1$, $task_2$ and $task_3$ are $641.21\mu s$, $481.01\mu s$, and $544.79\mu s$ respectively. The parameters of all of the tasks in test case 10 are shown in Table 6.19.

task	$c_i(\mu s)$	$c_{context}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
ETS Timer	7.33	3.58	0	1002	0	101
1	641.21	7.55	1.79	1500	0	100
2	481.01	3.58	1.79	2000	0	75
Unnamed Thread1	10.71	3.58	0	2004	0	74
Unnamed Thread2	2.03	3.58	0	2004	0	73
3	544.79	3.06	1.79	3000	0	50

Table 6.19: Summary of tasks used in test case 10.

In Table 6.20 the results of giving the LabVIEW created tasks a higher priority can be seen. The LRCM predicts the scheduling will fail at a processor utilization of 86.75% and RMA predicts failure at 77.98%. The measured failure occurs at 99.46% processor utilization. This is results in an error of 17.77% for RMA and 8.52% for the LRCM. The execution time of $task_1$ (which is the only one being adjusted) ranges from 536.49 μs at the RMA predicted failure to $789.22\mu s$ at the measured failure.

6.2.11 Test Case 11: CTS Application

The final test conducted is based on current development being done on the CTS system at the NRC-IAR. This project consists of controlling a redundant robotic

failure	$c_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	536.49	77.98	17.77
LRCM predicted	668.07	86.75	8.52
measured	789.22	94.82	-

Table 6.20: Test Case 10: predicted and measured failure points.

manipulator with eight joints. As the robot had not been built at the time of testing, the calculated robot dynamics are used to simulate the actual movement and to provide feedback to the control system.

The application consists of three independent tasks. The first task (t_1) is the control module. This module implements eight PID control loops to move and maintain the positioning of the robot. The output of the PID control feeds the dynamic model of the robot, which in turn provides feedback on its current position. The final design for the control module will include a more advanced controller which is not implemented at this stage. The dynamics of the robot will also be replaced by the actual robot in the final implementation.

The second task (t_2) consists of the trajectory generator. Based on an input position, the trajectory generator plans the path the robot will take to reach its final destination. Each iteration of the trajectory generator (i.e. each release of the task) supplies the next position on the path the robot will move to. The results of the trajectory generation are then supplied as input to the control module to position the robot.

The trajectory generator is implemented using a state machine, each state having a different execution time. In order to keep the execution time consistent for the purposes of testing, the trajectory generator is held in the state which calculates the next step, or position, the robot needs to move to in order to reach the final

destination in a smooth manner. The state used to calculate the next position has the longest execution time, thus successful scheduling of the trajectory task while in this state will guarantee all the shorter states will also be able to meet their deadlines. The state was held by setting the end position of the robot to a position far away, such that the test case completes its execution before the robot reaches its final position. Since the tests conducted were very short, the program running on the host system is not run during testing as user interaction is too slow. Instead, the reflective memory is pre-configured to hold the final destination of the robot used in the test case.

The final task (t_3), called the supervisory module, provides a mechanism to receive input and display output to the user. This is achieved by the use of reflective memory linked to a host system running the user interface. Positioning input provided by the user is sent to the trajectory generator to create the necessary path the robot will follow. Feedback on the robot's position and current state of the requested move are passed back to the user, once again using reflective memory. This transfer of information to and from the host system is conducted by the supervisory module during every period in its execution. In future design iterations of the system, the supervisory module will also take on other responsibilities not included at this stage of development. A block diagram of the CTS control application being tested is shown in Figure 6.10.

In order to pass data between the three tasks, a shared variable system was set up. Consistency among the data is maintained by the use of two sets of variables and a flag. When a lower priority task is the writer of data, it will write to the location that the flag does not currently indicate. Once the write is complete, the flag is updated atomically indicating to the higher priority task that it should now read the second set of variables. The higher priority task will always read the flag first, and then

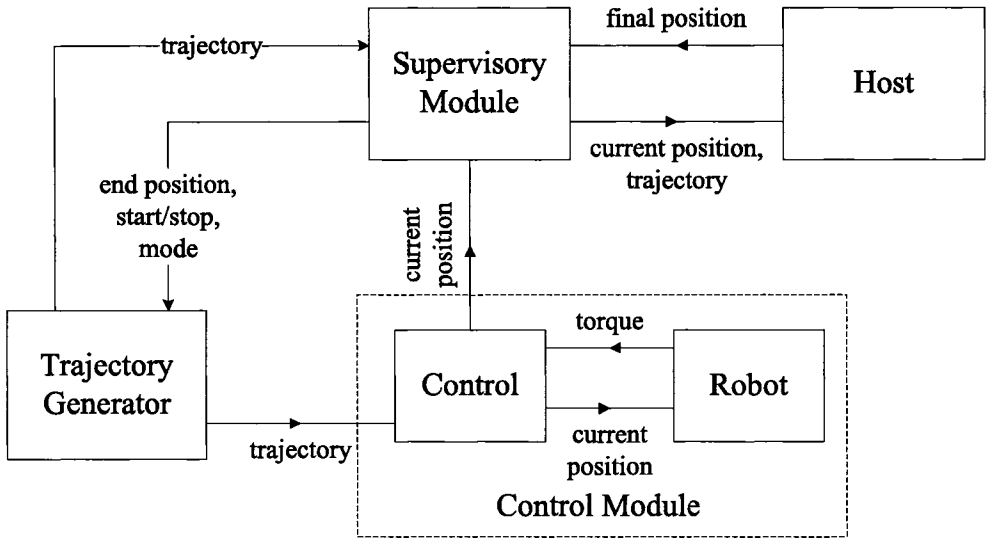

Figure 6.10: Block diagram of the CTS control application.

read from the set of variables indicated by the flag, preventing it from reading the variables that may be in the process of updating.

