ABSTRACT

REAL-TIME OPERATING SYSTEM SERVICES FOR NETWORKED EMBEDDED SYSTEMS

by
Khawar M. Zuberi

Chair: Kang G. Shin

This dissertation examines issues related to the design and implementation of real-time operating system (RTOS) services for consumer item embedded systems such as cars, webTVs, and cellular phones. Since these devices are mass-produced, keeping production costs low is all important, meaning that RTOS services must be highly efficient to enable use of low-cost hardware in these devices. This dissertation explores optimization techniques for basic RTOS services (task scheduling, synchronization, and communication) to meet the requirements of embedded systems.

First, dynamic schedulers like *earliest-deadline first* (EDF) give high processor utilization but incur high run-time overhead while the opposite is true for static schedulers like *rate-monotonic* (RM). We design the *combined static/dynamic* (CSD) scheduler which employs both EDF and RM to deliver run-time overhead comparable to RM and utilization comparable to EDF.

Second, increasing use of OO programming has made efficient synchronization important for embedded systems. In an OO multitasking environment, accesses to objects have to be protected through semaphores to ensure mutual exclusion. We present a new semaphore implementation scheme which saves one context switch per lock operation in most circumstances by rescheduling threads to minimize blocking.

Third, embedded devices with Internet connectivity require efficient processing of network data. We present a protocol architecture for reducing receive overhead for audio and video messages. The architecture reduces I-cache misses by safely bypassing multiple protocol layers (benefiting short audio messages) and real-time message data is copied only once

without any hardware support (benefiting long video messages).

Finally, embedded systems incoporating local area networks (LANs) require proper LAN message scheduling. Controller Area Network (CAN) is a popular LAN in automotive applications. Pure EDF scheduling is not feasible for CAN because of unacceptable overhead while deadline-monotonic (DM) scheduling is feasible but yields low network utilization. We present the feasible *mixed traffic scheduler* (MTS) which combines EDF and DM using quantized deadlines to deliver higher utilization than DM.

Using these optimizations, we built and evaluated an RTOS that provides a rich set of services to programmers and meets the efficiency/cost requirements of embedded systems.

REAL-TIME OPERATING SYSTEM SERVICES FOR NETWORKED EMBEDDED SYSTEMS

by

Khawar M. Zuberi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1998

Doctoral Committee:

Professor Kang G. Shin, Chair
Assistant Professor Peter Chen
Associate Professor Farnam Jahanian
Professor Galip Ulsoy

To my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Real-time computing [103, 108] deals with predictable and timely execution of tasks to meet application-specific timing constraints. In the past, real-time computing focused almost exclusively on large and expensive projects related to avionics, space, and defense applications. Considerable research efforts were devoted to developing large-scale multiprocessor/distributed systems for air traffic control, radar tracking systems, and planetary exploration robots. In such applications, actual hardware costs tend to be much less than development costs, so real-time software algorithms developed for such systems focus primarily on providing correct functionality and treat efficiency as only a secondary concern at best. If performance needs to be improved, it is usually done by using faster processors and networks (since hardware cost is not much of an issue).

The sharp drop in microprocessor prices over the past several years has resulted in digital control being used in much smaller and simpler embedded applications [30] such as automotive controllers, cellular phones, and home electronic appliances. Also, the increasing popularity of the Internet has led to a new class of computing/communication devices called *information appliances* (IAs) [71]. IAs are specialized devices with Internet connectivity. Examples include web televisions (webTVs), personal digital assistants (PDAs), web video phones, and digital cellular phones with e-mail and web browsing capabilities. All of these devices (cars, IAs, and home appliances) are consumer products. They are mass-produced in volumes of tens of millions of units, so that unlike the large-scale applications mentioned previously, manufacturing costs (including hardware costs) tend to be much more than development costs. Keeping production costs to a minimum is paramount in such applications, meaning that real-time software must be very efficient in its use of central processing unit (CPU) and network resources.

Such consumer applications are now widespread and important enough to make efficiency the primary concern in the design of algorithms for real-time system-level services. Building upon the schemes developed for large-scale applications, new schemes need to be developed for real-time task scheduling, synchronization, and network communication. Like the schemes for large-scale systems, these new schemes must provide correct functionality, but beyond that, they must also satisfy the efficiency requirements of emerging real-time embedded applications.

1.1 Embedded Systems

All digital systems contained in a larger environment and controlling that environment in some way can be called *embedded systems* [30, 35]. In most of these systems, the environment being controlled imposes response time restrictions on the embedded system in which case it is referred to as a *real-time* embedded system [30, 35]. The on-board navigation system of an aircraft is a real-time embedded system as is the engine controller of an automobile and the digital signal processing (DSP) controller in a cellular phone.

We are interested in *consumer item embedded systems*. These are embedded systems used in consumer products such as cars, home electronic appliances, and IAs. Just a few years ago, embedded systems used in consumer products were simple microcontrollers running a few tasks written in assembly or C. But the embedded systems of today (Figure 1.1) tend to be networked, run application code written in object-oriented (OO) languages such as Java, execute an increasing number of complex tasks, and need real-time operating system (RTOS) [95] support — either to handle audio/video or to interact with the environment. The challenge is to provide a functionally-rich operating system (OS) able to support these new applications while keeping OS overheads to a minimum. An efficient OS enables low-cost hardware to be used in consumer products and this lowers per-unit costs. Moreover, slow hardware uses less electric power which prolongs battery life in mobile devices like cell phones and PDAs. Both these factors (low cost and longer battery life) make the product attractive for customers.

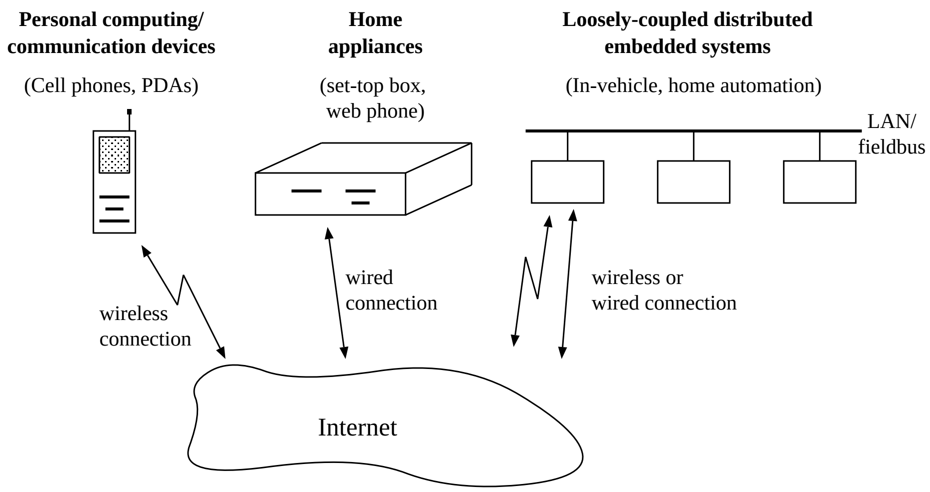

Figure 1.1: Examples of consumer item embedded systems.

1.2 Research Objectives

Existing solutions for basic problems in real-time computing (predictable and timely scheduling, synchronization, and communication) — which were developed with large-scale applications in mind and may have acceptable performance for such systems — are not efficient enough to be feasible for cost-conscious consumer products (as described later in this section). This results in embedded systems programmers often attempting to improve performance through hand-crafted — thus ad hoc — techniques which not only increases design-time but also tends to be error-prone.

Researchers in the past have focused on large-scale applications [3, 17, 51, 72] and with good reason: smaller applications used either mechanical or analog electronic control (as in automobiles) or, if using digital control, had a simplistic design where the controller performed only very basic functions (as in various home appliances). These consumer item embedded systems were simple enough that hand-crafted software was feasible. All hardware was controlled directly by the application code; in fact, there was no distinction between OS and application software (Figure 1.2-a). This resulted in efficient use of available hardware resources but at the cost of complete non-portability of software (which was acceptable in these simple systems). Today, the complexity of embedded systems is increasing rapidly. Cellular phone controllers — besides running traditional DSP algorithms — must also handle e-mail, web browsing, and security/encoding features. Automotive cruise controllers do more than maintain a constant speed; they also provide collision avoidance features. Another factor contributing to this rapidly increasing complexity is networking. More and more embedded devices are connected to the Internet (such as web TVs and cellular phones). Also, applications which have multiple controllers (such as automobiles) now have these controllers interconnected by a *field bus* [10, 91] (local area networks (LANs) specially designed for real-time control applications) to allow the controllers to coordinate their activities, thus providing new features and better performance.