If a higher priority task is the writer of data, it will write to the location pointed to by the flag. When the lower priority task wants to perform a read, it will first toggle the flag and then perform the read on the set of variables that was previously indicated by the flag. This will ensure that the higher priority tasks will never write to the set of variables while they are being read.

The periods of the tasks are initially set to values discussed among the CTS designers as ones that will be used on the final system. The period of the control module (t_1) is set to $1000\mu s$, while both the trajectory generator (t_2) and supervisory module (t_3) periods are set to $10000\mu s$. The execution time of each of the tasks is determined based on an average of 1000 runs of each task. After running the application, it was found that the periods of the tasks were overly pessimistic since

the application did not contain all of the code that would be run in the final system. Thus, the periods of trajectory generator and supervisory module was reduced to $2500\mu s$ and $5000\mu s$ respectively. A summary of each of the tasks' parameters used in the test can be seen in Table 6.21.

task	c_i (μs)	p_i (μs)	priority
1	84.401	1000	100
ETS Timer	7.33	1002	99
Unnamed Thread1	10.71	2004	98
Unnamed Thread2	2.03	2004	97
2	1920.555	2500	50
3	264.098	5000	25

Table 6.21: Summary of tasks used in the CTS application test case.

As the execution time of each of the tasks is fixed by the application, the processor utilization is increased by changing the period of the control module. The results of the test can be seen in Table 6.22. The period of the control module is decreased in $1\mu s$ intervals while holding the periods of the trajectory generator at $2500\mu s$ and the supervisory module at $5000\mu s$. As can be seen in Table 6.22, the LRCM predicted failure occurs when the control module has a period of $825\mu s$ with the total processor utilization of 92.34%. The measured failure finally occurs when the control module period is $815\mu s$ and the total processor utilization is at 92.46%. The RMA predicted failure at a processor utilization of 77.98% is lower than the processor utilization of just the trajectory generator and supervisory module (calculated to be 82.10%). This means that the processor utilization of just two of the three user-created tasks is already greater than the point at which failure will occur according to RMA. Thus, according to RMA, the set of tasks will not meet their deadlines no matter what period is selected for the control module.

failure	$p_1(\mu s)$	*U (%)*	*% error (%)*
RMA predicted	-	77.98	15.67
LRCM predicted	825	92.34	0.14
measured	815	92.46	-

Table 6.22: Results of the CTS test application.

6.3 Analysis of Test Case Results

	U (%)			*% error*	
Test Case	*RMA*	*LRCM*	*Measured*	*RMA*	*LRCM*
1	82.84	63.59	67.11	23.44	5.25
2	82.84	71.33	73.57	12.60	3.04
3	82.84	77.22	81.37	1.81	5.10
4	82.84	49.39	53.95	53.55	8.46
5	82.84	91.82	94.16	12.02	2.49
6	77.98	45.93	46.89	66.30	2.03
7	77.98	62.82	64.11	21.64	2.00
8	77.98	78.77	80.55	3.19	2.20
9	77.98	68.98	69.61	12.01	0.91
10	77.98	86.75	94.82	17.77	8.52
CTS	77.98	92.34	92.46	15.67	0.14
Average % error				21.82	3.65

Table 6.23: Summary of test case results.

The summary of the results of all of the test cases can be found in Table 6.23. In each of the test cases, the LRCM predicted failure occurs at a processor utilization which is less than that of the measured failure. Part of this can be attributed to the criteria being used to measure a failure. The LRCM considers a failure to be a missed deadline by either a user task or a LabVIEW created task. However, no simple means of detecting a missed deadline of a LabVIEW created task is available while executing the application. The only way to detect a LabVIEW task missed deadline

is by making use of the Execution Trace Toolkit, which adds additional overheads to the application. Thus, the measure of failure being used is the "Finished Late" flag of a user task. Since the LabVIEW scheduler ensures user tasks are higher priority than LabVIEW tasks, the LabVIEW tasks would be the first to fail. By the time the measured failure criterion has been triggered, all LabVIEW created tasks will have already failed.

Another reason for the LRCM predicted failure utilization being lower than the measured failure processor utilization may be due to the pessimistic nature of the measured overheads. The overhead values of the LabVIEW tasks were measured using the Execution Trace Toolkit. This toolkit allows the developer to gather data during run-time on all the tasks running on the system. The cost of running the toolkit is not known, and does not explicitly show up in the trace. As the measured times of the tasks are longer when running the Execution Trace Toolkit, the cost is assumed to be added to the execution time of the tasks. Thus, measurements of the execution times of the LabVIEW tasks are most likely higher than what they would be when running an application.

The context switch overhead may also contribute to the difference between the predicted and measured failure. As the overhead is added to the execution time of the tasks by the LRCM when performing prediction analysis (see Equation 5.1 and Equation 5.2), any error in the measured context switch value will result in an error in the predicted LRCM failure.

The RMA relation introduced by Liu and Layland (Equation 3.1) provides the least upper bound on processor utilization. This upper bound is dependent on the number of tasks running on the system. The relation is supposed to be pessimistic in nature. However, as can be seen in the test results, certain test cases have the RMA

prediction as more optimistic (i.e. the processor utilization at RMA least upper bound is higher than the processor utilization at the measured failure). The reason for the optimism in certain cases and pessimism in other cases is dependent on the particular test case. The influence of the test case can be seen by looking at the following examples.

Consider two user tasks with the parameters given in Table 6.24 (Example 1). The period of $task_1$ is $75\mu s$ with an execution time of $20\mu s$, and the period of $task_2$ is $350\mu s$ with an execution time of $100\mu s$. The overhead associated with each task is $7.55\mu s$. Using Liu and Laylands relation, the RMA least upper bound on processor utilization is determined to be 82.8% based on two tasks. To determine the measured failure, the execution time of $task_1$ (the higher priority task) is increased until the first job of $task_2$ no longer has enough available processor time in order to complete its execution on time (i.e. fails to meet its deadline). The failure occurs when the execution time of $task_1$ is $35.2\mu s$, resulting in the measured failure occurring at an approximate processor utilization of 75.6%. This measured value of 75.6% does not including the overheads, which account for 24.4% of the processor utilization. The measured failure occurs below the RMA least upper bound since the overheads are not accounted for in the RMA model. This makes the RMA prediction optimistic when compared to the measured failure.

$task$	$c_i(\mu s)$	$p_i(\mu s)$	$c_{context}(\mu s)$
1	20	75	7.55
2	100	350	7.55

Table 6.24: Example 1 - set of tasks where RMA is optimistic.

Now, consider a second example (Example 2) where the period of $task_1$ is increased to $200\mu s$ (as shown in Table 6.25) with all other parameters remaining the same as

in Example 1. As there are still only two user tasks, the RMA least upper bound on processor utilization remains at 82.8%. However, this time when the execution time of $task_1$ is increased to until failure, the measured failure occurs at approximately 88.1% processor utilization with $task_1$'s execution time at 119.1μs. At this point the first job of $task_2$ has been starved of the processor and fails to meet its deadline. Due to the longer period of $task_1$, the overheads only account for 11.9% of the processor utilization. The RMA least upper bound is pessimistic, but only because the processor utilization of the overheads is not large enough to create the cross over from pessimism to optimism.

$task$	$c_i(\mu s)$	$p_i(\mu s)$	$c_{context}(\mu s)$
1	40	200	7.55
2	100	350	7.55

Table 6.25: Example 2 - set of tasks where RMA is pessimistic.

Figure 6.11 visually illustrates the effect of changing the period of $task_1$. The plot shows the period of $task_1$ versus the processor utilization of the tasks at LRCM predicted failure, measured failure, and the RMA least upper bound on processor utilization. The point at which failure occurs in the Examples 1 and 2 above is highlighted on the plot. As can be seen in this plot, the RMA least upper bound on processor utilization remains fixed no matter what the period of the user tasks. This is because the least upper bound is only based on the number of tasks. This differs from the processor utilization for the LRCM predicted failure and the measured failure which both increase as the period of $task_1$ is increased. This is due to the fact that increasing the period of the user tasks results in a lower penalty from the overheads. Thus, RMA can be optimistic or pessimistic depending on parameters of the test case. This makes RMA a poor predictor of failure when overheads are considered,

unlike the LRCM which takes into account the overheads.

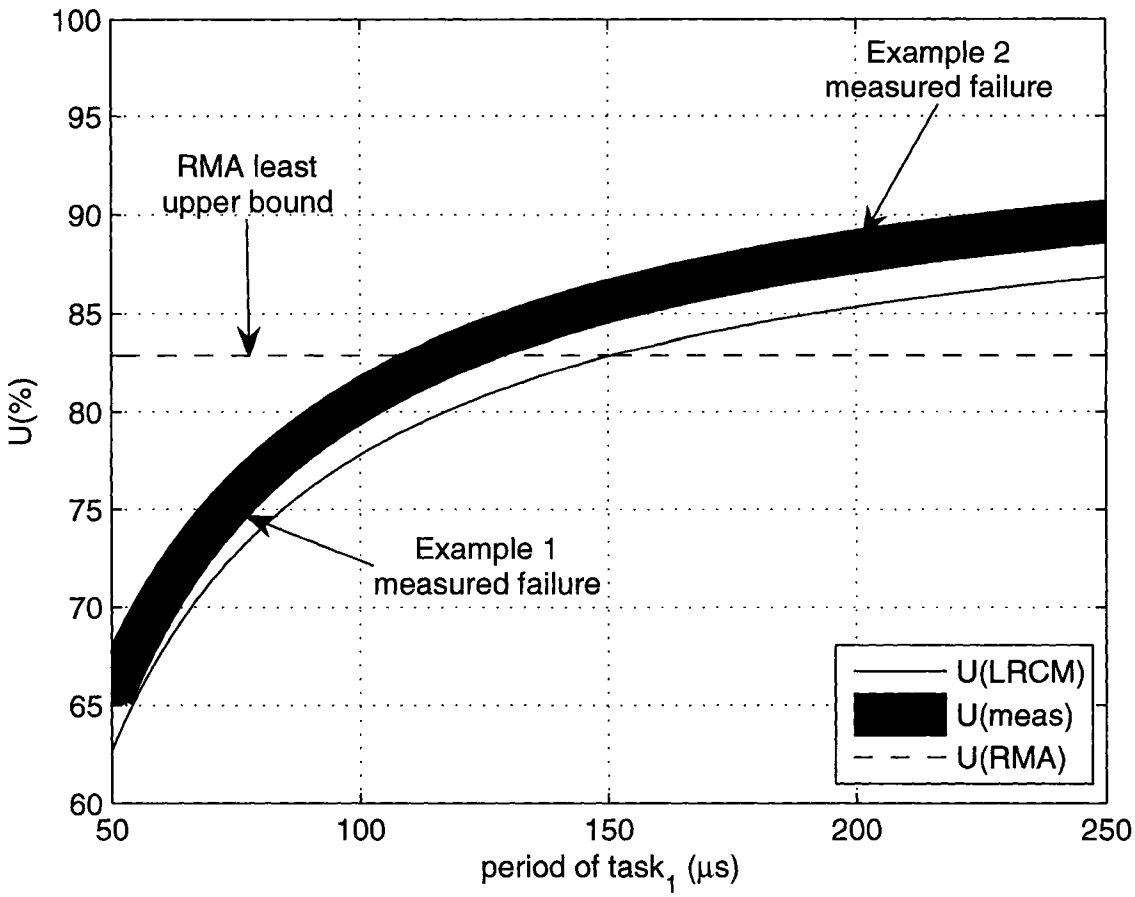

Figure 6.11: The effects of task period on the predicted and measured failures.