The increased complexity of today's embedded systems[1] necessitates that an RTOS be used to manage various resources (Figure 1.2-b). The OS services needed in these embedded systems are essentially the same as in any real-time system: the multiple application tasks must be properly scheduled to meet their deadlines, access to various resources (such as critical sections) must be managed through synchronization primitives to ensure mutual exclusion, and network communication must be handled predictably and in a timely manner. But what is unique about embedded systems used in consumer products is that these systems are mass produced. This makes low per-unit costs one of the primary concerns in the design of these systems. Automotive applications alone account for tens of millions of embedded systems produced every year and the same is true for the simple (without Internet connection) cellular phones in use today. Annual production volumes of IAs (including cellular phones with Internet connectivity) are expected to reach 48 million units by year 2001 [5]. At these volumes, extra costs of even a few dollars per unit translate into a loss of millions of dollars overall. So, the microprocessors used in these cost-conscious applications are those which have been in production for several years and their prices have dropped to a few dollars per chip. IAs communicate through wireless networks or over phone lines, so the network bandwidth available to them is much less than that available to desktop workstations. Similarly, the field bus networks are of low bandwidth, usually 1–3 Mbits/s [94]: on one hand this keeps costs down and on the other, this is all the bandwidth that is really needed to exchange short messages of sensor readings and actuator commands. This gives embedded systems a very different "flavor" than larger applications with faster processors and networks. As a result, solutions appropriate for large systems have too high an overhead for embedded systems. For example, a multitude of RTOSs have been designed to date which provide a predictable platform for task execution and inter-process

[1] From here on, we use the term *embedded system* to mean consumer item embedded system, unless stated otherwise.

communication, but most of these RTOSs like Alpha [51] and the Spring Kernel [109] were designed for large parallel and distributed systems with powerful processors and fast interconnection networks. But in the context of smaller embedded systems, these RTOSs are too large to fit in on-board ROM (since most of these systems do not have disks, ROM is used for non-volatile storage), their communication protocol architectures are too general (and hence, too slow), and their run-time overhead is too high for low-cost processors used in embedded systems. Attempts to reduce the RTOS overheads have till now been in the direction of taking a large RTOS and removing useful features from it like threads, preemptive scheduling, and memory protection, which tends to make an RTOS difficult to use. As a result, embedded system designers tend to shy away from off-the shelf RTOSs, opting for customized solutions which are inflexible, and costly to design and port.

In this thesis, we look at the basic problems of real-time computing (task scheduling, synchronization, and network communication) from the perspective of embedded systems and either adapt existing solutions or propose new ones which would be low enough in cost and overhead to be feasible for embedded systems. In our solutions to these problems, we do not place any restrictions on the application programmer (which makes application software design easier), nor do we rely on existence of special hardware support (since adding special hardware features increases costs, thus being contrary to our prime objective). Instead, our research focused on identifying characteristics peculiar to real-time systems in general or embedded systems in particular and exploiting these properties to arrive at efficient solutions for OS services needed in embedded systems.

1.2.1 Implementation Platform

We evaluate the performance of our various OS optimizations by implementing them within the EMERALDS (Extensible Microkernel for Embedded, ReAL-time, Distributed Systems) RTOS. EMERALDS is a small kernel we designed for cost-conscious embedded devices. It provides multitasking, efficient context switching, interrupt handling, and other basic OS primitives (details are in Chapter 3).

EMERALDS runs on the Motorola 68040 processor which is typical of the low-cost CPUs used in embedded systems. Our proposed schemes for task scheduling, synchronization, and network communication (both field bus communication as well as UDP/IP communication over the Internet) were all implemented within EMERALDS and performance measured on the 68040.

EMERALDS was also ported to the Motorola 68332 and PowerPC 505 microcontrollers in a joint project with Ford Scientific Research Lab. Ford evaluated performance of EMERALDS on these microcontrollers and compared it to several commercial small RTOSs. Results of these evaluations showed EMERALDS to have superior performance (see Chapter 3). Moreover, EMERALDS was also modified to conform to the OSEK automotive OS standard [89] and ported to the Hitachi SH-2 processor in a joint project with Hitachi Research Lab.

1.3 Outline of the Dissertation

The remainder of this dissertation is organized as follows.

Chapter 2 discusses various OS services needed by embedded systems, and motivates the need to optimize these services for embedded systems. It also gives an overview of our proposed approaches for optimizing these services and our primary contributions in achieving this goal.

Chapter 3 describes the base EMERALDS kernel and covers some of the unique features of EMERALDS including memory protection scheme (where the kernel is mapped into each address space) and low-cost local message passing using state messages.

Chapter 4 deals with task scheduling. It outlines the shortcomings of existing schedulers and describes how these may be overcome by combining the best features of existing schedulers to get combined static/dynamic scheduling.

Chapter 5 describes our low-overhead priority inheritance semaphore implementation scheme. We show that the scheme is broadly applicable to embedded systems and does not put any limitations on the application programmer (unlike some other semaphore optimization schemes).

Chapter 6 focuses on audio/video communication requirements of IAs. We present a communication protocol architecture which reduces both I-cache misses as well as data copying overheads (without relying on any special network or hardware features), thus improving performance for both short audio and long video messages.

Chapter 7 discusses scheduling messages on the *Controller Area Network* (CAN) which is a popular field bus used in automotive and factory automation applications. The short packet size of CAN makes certain scheduling schemes infeasible while other schemes lead to low network utilization. We develop a hybrid scheme which is feasible for CAN and delivers high utilization.

Finally, Chapter 8 concludes this dissertation by summarizing its primary contributions and suggesting directions for future research.

∞

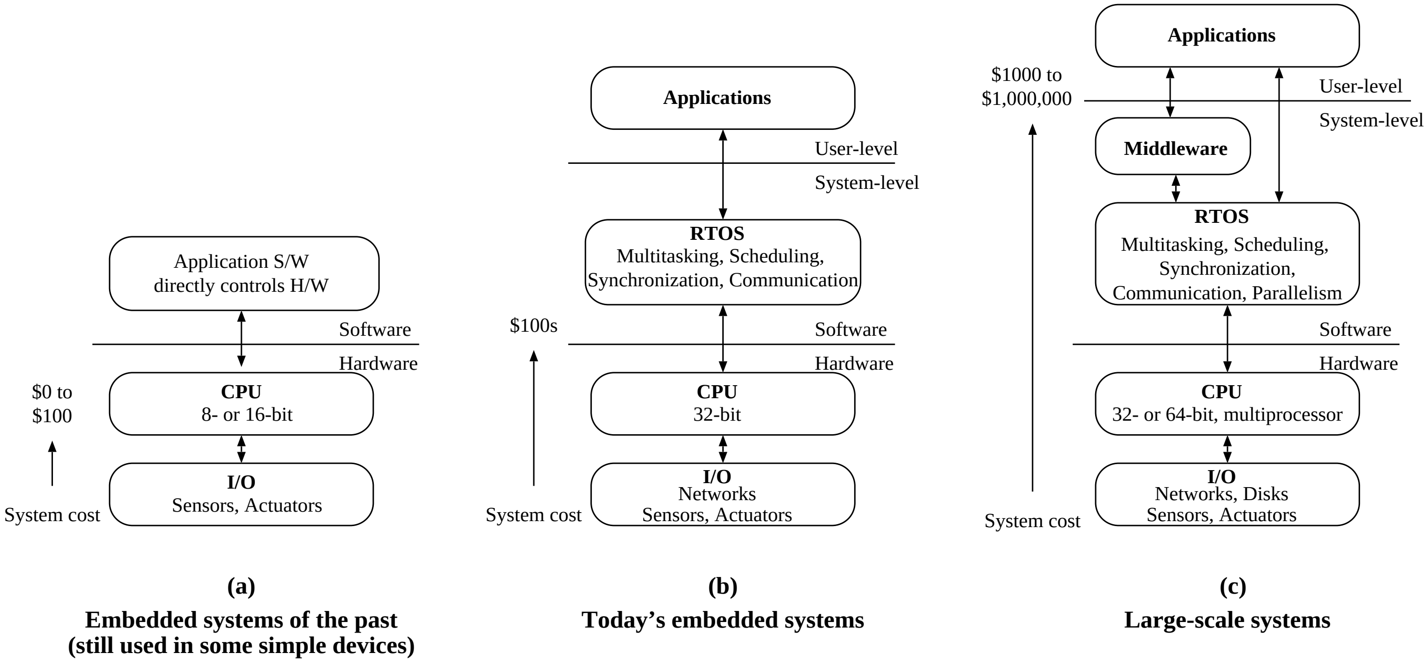

Figure 1.2: Software, hardware, and cost differences between consumer item embedded systems of the past, those of today, and large-scale systems.

CHAPTER 2

OS SERVICES NEEDED BY EMBEDDED SYSTEMS

System-level services can be classified as either OS or middleware services. The OS encompasses services which are widely used by almost all applications (thread/process management, memory management, communication primitives, etc.). Middleware services are implemented on top of the OS (e.g., in the form of daemon processes) and they provide services needed by specialized applications usually running on large-scale parallel or distributed platforms. Examples of middleware services include reliable/atomic multicast [1, 37], consistent event ordering [12, 59, 123], and task allocation in a parallel/distributed environment [107]. At present, embedded systems used in consumer items are not complex enough to require middleware services. Applications such as cellular phones and home electronics do not have a distributed architecture, so they simply do not need middleware services.[1] Automotive controllers are interconnected by a LAN, but the various subsystems in a car are only loosely-coupled so that simple message-passing is enough to satisfy application requirements. As such, we will not discuss middleware services any further and instead, focus on OS services needed in embedded systems.

2.1 Real-Time OS Services

The primary purpose of an RTOS is to provide a predictable platform for execution of application tasks. In embedded systems, there is an added requirement that OS services must be highly efficient. These efficiency and predictability requirements shape the design

[1] One exception is the Java virtual machine. It supports application portability rather than execution in a distributed environment, so it can be thought of as a specialized application rather than a middleware service.

of various OS services as discussed next.

2.1.1 Scheduling

In a real-time system, a task scheduler must be used to multiplex the CPU between the various tasks in a manner which ensures that task deadlines are met. Real-time schedulers can be classified into two broad categories: *time-slice cyclic scheduler* and *priority-based schedulers*. In time-slice scheduling, the entire execution schedule is constructed either off-line or at task admission time, and at run-time the scheduler simply assigns the CPU to tasks according to these schedules. This is a simple and efficient scheme provided that tasks have harmonic periods of execution. But cyclic scheduling usually does not work well for aperiodic tasks, tasks with mutually prime periods, and workloads with both short and long periods (as discussed in detail in Chapter 4). Note that all three of these conditions occur commonly in modern-day embedded systems.

Priority-based schedulers use some policy to prioritize tasks, then at run-time, the scheduler ensures that the CPU is always assigned to the highest-priority active task. This requires maintaining a sorted queue of tasks which results in more run-time overhead than cyclic schedulers. However, priority-based schedulers are more capable of handling aperiodic tasks and do not impose any restrictions on task periods.

Priority-based schedulers are further classified as *static* or *dynamic*. Static schedulers assign a fixed priority to tasks. The *rate-monotonic* (RM) scheduler is the best-known static scheduler under which tasks with shorter periods are assigned higher priorities. Under dynamic scheduling, each different invocation of a task can have a different priority. For example, the *earliest-deadline first* (EDF) scheme schedules tasks according to the absolute deadlines of individual task invocations: earlier the deadline, higher the priority.

On one hand, dynamic schedulers are able to deliver better CPU utilization than static schedulers, but on the other hand, they incur higher run-time overhead because they have to repeatedly re-sort tasks according to their changing deadlines. As such, for practical purposes, the two categories of priority-based schedulers often deliver the same low overall performance. As much as 10% of CPU cycles can be lost because of the scheduler. Till now, real-time system designers have accepted these lost cycles as the price to pay for ensuring predictability. However, such high scheduler overheads are not acceptable in cost-conscious embedded systems, which motivates development of new, low-overhead scheduling schemes.

2.1.2 Synchronized Access to Shared Resources

When tasks access a shared resource such as a critical section or I/O device, the tasks must synchronize with each other to ensure mutual exclusion. Semaphores are commonly used for this purpose. One common use of semaphores is in object-oriented (OO) programming. Updates to the state variables of objects have to be protected through semaphores to ensure mutual exclusion, and this represents significant run-time overhead. The advent of Java has made OO programming important for embedded systems since Java provides networked embedded systems with the capability to download and execute code on-demand. This underscores the importance of providing efficient, low-overhead semaphores in embedded systems.

2.1.3 Communication

An increasing number of embedded systems today tend to be networked. Some systems (such as cellular phones and web TVs) may be connected to the Internet whereas in other systems (such as an automobile), multiple controllers within the system may be interconnected by a field bus. All such systems require OS support for real-time network communication. Moreover, some applications exhibit heavy message-passing between tasks running on the same processor. An RTOS for embedded systems must provide efficient support for all these different forms of communication as discussed next.

Network Communication

Providing end-to-end real-time communication guarantees requires:

- Scheduling messages on the network (a wide area network (WAN) and/or a LAN/field bus) to ensure timely delivery, and
- Structuring end-host message processing to ensure predictable and timely delivery of messages to applications.

Supporting real-time traffic over WANs (such as the Internet) requires that routers be capable of handling prioritized traffic. Since WAN scheduling deals mostly with router issues, it does not affect the design of OS services for embedded systems and is beyond the scope of this thesis. Interested readers are referred to [24, 119].

Since a field bus is contained entirely within an embedded system, scheduling messages on a field bus *is* the responsibility of the embedded systems designer. Moreover, each message transmitted or received over a network (be it a LAN or a WAN) needs to have some processing done on it by the OS (such as attaching/stripping headers and determining destination). Just as application task processing must be scheduled properly in a real-time system, message processing also must be done in a predictable manner. These issues are discussed next.

Field Bus Scheduling: The recent proliferation of embedded systems has resulted in many network protocols being designed specifically to satisfy the real-time control requirements of embedded systems. These protocols include the Controller Area Network (CAN) [47], Profibus [32], FIP [28], SP-50 [44], MAP/TOP [80], SERCOS [46], and TTP [61]. All these protocols are known by the general name of *field bus* because they are meant to control the so-called field devices (sensors and actuators) and all of them have a bus topology. Buses are preferable to rings, stars, or other point-to-point topologies because they require the least amount of wiring [68] which keeps production costs down.

Scheduling messages on a field bus is made difficult by the fact that these messages are usually just 5–10 bytes long (which is all that's needed to exchange sensor readings and actuator commands). This is in sharp contrast to large-scale LANs such as FDDI, Ethernet, and ATM where the minimum message size is 52 bytes or more. This influences the design of various scheduling schemes which append headers to each message. In larger WANs, headers of even several bytes are acceptable, but in embedded systems where the entire message is just 5–10 bytes of data, the headers must be kept down to a few bits only.

End-Host Protocol Processing: Because of their web-centric nature, an efficient communication subsystem is an important part of an IA OS. The communication subsystem must efficiently and predictably handle both audio as well as video communication.

Predictability is needed to ensure timely processing of incoming and outgoing messages [82]. The protocol architecture must provide mechanisms to guarantee that high-priority real-time messages (such as live voice) do not get unnecessarily delayed by the processing of lower-priority or non-real-time messages. At the same time, the protocol architecture must provide for efficient execution of protocol code. The higher the protocol processing overhead, less the effective bandwidth delivered to applications. The architecture must be able to efficiently handle not only long messages (such as video) but also short, frequent ones (such as live audio). Also, studies have shown that receive-side protocol processing overhead is higher than send-side overhead [55, 57, 79] and this is what limits throughput; so, we focus on improving receive-side overhead while ensuring predictability.

Local Message-Passing

The traditional mechanism for exchange of information between tasks is message-passing using mailboxes. Under this scheme, one task prepares a message, then invokes a system call to send that message to a mailbox, from which the message can be retrieved by the destination task. While this scheme is suitable for certain purposes, it has two major disadvantages:

- Passing one message may take 50–100μs on a processor such as the Motorola 68040. Since tasks in embedded applications such as automotive usually need to exchange several thousand messages per second (related to engine RPM), this overhead is unacceptable.
- If a task needs to multicast the same message to multiple tasks, it must send a separate message to each.

Because of these disadvantages, application designers are typically forced to use global variables to exchange information between tasks. This is an unsound software design practice because reading and writing these variables is not regulated in any way which can introduce subtle, hard-to-trace bugs in the software. This motivates investigation of new mechanisms for intertask communication.

2.1.4 Miscellaneous Services

Besides task scheduling, synchronization, and communication support, the RTOS must also provide several standard basic primitives such as address spaces, threads, efficient context switching, and ability to interact with the environment through interrupts. Unlike OS services mentioned previously, the overheads of context switching and interrupt handling are dictated primarily by the hardware structure (such as number of registers and processor's interrupt handling mechanism). But still, these services need to be optimized by fine-tuning

OS code.

2.2 A Remark on Extensible OS Architectures

A recent hot topic of discussion in the OS community is the issue of extensibility. This deals with the OS providing mechanisms which allow the OS to be extended to provide new functionality. Extensibility falls into three broad categories: the microkernel approach [38,75], the grafting approach [7,100], and library OSs based on a thin kernel layer which securely exports hardware resources [22,53]. The microkernel approach envisions a small kernel implementing address spaces, interprocess communication, and other minimal core OS functionality while all other services (such as network communication and file systems) are provided by privileged user-level servers. The flexibility arises from the ability to change/modify servers without modifying the core kernel. The grafting approach allows for "direct" extension of the kernel by allowing code fragments (written in some safe language) to be added to the kernel. The kernel executes these fragments in a sandbox or other such safety mechanism to protect against malicious behavior. Finally, the library OS approach is similar to the microkernel approach in that it relies on a minimal kernel, but it differs from microkernels in that the kernel is even more stripped-down than in the case of microkernels. In fact, the kernel only provides an abstraction layer on top of actual hardware and all real functionality is implemented by library routines.

Terms such as "small" and "efficient" are commonly seen in literature related to extensibility. However, the reader should not confuse extensibility issues with the issues explored in this thesis. Extensible OSs lead to efficiency in the sense that application-specific optimizations can be easily incorporated in the OS. However, extensible OSs in no way suggest what those optimizations should be. Our work deals with devising OS optimizations for embedded systems. As such, our work is orthogonal to OS extensibility issues.