When comparing test cases 1 and 2, as well as test cases 6 and 7, it is interesting to note that while the execution times and periods of the tasks were set up to be the same, test cases 2 and 7 were able to achieve higher execution times than test cases 1 and 6. This is due to the fact that in test cases 1 and 6, offsets are used to ensure the higher priority tasks interrupt lower priority tasks. This causes the scheduler to be run for every task. By starting the tasks at the same time in test cases 2 and

7, the context switch overhead for the lower priority tasks is modeled as λ_{thr1} and λ_{thr2}. This is much lower than the context switch overhead used to model the tasks in test cases 1 and 6 (i.e. Δ_{thr}) which includes the scheduling costs. The reduction in overhead allows for more of the processor to be available for task execution, as seen in test cases 2 and 7.

The testing of the real-world application shows how the LRCM can also predict failure by adjusting the periods of the tasks. As the execution times of the tasks are fixed, the option of adjusting the periods plays an important role in predicting scheduling failure. The results of the CTS application test are similar to the tests with simulated loading, as the predicted failure occurs before the measured failure when small incremental reductions in the period are made.

One important issue that bears mentioning was the processor overheating while running the test cases. The Pentium 4 mobile chip used on the target platform slows down when the temperature hits a certain point. Repeatedly loading the chip for 100% processor utilization causes the chip to overheat and slow down. During early testing a measurable slowdown in processing was noticed when tests that previously met their deadlines started to fail. As the execution time of each task was stored during testing (refer to Section 6.1.2), the comparison of execution times showed an increase as the tests continued. In order to keep the tests consistent, testing took place on a cool chip. When effects of overheating were noticed, the CPU was allowed to cool down before continuing.

6.4 Analysis of the LabVIEW Real-Time Cost Model

The results of the simulated loading tests showed that the LRCM errors ranged from 0.91% to 8.52% depending on the set of tasks, while the RMA error ranged from 1.81% to 66.30%. Testing of the practical application gave an error of 0.14% for the LRCM while the RMA error was at 15.7%. The average RMA error for all tests was 21.82% while the LRCM had an average error of 3.65%. The larger variation in RMA error is due to the fact the prediction of failure is solely based on the number of tasks. If the execution costs are either very high or very low compared to the execution time of the tasks, the RMA prediction error can become quite large. In certain test scenarios, the RMA error was lower then the LRCM error, but this is seen as coincidental, as the failure of the tasks happened to occur at a point very close to the processor utilization, which RMA predicted based on the number of tasks.

Based on the tests conducted, the LRCM is able to predict the failure of user-created tasks better than RMA alone. The model was tested by adjusting both the execution times of tasks as well as the periods of tasks in order to change the utilization of the processor. Testing of the CTS application shows how the LRCM can be used to assist in determining if an application will meet its deadlines. If an application is predicted to fail, the developer can adjust the periods of the tasks, or look at methods to optimize the tasks to decrease their execution times. Using the LRCM, if task execution times are known, predictions can be made without running the application.

The results of the tests have given more insight into the performance of the LRCM. From the tests performed using the offset, it was found that starting the execution

of all tasks at the same time results in lower overheads. This allowed more time for execution of user tasks. The point defined as the critical instant, where a task is released at the same time as all higher priority task (as was shown by Liu and Layland to create worst case loading - refer to Section 3.1.2), did not actually create the greatest processor loading. The processor loading was actually higher when all the tasks were started at offsets to each other, as seen when comparing test cases 1 and 6 to test cases 2 and 7. This effect was also noted by Burns and Wellings (refer to Section 3.3.2). Further investigation of this phasing issue needs to be performed in order to determine the actual critical instant. As the tests conducted on the LRCM in Section 6.2 may not have been done for worst case loading, further testing of the LRCM should be conducted with consideration to the phasing issue.

The LRCM assigns priorities to all of the tasks using the rate monotonic approach. However, the LabVIEW created tasks are actually implemented as the lowest priority when running on the system. When performing analysis using the LRCM, assigning the LabVIEW created tasks with an artificially inflated priority does allow for inclusion of the overheads of the development environment, but does not actually guarantee that their deadlines will be met. As the period and execution time of each of the LabVIEW tasks is fixed, adjusting the model to include the cost of the LabVIEW tasks as a percentage of the overall processor utilization may provide a better model of the overall system.

While the LRCM prediction has a lower average %error than using RMA alone, improvements can still be made by refining the measured estimates of the overheads. Using the measured values from the Execution Trace Toolkit includes overheads introduced by the toolkit itself. The values obtained using the Execution Trace Toolkit also introduce a measurement error as no automated method exists to extract the

start and end times of the tasks, and all measurements are done by hand. The display of execution times can only occur at the pixel resolution of the screen; thus, rounding errors based on screen resolution are introduced. The zoom factor also determines how fine the measurement resolution is, as measurement bars need to be placed at the start and end times of a task in order to determine the execution time. Thus, the measured values have some introduced error and may be pessimistic due to the toolkit overheads.

While comparisons of the LRCM predicted failure are made to the measured failure by using the "Finished Late" flag of the lowest priority user task, a better metric for measured failure would provide more insight into the accuracy of the model. By the time the "Finished Late" flag of the lowest priority user task is set, the LabVIEW created tasks (which run at the lowest priority) have already been starved of the processor and thus failed to meet their deadlines. Even if the LabVIEW created tasks had a "Finished Late" flag, the flag would not be set as the LabVIEW tasks would never have a chance to run. A mechanism that could be used to determine LabVIEW task failure would allow for the measured failure to take into account the overheads introduced by the development environment. This would give a better insight to the accuracy of the LRCM when comparing the measured and predicted failures.