2.3 Proposed Approaches and Primary Contributions

In the previous subsections, we identified task scheduling, synchronization, and communication as key OS services which must be optimized to achieve good OS performance in embedded systems. Listed below are brief overviews of the approaches we took to optimize these services and our primary contributions.

• ***Task scheduling:*** The earliest-deadline-first (EDF) [76] scheduling scheme ideally schedules workloads with utilizations of up to 100%. But this is a theoretical limit which is not achieved in practice because of the high run-time overhead incurred by EDF in sorting tasks according to deadlines. On the other hand, the rate-monotonic (RM) [76] scheme has a much lower run-time overhead since it does not have to repeatedly re-sort the tasks, but on average, it delivers a schedulable utilization of only 88% [67]. For a practitioner, the sum of these two overheads (the run-time overhead plus schedulable utilization being less than 100%), which we call the *total scheduling overhead*, is a "true" measure of the performance of a scheduler because it indicates how many CPU cycles will actually be available for execution of application tasks.

We developed a mechanism to partition tasks in a given workload into two groups, and we demonstrated that when one group is scheduled by EDF and the other by RM, the resulting total scheduling overhead is less than that for either EDF or RM alone. This scheme — which we call *combined static/dynamic* (CSD) scheduling — lowers run-time overhead by using multiple scheduling queues, but at the same time, delivers high schedulable utilization by properly assigning tasks to the different queues. A proper partitioning of tasks is critical to the good performance of CSD. We present an iterative method to partition the tasks in a given workload into two groups. When tasks in one group are scheduled by EDF and tasks in the other group are scheduled by RM, the *total scheduling overhead* (run-time overhead plus schedulable utilization being less than 100%) is less than that of EDF or RM alone. By reducing this total overhead, CSD outperforms both RM and EDF in real systems.

• ***Efficient semaphores:*** Object-oriented programming can be feasible in multi-threaded embedded systems only if the OS provides efficient, low-overhead semaphores. We present a new semaphore implementation scheme which saves one context switch per semaphore lock operation in most circumstances. Note that an efficient semaphore scheme is useful not only for OO programming but for any application requiring synchronization between multiple threads of execution.

Previous work in improving semaphore performance has focused on either relaxing the semaphore semantics to get better performance [111], coming up with new semantics and new synchronization policies [114], or putting restrictions on the application programmer to disallow certain actions (such as making blocking system calls) while holding a semaphore [89]. The problem with these approaches is that these new/modified semantics may be suitable for some particular applications but usually they do not have wide applicability. We took the approach of providing full semaphore semantics (with priority inheritance [101]), but optimizing the implementation of these semaphores by exploiting certain features of embedded applications [126]. We rely on the fact that the order in which embedded applications access objects (which is the same as the order in which semaphores are used) can be determined at compile-time. This is true because of the sensor-controller-actuator loop executed by typical embedded applications. The compiler provides hints which enable the OS to schedule threads for execution only when the semaphores they need are available. Semaphore lock system calls succeed without blocking, resulting in reduced context switching and improved performance.

• ***Protocol architecture for audio/video:*** Real-time audio and video communication over the Internet is an integral part of many IAs which means that despite slow hardware, the communication subsystem within the OS must be able to efficiently handle heavy network traffic. The subsystem must be structured to handle both short as well as long messages with minimal overhead. Handling short messages efficiently is important for applications such as Internet telephony where live voice packets are usually just 30–50 bytes (as in the GSM audio encoding scheme [99] used in various Internet phones). On the other hand, video applications exchange long messages (10–15 kbytes [25]) and these must be handled efficiently as well.

We devised optimizations for reducing receive-side network protocol processing overhead thus enabling efficient handling of real-time audio and video messages [128]. (We focus on receive-side overhead since it usually exceeds send-side overhead.) In our scheme, I-cache miss overheads are minimized by safely bypassing multiple protocol layers, benefiting *short* messages such as live audio. Moreover, message data needs to be copied only once (without any hardware support from the network adapter or any restrictions on the network API) which benefits *long* messages such as video and streaming data.

• ***CAN scheduling and host support:*** The Controller Area Network (CAN) is being widely used in real-time control applications such as automobiles, aircraft, and automated factories [118]. CAN features priority-based bus arbitration, so scheduling real-time messages on CAN amounts to properly assigning priorities to messages to ensure transmission by their deadlines.

Pure earliest-deadline first (EDF) scheduling of messages is not useful for CAN: packets have only 8 bytes of payload so using a 20+ bit deadline as the priority (and including it with each packet) results in unacceptable network overhead. Fixed-priority deadline-monotonic (DM) scheduling needs fewer bits to express priorities but yields relatively low utilization. We designed a scheduler called the *mixed traffic scheduler* (MTS) which combines EDF and DM by using quantized deadlines [122, 125, 127]. Packets are scheduled based on deadlines if deadlines are distinguishable after quantization; otherwise, they are scheduled using DM priorities. Not only is MTS feasible for CAN (as demonstrated by its implementation within EMERALDS), it also delivers higher network utilization than DM. □

The following chapters discuss these scheduling, synchronization, and communication issues in detail.

CHAPTER 3

EMERALDS: A REAL-TIME OPERATING SYSTEM

EMERALDS is a small, fast kernel we have designed for use in embedded devices [124]. It features efficient context switches, interrupt handling, and memory usage. It provides full memory protection between processes, features an efficient system call mechanism, and has a low-overhead intra-node inter-process communication (IPC) scheme. We used EMERALDS as a platform for implementing and evaluating the various optimizations we developed for scheduling, semaphores, and communication.

The main goal in designing EMERALDS was to see which features of embedded systems can we use to reduce size and overhead. Embedded systems provide many opportunities for simplification. Processes tend to exchange short, simple messages like sensor readings and actuator commands. A file system is usually not needed: all executable code is in ROM and all dynamic memory requirements are satisfied by RAM. These characteristics allow us to reduce the system call overhead, simplify IPC, and keep EMERALDS' size down to a minimum.

An important question related to reducing size was which OS services to include in EMERALDS and which to leave out. Many RTOSs leave out common OS features like memory protection and threads in an attempt to reduce size and increase speed. We did not take this approach. Instead, we provide all common OS services but use novel mechanisms for optimizing these services. This enabled us to meet our goals of efficiency and small size without scaling back on functionality.

In the next section we give a brief overview of EMERALDS. Sections 3.2–3.5 give the details of EMERALDS, covering processes, threads, memory protection, IPC, etc. Section 3.6 gives some timing measurements and Section 3.7 states conclusions and future work.

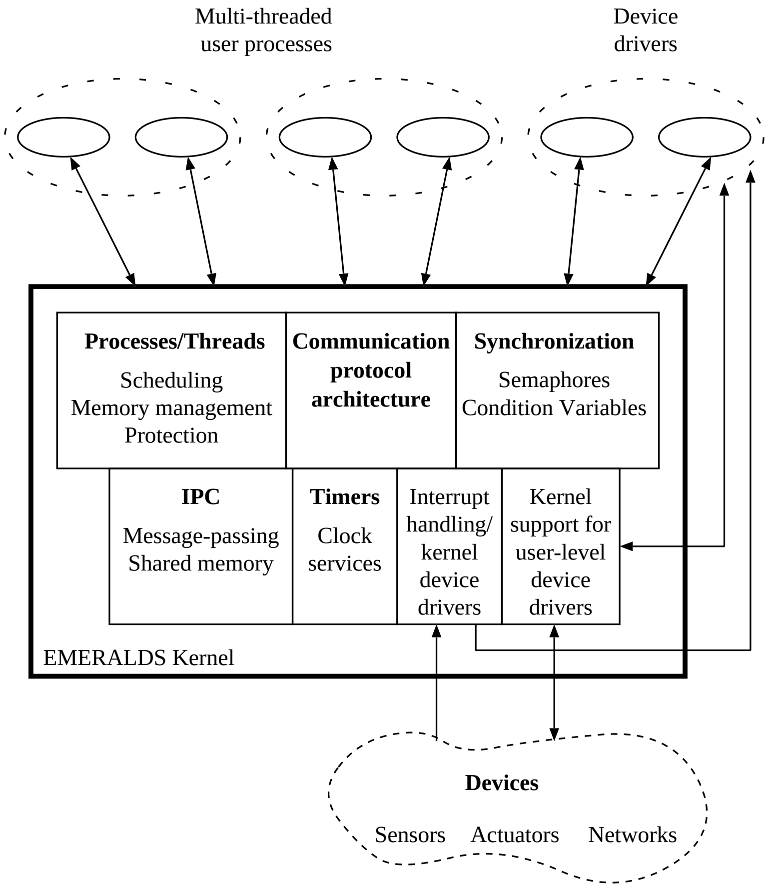

Figure 3.1: EMERALDS' architecture.

3.1 Architectural Overview

EMERALDS is a real-time operating system written in the C++ language. Following are EMERALDS' salient features as shown in Figure 3.1.