Chapter 7

Conclusions and Future Directions

The research conducted laid the foundation for creating a bridge to allow for theoretical research on real-time systems analysis to be applied to industrial systems. The goal was to show that theoretical research can be applied to industry calibre real-time development tools in order to obtain better a priori prediction than RMA alone. This was achieved by analyzing the LabVIEW development environment with the additional Real-Time Module to determine the execution costs introduced by the scheduler, context switch, and the development environment. The resulting model was then compared to analysis performed using RMA alone.

This chapter provides the conclusions of the research in Section 7.1. The contributions of the work are re-stated in Section 7.2. Finally, in Section 7.3, recommendations for directions of further study are highlighted.

7.1 Conclusions

The LabVIEW Real-Time Cost Model was created to assist in determining if it is possible to align research theory with industry calibre real-time development tools to perform better a priori prediction compared to RMA alone. The LRCM profiled the LabVIEW development environment with the addition of the Real-Time Module. The model included the overheads introduced by the tool for context switching between tasks, scheduling of tasks, and performance of background work to ensure applications created in the environment could run in a real-time environment. Given a set of user tasks with known execution times and periods, analysis to determine if their timing deadlines can be achieved is possible.

From the results of the test cases conducted, the LRCM had a 3.84% error versus the 21.82% error for RMA alone. However, the tests conducted assumed that the critical instant is the point at which a task is started at the same time as all other higher priority tasks, as shown by Liu and Layland. As it turns out, this is not actually the point of worst case loading when the execution costs of the system are introduced. Thus, further testing on the model is required in order to determine its performance under worst case loading versus RMA.

While the tests did not necessarily create worst case loading of the system, the LRCM prediction of failure is still closer to the measured failure on average when compared to RMA prediction alone. Thus, further study into the profiling of COTS development environments is warranted, and should be continued. This will allow for a better set of tools to assist developers in creating real-time applications that are not only functionally correct, but will also able to meet their strict timing deadlines.

7.2 Contributions

This thesis is based on theoretical research in the area of scheduling analysis using RMA. The theory is applied to the LabVIEW development environment with the additional Real-Time Module to produce a model that will provide the developer with more confidence that a set of tasks will meet their deadlines. As a result of this research, several contributions have been made in advancing theory in order to perform analysis of practical applications. The major contributions are listed below.

- In Section 5.1, a breakdown of the characteristics of a development environment that need to be considered when applying RMA theory to profile COTS real-time development environment is provided. Using this guideline, a development environment can be analyzed in order to create a model based on RMA theory for a priori analysis of a set of tasks.

- This research also conducts an in-depth analysis of the overheads of LabVIEW development environment with the Real-Time Module running on a specified real-time target (Section 5.5). The overheads relating to scheduling and context switch, as well as to cost of the additional tasks introduced by the development environment, are identified and measured for inclusion in the LRCM.

- A direct result of the research is the creation of the LabVIEW Real-Time Cost Model that can be used by LabVIEW developers to fit an application to a theoretical algorithm and gain confidence that the application will meet its deadlines (Section 5.4.7). The use of the LRCM will benefit development projects such as the CTS system being developed at the NRC-IAR.

- In creating the LRCM, the development environment was approached as a black

box. The overheads on the system were determined by making use of only the tools provided by the vendor (in this case NI) and measurement techniques outlined by Stewart (refer to Section 3.2). Creation of the resulting LRCM shows that a model can be created without knowing the inner workings of a development environment.

- The LRCM is created using current theoretical research based on RMA. While RMA alone has a number of simplistic assumptions that make analysis of practical systems difficult, theoretical research on relaxing theses assumptions has allowed for the creation of a model without some of the underlying assumptions. Testing of the LRCM has shown that high-level development environments can be modeled using current theoretical methods in order to assist in determining if a set of tasks is schedulable.

- The LRCM was created by combining theoretical research on time demand analysis, nonintegrated interrupt event-driven scheduling, and overhead estimation techniques. The resulting model is a practical solution to scheduling analysis for industrial applications.

7.3 Future Directions

The LabVIEW Real-Time Cost Model provides a basic analysis of the LabVIEW development environment with the Real-Time Module. Further work on the model is needed to give more accurate analysis of the tool. Some of the proposed areas for future study are outlined below.

While testing of the LRCM did show that more confidence can be achieved com-

pared to RMA alone in determining if a set of tasks can meet their deadlines, the tests conducted did not create worst case loading of the system. Introduction of execution costs of the system in the model resulted in the critical instant of each task to be offset to each other. This was in conflict with the assumed critical instant (as shown by Liu and Layland) when a task is started at the same time as all other higher priority tasks (i.e. all tasks share a critical instant at time t_0). Due to the complexity of this phasing issue, further research into determining the critical instant, and thus the worst-case execution time, needs to be performed. The LRCM can then be tested again to ensure that it still performs better than RMA alone.

The LRCM used measured estimates of the overheads introduced by the system. In order to achieve better overhead estimates, a better set of timing tools needs to be developed. The Execution Trace Toolkit from NI is a possible tool that can be used by extending it capabilities. A feature that would be beneficial is the inclusion of a utility that allows for the execution time of a task to be extracted automatically. Currently the tool requires tedious measurement by hand which can be influenced by the zoom factor, the pixel resolution of the screen, and by the hand-eye coordination of the user. Also, the cost of running the trace toolkit needs to be established and separated from the measured times of tasks.