- Multi-threaded processes:
  - Full memory protection between processes.
  - Threads are scheduled by the kernel.
- IPC based on message-passing and mailboxes. Shared memory is also provided.
  - Optimized local message passing
- Semaphores and condition variables for synchronization; priority inheritance for semaphores.
- Support for communication protocol stacks.

- Highly optimized context switching and interrupt handling.
- Support for user-level device drivers.

Note that EMERALDS does not include a file system since our target applications are in-memory: ROM is used as non-volatile storage and on-board RAM satisfies all run-time memory requirements of the application. Leaving out the file system significantly reduces the size of the OS. Also, in most embedded control systems (such as automotive), the different nodes exchange only short sensor readings and actuator commands over a field bus. Threads can exchange such simple messages by talking directly to the field bus device driver without using any protocol stack, so EMERALDS does not have a built-in communication protocol stack. However, IAs do need a protocol stack for Internet communication, so EMERALDS incorporates a protocol architecture which can be used to extend EMERALDS to include a stack if needed (see Chapter 6).

The rest of this chapter discusses various system calls provided by EMERALDS for process/thread management, synchronization, and communication; as well as simple optimizations we developed for memory protection, efficient system calls, and local message passing. EMERALDS also uses novel techniques for task scheduling, semaphores, and inter-node communication, and these techniques are described in detail in later chapters.

3.2 Processes and Threads

EMERALDS provides multi-threaded processes. A process in EMERALDS is a passive entity, representing a protected address space in which threads execute. Each thread has a user-specified priority and is preemptively scheduled by the kernel based on this priority. Table 3.1 lists the EMERALDS system calls related to processes and threads.

System call	*Important Parameters*	*Function*
`create_proc()`	Thread priority	Create process with 1 thread
`create_thread()`	Thread priority	Create thread
`join_thread()`	Thread ID	Wait for child thread to finish
`detach_thread()`	Thread ID	Tell kernel: will not wait for thread

Table 3.1: Process and thread system calls.

The two most important features of EMERALDS processes and threads are memory protection and real-time scheduling. Scheduling is covered in detail in Chapter 4. Here we discuss EMERALDS' memory protection scheme.

Memory Protection

The need for memory protection in time-shared systems is indisputable. One user's processes must be protected from all other — possibly malicious — users. In single-user embedded systems, memory protection is useful for slightly different reasons. In relatively isolated embedded systems such as automotive controllers, memory protection provides software fault isolation. Bugs in application code can manifest themselves as malicious faults, which, without memory protection, can corrupt the memory of other processes or even the kernel. With memory protection, a memory access outside of the process' address space will cause a TRAP to the kernel and recovery action may be taken.

Memory protection is even more important in relatively open embedded systems such as IAs. Downloaded Java code may be intentionally malicious, making memory protection a must in such systems.

All these benefits of memory protection will not be of much practical use if the implementation of memory protection was not efficient and small-sized. To meet these goals, we made full use of the fact that our target applications are in-memory. This enabled us to reduce the total size of a page table to a few kbytes compared to several megabytes for virtual memory systems with disk backing stores. In the latter, the entire page table must exist, even if most of the address space is unused. This is needed to distinguish unmapped pages from those which have been swapped out to disk.[1] But for in-memory systems, this distinction is not needed. This allows the page table to be trimmed down using the hierarchical nature of most page tables. For example, the Motorola 68040 has three-level page tables. Each third level page table represents 256 kbytes of address space. So, if a process has three segments — code, data, and stack — and each is less than 256 kbytes, then its page table will be as shown in Figure 3.2.

All but three entries in the first-level page table are null, so only three second-level page

[1] Some OSs such as Linux use segment registers present in x86 processors to distinguish unmapped and swapped out pages. The VAX-11 provided a page table length register to achieve the same goal [70]. However, such hardware support is not available on many popular processors used in embedded systems such as 680x0, so an alternate scheme is needed in these CPUs.

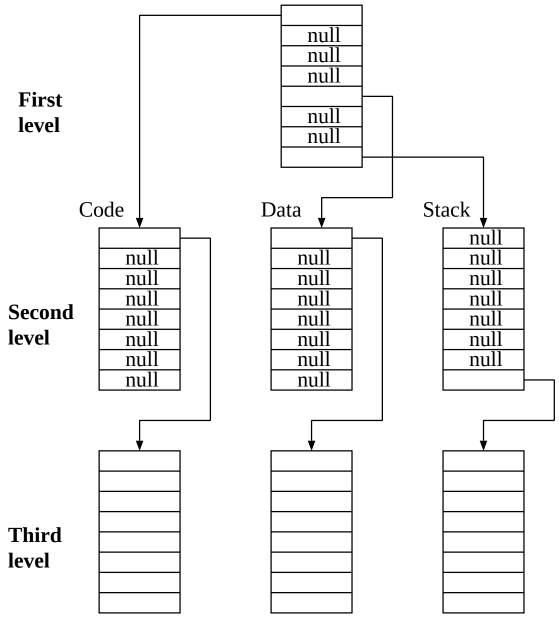

Figure 3.2: A typical page table in EMERALDS. The hierarchical structure is used to reduce the size of the page table.

tables exist. An attempt to access an address covered by an invalid entry will result in a TRAP to the kernel indicating a bug in the software. Similarly, in each second-level page table, only one entry is valid and all other third-level page tables do not exist. This way, total size of the page table is just 2432 bytes for a page size of 8 kbytes. (More third-level page tables are needed if any segment exceeds 256 kbytes). While this example is specific to the MC 68040, most other modern CPUs also provide three-level page tables with similar parameters.

The small size of page tables not only saves memory, but also enables other optimizations like mapping the kernel into every address space (see Section 3.3). This greatly reduces the overhead associated with system calls, making our implementation of memory protection feasible for embedded systems.

3.3 Efficient System Call Mechanism

Above we mentioned the advantages of memory protection. Its disadvantage is the context switch overhead incurred when making system calls (because the user and kernel usually exist in separate address spaces). This is why some RTOSs omit memory protection so they only have to make subroutine calls to access kernel services; not so with memory

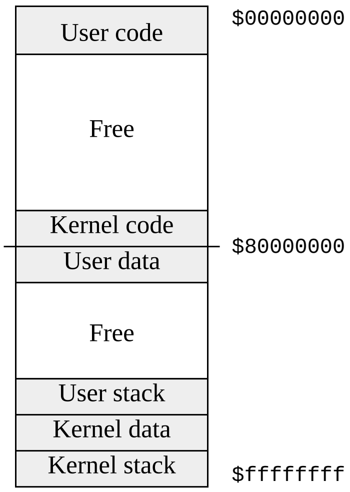

Figure 3.3: A typical address space in EMERALDS. Area labeled *kernel stack* is used for interrupts and area labeled *user stack* is used by both the user and the kernel.

protection.

We resolved this problem by mapping the kernel into each user-level address space (unlike other OSs in which the kernel runs in its own address space). A typical 32-bit EMERALDS address space is shown in Figure 3.3.

With this type of mapping, a switch from user to kernel involves just a TRAP (which switches the CPU from user to kernel/supervisor mode) and a jump to the appropriate address; there is no need to switch address spaces. Also, system call code in EMERALDS is designed to take parameters straight off the user's stack (possible since both kernel and user are in the same address space). This scheme has the following advantages.

- No need to copy parameters from user space to kernel space. All that Locore.S (assembly code used for making system calls) does is point the kernel stack pointer to the user stack and some other minor stack adjustments. As a result, system calls in EMERALDS (except those involving servers) have an overhead comparable to that of a subroutine call (see Section 3.6).

- No need to translate pointers. If the user and kernel are in separate address spaces, the data pointed to by a pointer must be copied to the kernel's address space, and a pointer to this copy passed to the system call routine (or at least, the appropriate user memory pages need to be mapped into the kernel's address space). But in EMERALDS, all user pointers are valid inside the kernel, so no need to do any data copying.

Implementation: Mapping the kernel into each user address space is feasible in EMERALDS because both the kernel and its data segment are so small. In other operating systems with standard virtual memory, the size of the kernel's data segment is so large (due to large page tables) that mapping it into each user address space is not feasible (unless hardware support is available as already mentioned in Section 3.2). The mapping is achieved by having appropriate second-level page table entries point to common third-level page tables which map the kernel. Thus, size of a process's page table is not affected. Also, the kernel areas are protected from corruption by faulty user code by using page table entries to mark them as read-only for user mode. This way, user processes are protected from each other and the kernel is protected from user processes.

3.4 Inter-Process Communication (IPC)

The primary IPC mechanism in EMERALDS — for both inter- and intra-processor communication — is message-passing. For intra-processor communication, EMERALDS also provides shared memory as well as a specialized, high-efficiency local message passing scheme.

3.4.1 Message-Passing Using Mailboxes

EMERALDS provides the system calls listed in Table 3.2 for exchanging messages between threads. These calls are used to create & delete mailboxes and send & receive messages. EMERALDS also allows a 32-bit priority to be assigned to each message which is used to sort messages in a mailbox so that the receiver thread retrieves the highest-priority message first.

Message-passing in EMERALDS has been designed with efficiency and flexibility in mind. Most communication networks designed for embedded, real-time systems such as CAN [47], TTP [61], SERCOS [46], SP50 [44], etc., provide the bottom two layers of the ISO OSI reference stack (the physical and data-link layers) which is sufficient for exchanging simple messages (all that the sender has to do is talk directly to the network device driver). For more complex IPC, the remaining stack layers must be implemented in software. So EMERALDS allows both direct network access as well as use of protocol stacks. EMER-

System call	*Important Parameters*	*Function*
`mbox_create()`	CPU-wide unique identifier	Create mailbox
`mbox_delete()`	Mailbox identifier	Delete mailbox
`msg_send()`	Destination node and mailbox, local server mailbox	Send message to mailbox
`msg_receive()`	Mailbox identifier	Retrieve message from mailbox
`try_msg_receive()`	Mailbox identifier	Non-blocking version of `msg_receive()`

Table 3.2: Message-passing system calls. The last two calls are for use by protocol servers.

ALDS also provides optimizations for local message-passing between threads on the same node. Here we describe the EMERALDS mechanisms for local message-passing. Details of the network communication architecture of EMERALDS are in Chapter 6.

Local Message-Passing:

Suppose thread $T1$ wants to send a message to another thread $T2$ on the same node. The latter has a mailbox with identifier $M2$. Thread $T1$ will use the `msg_send()` system call to send this message, specifying the destination mailbox ($M2$ in this case) and message priority. The kernel deposits the message directly into $M2$, then unblocks $T1$. $T2$ can then retrieve this message from $M2$ in two ways:

1. $T2$ can execute a `msg_receive()` system call.
2. When $T2$ creates $M2$, it can specify an interrupt service routine (ISR) to be executed whenever a message arrives in $M2$. This ISR may execute `msg_receive()` to retrieve the message.

The first mechanism is suitable for periodic messages while the second one is for infrequent sporadic and aperiodic messages.

Note that `msg_receive()` is a blocking system call which may not always be suitable for real-time systems. Thus, EMERALDS provides its non-blocking counterpart `try_msg_receive()` which returns an error if a mailbox has no messages.

3.4.2 Local Message-Passing Using State Messages

From the performance point of view, global variables are ideal for sharing information between tasks, but if reading from and writing to global variables is not regulated, subtle bugs can crop up in the application code. State messages [60] use global variables to pass messages between tasks, but these variables are managed by code generated automatically by a software tool, not by the application designer. In fact, the application designer does not even know that global variables are being used: the interface presented to the programmer is almost the same as the mailbox-based message-passing interface.

State messages are not meant to replace traditional message-passing, but are meant as an efficient alternative to traditional message-passing for a wide range of situations as explained next.

State Message Semantics

State messages solve the single-writer, multiple-reader communication problem. One can imagine that state message "mailboxes" are associated with the senders, not with the receivers: only one task can send a state message to a "mailbox" (call this the *writer* task) but many tasks can read the "mailbox" (call these the *reader* tasks). This way, state message "mailboxes" behave very differently from traditional mailboxes, so from now on we will call them *SMmailboxes*. The differences are summarized below:

- SMmailboxes are associated with the writers. Only one writer may send a message to an SMmailbox, but multiple readers can receive this message.
- A new message overwrites the previous message.
- Reads do not consume messages, unlike standard mailboxes for which each read operation pops one message off the message queue.
- Both reads and writes are non-blocking. This reduces the number of context switches suffered by application tasks.

Usefulness

In real-time systems, a piece of data such as a sensor reading is valid only for a certain duration of time, after which a new reading must be made. Suppose task T_1 reads a sensor and supplies the reading to task T_2. If T_1 sends two such messages to T_2, then the first message is useless because the second message has a more recent and up-to-date sensor reading. If traditional mailboxes with queues are used for communication, then T_2 must first read the old sensor reading before it can get the new one. Moreover, if multiple tasks need the same sensor reading, T_1 must send a separate message to each.

State messages streamline this entire process. An SMmailbox $SM1$ will be associated with T_1 and it will be known to all tasks that $SM1$ contains the reading of a certain sensor. Every time T_1 reads the sensor, it will send that value to $SM1$. Tasks which want to receive the sensor value will perform individual read operations on $SM1$ to receive the most up-to-date reading. Even if T_1 has sent more than one message to $SM1$ between two reads by a task, the reader task will always get the most recent message without having to process any outdated messages. More importantly, if a reader does two or more reads between two writes by T_1, the reader will get the same message each time *without blocking*. This makes perfect sense in real-time systems because the data being received by the reader is still valid, up-to-date, and useful for calculations.

The single-writer, multiple-reader situation is quite common in embedded real-time systems. Any time data is produced by one task (may it be a sensor reading or some calculated value) and is to be sent to one or more other tasks, state messages can be used. But in some situations, blocking read operations are still necessary such as when a task must wait for an event to occur. Then, traditional message-passing and/or semaphores must be used. Hence, state messages do not replace traditional message passing for all situations, but they do replace it for most inter-task communication requirements in embedded applications.

Previous Work

State messages were first used in the MARS OS [60] and have also been implemented in ERCOS [92]. The state message implementation used in these systems as described in [62] is as follows. The problem with using global variables for passing messages is that a reader may read a half-written message since there is no synchronization between readers and writers. This problem is solved by using an N-deep circular buffer for each state message. An associated pointer is used by the writer to post messages, and used by readers to retrieve the latest message. With a deep enough buffer, the scheme can guarantee that data will not be corrupted while it is being read by a reader, but a large N can make state messages infeasible for our limited-memory target applications.

The solution presented in [62] limits N by having readers repeat the read operation until they get uncorrupted data. This saves memory at the cost of increasing the read time by as much as several hundred microseconds, even under the assumption that writers and readers run on separate processors with shared memory. With such an architecture, it is not possible for a reader to preempt a writer. But we want to use state messages for communication between readers and writers on the same CPU without increasing the read overheads. For this situation, depending on the relative deadlines of readers and writers, N may have to be in the hundreds to ensure correct operation.

Our solution to the problem is to provide OS support for state messages to reduce N to no more than 5–10 for all possible cases. In the following, we describe our implementation scheme for state messages including the calculation of N for the case when both readers and writers are on the same CPU. Then, we describe a system call included in EMERALDS to support state messages.

Implementation in EMERALDS

Let B be the maximum number of bytes the CPU can read or write in one instruction. For most processors, $B = 4$ bytes. We have implemented a tool called **MessageGen** which produces customized code for the implementation of state messages depending on whether the message length L exceeds B or not.

The case for $L \leq B$ is simple. **MessageGen** assigns one L-byte global variable to the state message and provides macros through which the writer can write to this variable and readers can read from it. Note that for this simple case, it is perfectly safe to use global variables. The only complication possible for a global variable of length $< B$ is to have one writer accidentally overwrite the value written to the variable by another writer. But this problem cannot occur with state messages because, by definition, there is only one writer.

For the case of $L > B$, **MessageGen** assigns an N-deep circular buffer to each state message. Each slot in the buffer is L bytes long. Moreover, each state message has a 1-byte index I which is initialized to 0. Readers always read slot I, the writer always writes to slot $I + 1$, and I is incremented only after the write is complete. This way readers always get the most recent, consistent copy of the message.

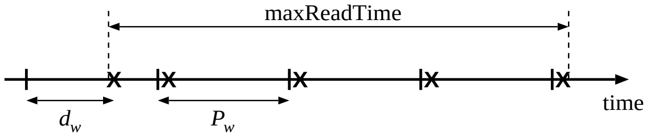

Figure 3.4: Calculation of x_{max}. Write operations are denoted by **X**. Excluding the first write, there are $\lfloor(maxReadTime - (P_w - d_w))/P_w\rfloor = 4$ writes, so $x_{max} = 5$.

Calculating Buffer Depth N: Now, we address the issue of how to set N, the depth of the buffer. It can happen that a reader starts reading slot i of the buffer, is preempted after reading only part of the message, and resumes only after the writer has done x number of write operations on this message. Then, N must be greater than the largest value x can take:

$$N = \max(2, x_{max} + 1).$$

Let maxReadTime be the maximum time *any* reader can take to execute the read operation (including time the reader may stay preempted). Because all tasks must complete by their deadlines (ensured by the scheduler), the maximum time any task can be preempted is $d - c$, where d is its deadline and c is its execution time. If c_r is the time to execute the read operation, then maxReadTime $= d - (c - c_r)$.

The largest number of write operations possible during maxReadTime occur for the situation shown in Figure 3.4 when the first write occurs as late as possible (just before the deadline of the writer) and the remaining writes occur as soon as possible after that (right at the beginning of the writer's period). Then,

$$x_{max} - 1 = \left\lfloor \frac{\text{maxReadTime} - (P_w - d_w)}{P_w} \right\rfloor$$

where P_w and d_w are the writer's period and deadline respectively. Then, N can be calculated using x_{max}.