In determining the overheads of the system and the execution times of the tasks, the LRCM made use of average case values. In order to provide more confidence in the model's results, the measurements need to be extended to worst case values. This will allow for a more complete analysis in determining if a set of task will be schedulable in all possible scenarios. In the past, this has proven to be a very difficult problem to solve and may require significant resources and collaboration with the tool vendors.

The LRCM only relaxed one of the assumptions imposed by RMA. Future work should include relaxation of some of the other assumptions. In newer versions of the LabVIEW Real-Time Module, the criteria that the deadline of a task is the same at the start of the next period of the task has been dropped. By extending the LRCM, relaxation of this assumption can be included in the model. Also, as seen in the CTS application test case, the sharing of data is an important aspect in practical industrial applications. Updating the model to relax this assumption would be highly beneficial as well.

The effects of caching were assumed to be minimal and were not considered by the LRCM. If an application is small and can reside entirely within the processor cache, no cache miss would take place. However, as an application increases in size all data may not be held in cache and the cost of a cache miss requiring updates from main memory will introduce an overhead. Thus, future work is required to model the costs associated with memory cache.

Finally, this research focused on the modelling of the LabVIEW tool. Future work needs to extend the modeling to other COTS development environments. This will allow for a more complete set of tools in the market giving the developer more choice when selecting the appropriate tool for the task at hand.

References

[1] E. A. Lee, "Absolutely positively on time: what would it take? [embedded computing systems]," *Computer*, vol. 38, no. 7, pp. 85–87, 2005.

[2] National Instruments, 11500 N. Mopac Expressway, Austin, TX. http://www.ni.com [Last accessed: Jan. 14, 2007].

[3] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[4] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer Science+Business Media Inc, second ed., 2004.

[5] J. A. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[6] Wikipedia, "Real-time operating system." Online. http://en.wikipedia.org/wiki/Real-time_operating_system. [Last accessed: Jan. 14, 2007].

[7] National Instruments, *LabVIEW User Manual (version 7.1)*, April 2003.

[8] National Instruments Support, Online. http://www.ni.com/support/ [Last accessed: Jan. 14, 2007].

[9] National Instruments, *LabVIEW Real Time Module User Manual (version 7.1)*, April 2004.

[10] Ardence, Inc., 266 2nd Avenue, Waltham, MA. http://www.ardence.com [Last accessed: Jan. 14, 2007].

[11] National Instruments, *LabVIEW Execution Trace Toolkit User Guide (version 7.1)*, April 2004.

[12] "Using the Timed Loop to write multirate applications in LabVIEW," Application Note 200, National Instruments, March 2004.

[13] National Research Council of Canada Institute for Aerospace Reseach, Building U-66 Research Road, Ottawa, ON.

[14] M. Ahmadi, D. Orchard, and F. C. Tang, "Captive trajectory simulation: On robotic performance effects," in *Mechatronics and Robotics Conference (Mechrob)*, (Aachen, Germany), pp. 802–807, September 2004.

[15] M. Ahmadi, M. Jaber, and F. C. Tang, "High-performance multi-body collision detection for the real-time control of a CTS system," *Transactions of the CSME, Special Edition*, vol. 29, no. 2, 2005.

[16] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 10, no. 1, pp. 46–61, 1973.

[17] J. W. S. Liu, *Real-Time Systems.* Prentice Hall, 2000.

[18] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 920–934, 1993.

[19] D. B. Stewart, "Measuring execution time and real-time performance," in *Embedded Systems Conference*, April 2001. Class 470.

[20] R. Obenza, "Rate monotonic analysis for real-time systems," *Computer*, vol. 26, no. 3, pp. 73–74, 1993.

[21] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, July 1993.

[22] M. H. Klein, J. P. Lehoczky, and R. Rajkumar, "Rate-monotonic analysis for real-time industrial computing," *Computer*, vol. 27, no. 1, pp. 24–33, 1994.

[23] K. A. Kettler, D. I. Katcher, and J. K. Stronsnider, "A modeling methodology for real-time/multimedia operating systems," in *Proceedings of Real-Time Technology and Applications Symposium*, pp. 15–26, May 1995.

[24] D. del Val and A. Viña, "Applying RMA to improve a high speed, real time data acquisition system," in *Proceedings of Real-Time Systems Symposium*, pp. 159–164, December 1994.

[25] A. Burns and A. J. Wellings, "Engineering a hard real-time system: From theory to practice," *Software - Practice and Experience*, vol. 25, no. 7, pp. 705–726, 1995.

[26] R. George and Y. Kanayama, "A rate-monotonic scheduler for the real-time control of autonomous robots," in *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, pp. 2804–2809, April 1996.

[27] D. B. Stewart and G. Arora, "A tool for analyzing and fine tuning the real-time properties of an embedded system," *IEEE Transactions On Software Engineering*, vol. 28, no. 4, pp. 311–326, 2003.

[28] J. Huang, J. P. M. Voeten, A. Ventevogel, and L. van Bokhoven, "Platform-independent design for embedded real-time systems," in *Languages for system specification: Selected contributions on UML, systemC, system Verilog, mixed-signal systems, and property specification from FDL'03*, Kluwer Academic Publishers, 2004.

[29] W. Winiecki and P. Bilski, "Time aspects of virtual instrument designing," in *Proceedings of IEEE Instrumentation Measurement Technology Conference (IMTC)*, vol. 2, pp. 913–918, May 2003.

[30] W. Winiecki, "The methodology of virtual instruments time analysis," in *IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems:Technology and Applications (IDAACS'03)*, pp. 432–436, September 2003.

[31] P. Bilski and W. Winiecki, "Time optimization of soft real-time virtual instrument design," *IEEE Transactions on Instrumentation and Measurement*, vol. 54, no. 4, pp. 1412–1416, 2005.

[32] M. Javer, T. W. Pearce, M. Gibeault, and M. Ahmadi, "Profiling of the LabVIEW development environment and real-time module," in *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*, May 2006.

Appendix A: Sample LRCM Calculation

This Appendix provides an example of how a predicted failure is determined using RMA and the LRCM. The example is based on the parameters used in test case 2 (Section 6.2.2). In Section A.1, the parameters used in test case 2 are summarized. Section A.2 shows how a predicted failure is determined using RMA. In Section A.3, the calculations used to determine a predicted failure by the LRCM is outlined. Finally, Section A.4 shows the calculation of % error between the predicted and measured failures.

A.1 Model Parameters

Table A-1 provides a summary of the parameters used in test case 2. This is similar to Table 6.3 except that the value of c_1 shown is the execution time of $task_1$ when a failure is predicted by the LRCM.

task	$c_i(\mu s)$	$c_{sched}(\mu s)$	$\gamma_{timing}(\mu s)$	$p_i(\mu s)$	$offset(\mu s)$	*priority*
1	41.07	7.55	1.79	100	0	100
2	30.26	3.58	1.79	100	0	50
ETS Timer	7.3275	3.58	0	1002	0	49
Unnamed Thread1	10.7061	3.58	0	2004	0	48
Unnamed Thread2	2.0255	3.58	0	2004	0	47

Table A-1: Summary of tasks used in test case 2.

A.2 RMA Prediction

The maximum processor utilization bound for RMA can be determined by making use of Equation 3.1. As only two user tasks are created in this test case, the maximum processor utilization before a set of tasks is predicted to fail by RMA becomes:

$$\begin{aligned} U(RMA) &= n(2^{1/n} - 1) \text{ , where } n = 2 \\ &= 2(2^{1/2} - 1) \\ &= 0.8284 \end{aligned}$$

After determining the processor utilization at which the RMA predicted failure occurs, the value for c_1 at which the predicted failure will occur can be found by solving for c_1 in the following processor utilization calculation:

$$
\begin{aligned}
U(RMA) &= \sum_{user\_tasks} \frac{c_i}{p_i} \\
&= \frac{c_1}{p_1} + \frac{c_2}{p_2} \\
c_1 &= \left(U(RMA) - \frac{c_2}{p_2} \right) * p_1 \\
&= \left(0.8284 - \frac{30.26}{100} \right) * 100 \\
&= 52.58\mu s
\end{aligned}
$$

A.3 LRCM Predition

The LRCM prediction is based on actual task execution times and periods of the application under consideration. The values in Table A-1 are the values of each of the tasks' execution times and periods used in the test case when a predicted failure is determined by the LRCM. In the test case, the execution time of $task_1$ is increased from a value of c_1 where the set of tasks is predicted and measured to pass. As c_1 is increased, a prediction using the LRCM is made for each value of c_1. Once a failure is predicted for a value of c_1, further calculations with increasing c_1 are not necessary as any additional loading of the processor will still result in a predicted failure.

As calculation of the predicted failure by the LRCM is repetitive for each value of c_1 tested, only one calculation is shown. This will be done using the value of c_1 for which the predicted failure occurs (i.e. $c_1 = 41.07\mu s$ as noted in Table A-1).

The determination of failure begins by calculating the LRCM processor utilization (Equation 6.1). This utilization calculation includes the user tasks as well as the LabVIEW created tasks and also includes the overheads for the scheduler and context

switch. An example of the calculation using the parameters in Table A-1 is shown below.

$$
\begin{aligned}
U_{LRCM} =& \sum_{i=1}^{n} \frac{c_i + 2c_{context} + \gamma_{timing}}{p_i} \\
=& \frac{c_1 + 2\Delta_{thr} + \gamma_{timing}}{p_1} + \frac{c_2 + 2\lambda_{thr1} + \gamma_{timing}}{p_2} \\
&+ \frac{c_{ETS} + 2\lambda_{thr1}}{p_{ETS}} + \frac{c_{Unnamed1} + 2\lambda_{thr1}}{p_{Unnamed1}} + \frac{c_{Unnamed2} + 2\lambda_{thr1}}{p_{Unnamed2}} \\
=& \frac{41.07 + 2(7.55) + 1.79}{100} + \frac{30.26 + 2(3.58) + 1.79}{100} \\
&+ \frac{7.32752(3.58)}{1002} + \frac{10.7061 + 2(3.58)}{2004} + \frac{2.0255 + 2(3.58)}{2004} \\
=& 0.99996
\end{aligned}
$$

Since the calculated LRCM processor utilization is less than 1, analysis can continue using the iterative technique described in Section 5.4.7 using Equation 6.2.

Beginning with $task_1$, the demand for processor time is calculated for $0 < t \leq p_1$.

$$
\begin{aligned}
w_1(t) =& (c_1 + 2c_{context} + \gamma_{timing}) + (n-1)c_{sched} \\
&+ \sum_{k=1}^{1-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context} + \gamma_{timing}) \\
=& (41.07 + 2(7.55) + 1.79) + 1(5.09) \\
=& 59.09\mu s \\
w_1(t = 60) =& 59.09\mu s
\end{aligned}
$$

The value of $t = 60\mu s$ satisfies the relationship that $w_1(t) < t$ for a value of t where $0 < t \leq p_1$. Thus, $task_1$ will be able to meet its deadlines.

Next, the same calculation is done for $task_2$ and all other higher priority tasks.

$$\begin{aligned} w_2(t) =& (c_2 + 2c_{context} + \gamma_{timing}) + (n-1)c_{sched} \\ &+ \sum_{k=1}^{2-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context} + \gamma_{timing}) \\ =& (30.26 + 2(3.58) + 1.79) + \left\lceil \frac{t}{p_1} \right\rceil (41.07 + 2(7.55) + 1.79) \\ w_2(t = 98) =& 97.20\mu s \end{aligned}$$

At $t = 98\mu s$, the demanded processor time is $w_2(98) = 97.20\mu s$. Thus, $task_2$ and all other higher priority tasks will meet their deadlines.