Slow Readers: If it turns out that one or more readers have long periods/deadlines (call them *slow* readers) and as a result, x_{max} is too large (say, 10 or more) and too much memory will be needed for the buffer, then EMERALDS provides a system call which executes the same read operation as described above, but disables interrupts so that copying the message from the buffer becomes an atomic operation. This call can be used by the slow readers while the faster readers use the standard read operation. By doing this, N depends only on the faster readers and memory is saved. The disadvantage is that the system call takes longer than the standard read operation. But this system call is invoked only by slow readers, so it is invoked infrequently and the extra overhead per second is negligible. Note that the write operation is unchanged no matter whether the readers are slow or fast.

3.4.3 Shared Memory

EMERALDS allows page-based sharing of memory between processes running on the same CPU. Two system calls are provided for this purpose: `shm_attach()` and `shm_detach()`.

The system call `shm_attach()` is called with an identifier. If no shared-memory segment exists with this identifier, then physical memory is allocated and a new segment is created. This segment is mapped into the calling process's address space and a pointer to the start of the segment is returned. When `shm_attach()` is called again with the same identifier by any process on the same CPU, the kernel finds the segment with that identifier and maps it into the calling process's address space (no new memory is allocated).

The system call `shm_detach()` does the opposite of `shm_attach()`. It unmaps the named segment from the calling process's address space. Moreover, if no other process has this segment mapped into its address space, then the physical memory associated with the segment is also freed up. This provides a simple programming model. When processes need to use a shared memory segment, they call `shm_attach()` with that segment's identifier. The first such call allocates physical memory and all the later ones just map in the segment. When processes no longer need a segment, they call `shm_detach()`. These calls unmap the segment from their address space, except the last call which also frees up the physical memory. These semantics are easier to use than, for example, UNIX [110] semantics where shared memory must be explicitly created before mapping it into an address space, and must be explicitly deleted after unmapping it from each address space.

3.5 Miscellaneous OS Services

3.5.1 Semaphores

Threads often need to ensure mutual exclusion when accessing critical regions of code dealing with shared resources. EMERALDS provides semaphores (sometimes also known as *mutexes* as in POSIX terminology) for this purpose. The system calls in Table 3.3 are used to create, delete, lock, and unlock semaphores. If a thread tries to acquire a semaphore which is already locked, that thread will block and will be added to a queue of threads waiting for that semaphore. When the lock holder releases the semaphore, the highest-priority thread in the queue will be unblocked. An alternative to the blocking `sem_lock()` call is the `sem_trylock()` call which returns an error if the semaphore is already locked. If the semaphore is free, it will be locked. EMERALDS uses certain optimizations to reduce the overhead of semaphore locking. Details are in Chapter 5

System call	*Important Parameters*	*Function*
`sem_create()`	CPU-wide unique ID	Create sem.
`sem_delete()`	Semaphore identifier	Delete sem.
`sem_lock()`	Semaphore identifier	Acquire sem.
`sem_trylock()`	Semaphore identifier	Non-blocking
`sem_unlock()`	Semaphore identifier	Release sem.
`cv_create()`	CPU-wide unique ID	Create CV
`cv_delete()`	CV identifier	Delete CV
`cv_lock()`	CV identifier	Acquire CV
`cv_unlock()`	CV identifier	Release CV

Table 3.3: System calls for semaphores and condition variables.

3.5.2 Condition Variables

Condition variables differ from semaphores in the effect of signaling the variable. When a semaphore is signaled (using `sem_unlock()`), its effect lasts. This means that even if no thread is currently blocked waiting for the semaphore, the signal will not be lost. If later on a thread tries to acquire the semaphore, it will succeed. On the other hand, signaling a condition variable has no effect if no threads are waiting on that condition variable at the time of the signal.

The system calls for condition variables are similar to those for semaphores and are listed in Table 3.3. Note that a system call such as `cv_trywait()` does not make sense with condition variable semantics, so it is not provided by EMERALDS.

3.5.3 Device Drivers

Since there are so many devices (e.g., sensors, actuators, network adapters) used in embedded systems, it is virtually impossible for the OS designer to supply device drivers for all of them. The next best thing is to make it as easy as possible for users to write their own user-level device drivers. EMERALDS does just that. A device driver is a user process (instead of being part of the kernel), and special system calls are available for device drivers to access devices and deal with interrupts. EMERALDS also uses in-kernel device drivers where appropriate for efficiency reasons.

EMERALDS provides two special system calls to write user-level device drivers. The first, `map_device()`, allows a device driver to map a memory-mapped device into its address space.[2] From then on, the device driver can use standard memory operations to access the device. The `set_isr()` system call allows device drivers to handle interrupts. Device drivers use this call to tell the kernel which ISR subroutine to execute when an interrupt occurs. A separate ISR can be attached to each interrupt level. As far as communication between user threads and device drivers is concerned, standard EMERALDS IPC mechanisms (message-passing and shared memory) can be used since EMERALDS device drivers run as user-level threads.

EMERALDS allows even non-device driver threads to use the above system calls. When used responsibly, this can be a great asset in embedded real-time systems. Usually, just one (possibly multi-threaded) process is responsible for directly communicating with a certain device like a sensor or an actuator. In this situation, it becomes very efficient to integrate the device driver with that process. This way, the device can be accessed using subroutine calls — completely avoiding context switch overheads.

3.5.4 Memory Management

The system call `mem_alloc()` can be used by a process to get the desired number of pages of physical memory mapped into its address space. This call returns the starting address of the allocated space, and can be used to build library-based memory allocators to provide C calls like malloc() and free(). Memory obtained through `mem_alloc()` is retained by a process until it terminates, at which time all its memory is reclaimed by the system.

[2] Currently, EMERALDS does not support I/O-mapped devices because the MC 68040 — on which EMERALDS is presently implemented — does not have a separate I/O space.

3.5.5 Timers

The call `start_timer()` can be used to create and start a timer. This call has two variants. The first is a blocking version, in which the calling thread blocks for the specified duration of time. The second non-blocking version is used to execute a timer ISR. The calling thread specifies a routine to be executed as the ISR and a time delay. When the timer expires, the ISR is executed, and it can reset the timer for the next interrupt. This way, the ISR can execute periodically.

3.6 Performance

We have completed a uniprocessor version of EMERALDS for the MC 68040 processor which is one of the most popular and commonly used processor in embedded systems with a wide installed base. The size of this version of EMERALDS is about 13 kbytes. Comparing this to other major RTOSs for embedded applications (Table 3.4), we see that our goal of a small-sized RTOS has been achieved.

RTOS	*Size (kbytes)*
QNX	101
VxWorks 5.1	286
EMERALDS	13

Table 3.4: Sizes of various RTOSs (uniprocessor versions). Size of QNX is from [39] and includes the "kernel," `Proc`, and `Dev` modules which is the minimal configuration with device driver support. VxWorks' size is from a compiled stand-alone version.

Table 3.5 shows the latencies of some system calls and other operations in the current version of EMERALDS on a 25 MHz 68040 processor with two independent 4 kbyte instruction and data caches. Latencies are measured using a 5 MHz clock (the fastest clock available on the Ironics IV-3207 boards we use). The operations labeled with * involve a context switch to another thread. All other operations return to the calling thread.

Comparing the `null()` system call to the `null()` subroutine call, we see that EMERALDS' technique of mapping the kernel into each address space results in efficient system calls, incurring only a 1.8 μs more overhead than subroutine calls. Even when a context

Operation	*Latency (µs)*
Context switch (thread to thread)	9.2
`null()` subroutine call	0.2
`null()` system call	2.0
`create_proc()`*	194.4
`create_thread()`*	50.4
`join_thread()` (thread already exited)	17.2
`detach_thread()` (thread already exited)	16.4
`detach_thread()` (thread has not exited)	2.2
`shm_attach()` (one page mem. allocated)	10.6
`shm_attach()` (attach existing segment)	8.6
`shm_detach()` (one page mem. deallocated)	10.0
`shm_detach()` (just unmap segment)	8.0
`sem_create()`	4.8
`sem_delete()`	3.2
`sem_lock()` (semaphore free)	5.8
`sem_lock()` (with threads waiting)	Chap 5
`sem_unlock()` (no thread waiting)	7.0
`sem_unlock()` (with threads waiting)	Chap 5
`cv_create()`	6.2
`cv_delete()`	5.4
`cv_wait()`*	25.4
`cv_signal()` (no thread waiting)	3.4
`cv_signal()`* (thread waiting)	30.4
`mbox_create()`	6.4
`mbox_delete()`	3.6
`msg_send()` (8 bytes)	16.0
`msg_receive()` (8 bytes)	7.6
State message `send` (8 bytes)	2.4
State message `receive` (8 bytes)	2.0
State message `receive_slow` (8 bytes)	4.4

Table 3.5: Timing of various operations in EMERALDS.

switch to a different address space is required, it incurs less than 10 μs overhead.