The calculation of demanded processor time for the ETS Timer task and all other higher priority tasks is shown below.

$$\begin{aligned} w_{ETS}(t) =& (c_{ETS} + 2c_{context}) + (n-1)c_{sched} \\ &+ \sum_{k=1}^{2-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context} + \gamma_{timing}) \\ =& (7.3275 + 2(3.58)) + \left\lceil \frac{t}{p_1} \right\rceil (41.07 + 2(7.55) + 1.79) \\ &+ \left\lceil \frac{t}{p_2} \right\rceil (30.26 + 2(3.58) + 1.79) \\ w_{ETS}(t = 598) =& 597.69\mu s \end{aligned}$$

The demand for processor time is met when $t = 598\mu s$ at which point $w_{ETS} = 597.69\mu s$. As the time t is still less than the period of the ETS Timer task of $1004\mu s$, the ETS Timer task and all other lower priority tasks will meet their deadlines.

Next, the calculation of the demanded processor time for Unnamed Thread1 and

all other higher priority tasks can take place.

$$w_{Unnamed1}(t) = (c_{Unnamed1} + 2c_{context}) + (n-1)c_{sched}$$

$$\begin{aligned} w_{Unnamed1}(t) =& (c_{Unnamed1} + 2c_{context}) + (n-1)c_{sched} \\ &+ \sum_{k=1}^{2-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context} + \gamma_{timing}) \\ =& (10.7061 + 2(3.58)) + \left\lceil \frac{t}{p_1} \right\rceil (41.07 + 2(7.55) + 1.79) \\ &+ \left\lceil \frac{t}{p_2} \right\rceil (30.26 + 2(3.58) + 1.79) + \left\lceil \frac{t}{p_{ETS}} \right\rceil (7.3275 + 2(3.58)) \end{aligned}$$

$$w_{Unnamed1}(t = 1700^{-}) = 1699.25\mu s$$

The demanded processor time is less than the available time when $t = 1700^{-}\mu s$ (i.e. just before $t = 1700\mu s$ so that the processor load is not increased by a newly arrived user-tasks at $1700\mu s$). This is within the period of the Unnamed Thread1; therefore all the tasks until now will still be able to meet their deadlines.

Finally, the calculation is performed for the Unnamed Thread2 and all other higher priority tasks.

$$
\begin{aligned}
w_{Unnamed2}(t) =& (c_{Unnamed2} + 2c_{context}) + (n-1)c_{sched} \\
&+ \sum_{k=1}^{2-1} \left\lceil \frac{t}{p_k} \right\rceil (c_k + 2c_{context} + \gamma_{timing}) \\
=& (2.0255 + 2(3.58)) + \left\lceil \frac{t}{p_1} \right\rceil (41.07 + 2(7.55) + 1.79) \\
&+ \left\lceil \frac{t}{p_2} \right\rceil (30.26 + 2(3.58) + 1.79) + \left\lceil \frac{t}{p_{ETS}} \right\rceil (7.3275 + 2(3.58)) \\
&+ \left\lceil \frac{t}{p_{Unnamed1}} \right\rceil (10.7061 + 2(3.58))
\end{aligned}
$$

$$
w_{Unnamed2}(t = 2004^{-}) = 2097.24\mu s
$$

At time $t = 2004^{-}\mu s$, the demand for processor time is still $w_{Unnamed1} = 2087.24\mu s$, which is greater than the available time. Thus, the criteria that the demanded processor time is less than or equal to the available processor time is not met before the period of the Unnamed Thread2 is reached. This means that the deadlines of the Unnamed Thread2 and all other higher priority tasks will not be met. Therefore, the set of tasks described in Table A-1 is predicted to fail in meeting their deadlines according to the LRCM.

In order to compare the LRCM to the measured failure as well as the RMA prediction, the processor utilization is calculated using Equation 6.3 as shown below. This allows for direct comparison between each of the measured and predicted failures.

$$
\begin{aligned}
U(LRCM) &= \sum_{user\_tasks} \frac{c_i}{p_i} \\
&= \frac{c_1}{p_1} + \frac{c_2}{p_2} \\
&= \frac{41.07}{100} + \frac{30.26}{100} \\
&= 0.7133
\end{aligned}
$$

A.4 % Error

In order to determine the error between the predicted and measured failures, the processor utilization at the failure point is calculated using Equation 6.3 as shown below. Since the measured failure occurs when $c_1 = 43.31\mu s$, this value is used in calculating the processor utilization at the failure point.

$$
\begin{aligned}
U(\text{meas failure}) &= \sum_{user\_tasks} \frac{c_i}{p_i} \\
&= \frac{c_1}{p_1} + \frac{c_2}{p_2} \\
&= \frac{43.31}{100} + \frac{30.26}{100} \\
&= 0.7357
\end{aligned}
$$

Using the processor utilization as the method of comparison, the % error between both the predicted failures (RMA and LRCM) and the measured failure can be determined using Equation 6.4 as shown below.

$$\begin{aligned}
\%error(RMA) &= \left|\frac{U(\text{meas failure}) - U(RMA)}{U(\text{meas failure})}\right| * 100 \\
&= \left|\frac{0.7357 - 0.8284}{0.7357}\right| * 100 \\
&= 12.60\%
\end{aligned}$$

$$\begin{aligned}
\%error(LRCM) &= \left|\frac{U(\text{meas failure}) - U(LRCM)}{U(\text{meas failure})}\right| * 100 \\
&= \left|\frac{0.7357 - 0.7133}{0.7357}\right| * 100 \\
&= 3.04\%
\end{aligned}$$