3.6.1 Comparison with Commercial RTOSs

The Scientific Research Laboratory (SRL) of Ford Motor Company evaluated the performance of EMERALDS and nine commercial embedded RTOSs for automotive engine control. These RTOSs include Nucleus, pSOS Select, RTX, RTXC, RTOS, C-Executive,

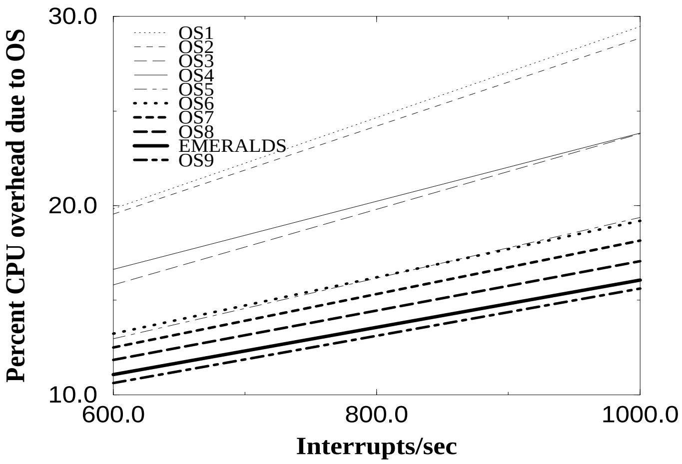

Figure 3.5: OS overhead due to interrupts, 250 periodic task switches/s and a 4ms clock tick timer.

VRTX mc, RTEK, and MTASK.

In initial testing, SRL has focused on measuring overheads of basic OS services like interrupt handling, task switching, timers, and clock tick on a 16.7MHz Motorola 68332 microcontroller. Their results[3] are shown in Figure 3.5 (the results released by SRL do not identify which measurements are for which OS, so we refer to the commercial RTOSs as OS1–OS9). The number of interrupts/second that the engine controller must service depends on the engine's speed. At high RPM (revolutions per minute), the controller sees about 1000 interrupts/s. At this rate, the various RTOSs have an overhead ranging from 15% to 30% of CPU time. EMERALDS is one of the best with only 16% overhead. Only OS9 has a lower overhead of 15.5%, but compared to other OSs, it has much higher RAM overhead (about 4000 bytes for 10 tasks compared to 500-1000 bytes for all the other OSs including EMERALDS) which makes OS9 infeasible for small-memory embedded systems. This makes EMERALDS the best OS among all the feasible OSs.

3.7 Conclusions and Future Work

Small to medium sized embedded real-time systems are becoming increasingly common in applications like automotive control, robotics, and industrial automation. To be competitive in the market, these systems must reduce cost to a minimum. Any RTOS to be used

[3]The 68332 does not have a MMU, so these results are for a version of EMERALDS without memory protection.

in these systems must therefore not only support predictability (essential in any real-time system and provided in EMERALDS in the form of predictable scheduling of threads) but also be efficient and small in size. Efficiency allows cheaper processors to be used and small size decreases the cost of ROM needed to store the executable code. Most other modern RTOSs are either too large in size (hundreds of kbytes or more) or they do not offer several popular OS features like memory protection and threads in an attempt to reduce size and increase speed. Our goal in designing EMERALDS was to develop an RTOS which was not only predictable but also small and efficient, without cutting back on standard OS services relevant to embedded systems. To achieve this goal, we made use of several features of embedded systems which allowed us to increase the efficiency of system calls and keep the size of EMERALDS to just 13 kbytes (uniprocessor version).

CHAPTER 4

COMBINED EDF AND RM SCHEDULING

Real-time computing systems must behave predictably even in unpredictable environments [103]. This predictability is ensured by system-level services, most important among them being the task scheduler in the RTOS.

Real-time task scheduling has been focus of active research for several decades [6, 69, 76, 106]. This has led to the development of well-known scheduling schemes such as *rate-monotonic* (RM) [76], *earliest-deadline-first* (EDF) [76], and *deadline-monotonic* [69]. But in recent years, the focus of research has shifted from uniprocessor task scheduling to scheduling tasks and messages in multiprocessors and distributed systems [2, 54, 117]. Uniprocessor task scheduling is treated as a "solved" problem and research in this area has tapered off.

Unfortunately, well-known uniprocessor task scheduling solutions such as RM and EDF are only "theoretical" solutions in the sense that they do not consider the practical implementation of these schedulers in real systems. For example, EDF delivers a processor utilization of 100%, but not all of this CPU capacity is available for execution of workload tasks. EDF incurs high run-time overhead in keeping tasks sorted by their (changing) deadlines. When this overhead is taken into consideration, the CPU capacity left for workload tasks is well below 100%. The static RM scheduler has much lower run-time overhead but its average-case schedulable utilization is only 88% [67] — well below that for EDF. In practice, neither EDF nor RM deliver good performance. In fact, for many real workloads, performance of both these schedulers is about the same [58].

The recent popularity of multimedia applications has led to renewed interest in making uniprocessor task scheduling efficient. In [33], a delayed preemption scheme is presented in which a running task is preempted only at quantized time boundaries. This scheme is useful for protocol data processing since it allows relatively short packet-handling tasks to execute to completion before being preempted. However, utility of such delayed preemption schedulers in handling application task workloads is yet to be demonstrated. The Rialto scheduler [52] uses time-slice scheduling to reduce run-time overhead to a minimum, but it employs heuristics for constructing the schedule, resulting in non-optimal solutions.

In embedded systems, scheduler inefficiencies become a major concern because of relatively slow CPUs. This led us to investigate ways to reduce scheduler overhead and improve performance.

Our approach to solving this problem was not to invent new scheduling theory but instead to make existing schedulers (EDF and RM in particular) work better in real implementations. We present a new uniprocessor task scheduling scheme called the *combined static/dynamic* (CSD) scheduler. It combines the best features of RM and EDF to deliver better performance than both *when execution overheads are factored in.* CSD lowers run-time overhead by using multiple scheduling queues, but at the same time, delivers high schedulable utilization by properly assigning tasks to the different queues. A proper partitioning of tasks is critical to the good performance of CSD. We present an iterative method to partition the tasks in a given workload into two groups. When tasks in one group are scheduled by EDF and tasks in the other group are scheduled by RM, the *total scheduling overhead* (run-time overhead plus schedulable utilization being less than 100%) is less than that of EDF and RM. The total scheduling overhead is a "true" measure of the performance of a scheduler. By reducing this total overhead, CSD outperforms both RM and EDF in real systems.

We have implemented CSD in EMERALDS. We measure the run-time overheads associated with CSD, EDF, and RM and we demonstrate that when these overheads are considered in schedulability tests, CSD feasibly schedules more workloads than EDF or RM.

The next section discusses some of the overheads associated with task scheduling and motivates the need for a new scheduling scheme which reduces these overheads. Section 4.2 describes the CSD scheduler and gives the theoretical basis for its superiority over EDF and RM. Section 4.3 gives an extension to the basic CSD scheme which overcomes some of the shortcomings of its original form. We experimentally evaluate the performance of CSD in

Section 4.4. Section 4.5 puts CSD in context of previous research in real-time scheduling and highlights the novelty of CSD. The chapter concludes with Section 4.6.

4.1 Task Scheduling Overheads

Consider a periodic task which runs once every 1ms. For just this one task, the scheduler must run twice every 1ms: once when the task is released and once when the task completes. Considering that typical OS operations usually take 40–50μs and that a typical task workload consists of 20–40 tasks with at least 5–7 tasks having periods less than 10ms, the scheduler's execution alone can use up 5–15% of CPU time. This is why some application programmers prefer cyclic time-slice scheduling techniques in which the entire schedule is calculated either off-line or at task-admission time, and at run-time, tasks are switched in and out according to this fixed schedule. This reduces the scheduler's run-time overhead but introduces several problems:

- The schedules must be calculated by hand, so they are difficult and costly to modify if the task characteristics change during the application design process. Heuristics can be used to calculate schedules, but they result in non-optimal solutions (some feasible workloads may get rejected).
- Cyclic schedulers give poor response times for high-priority aperiodic tasks because the arrival times of these tasks cannot be anticipated off-line.
- If a workload contains both short and long period tasks (as is often the case in control applications), the resulting time-slice schedule can be quite large, consuming significant amounts of memory.

With real-time systems now having more tasks and more aperiodic activities, cyclic schedulers are no longer suitable for task scheduling. The alternative is to turn to priority-driven schedulers like RM and EDF which use task priorities to make run-time decisions as to which task should execute when. These schedulers do not require any costly off-line analysis, can easily handle changes in the workload during the design process, and can handle aperiodic tasks as well using, for example, a sporadic server [106]. However, since priority-driven schedulers make run-time scheduling decisions, they incur overhead which can be 5–15% of CPU time. This calls for new task scheduling schemes with lower overheads which would free up more time for the execution of application tasks.

4.2 Combined Static/Dynamic Scheduler

The task scheduler's overhead can be broken down into two components: the *run-time overhead* and the *schedulability overhead*. The run-time overhead is the time consumed by the execution of the scheduler code. This has to do with managing the queues of tasks and selecting the highest-priority task to execute whenever some task blocks or unblocks.

The schedulability overhead is defined as $1-U^*$, where U^* is the *ideal schedulable utilization*. For a given workload and a given scheduler, U^* is the highest workload utilization that the scheduler can feasibly schedule under the ideal conditions that the scheduler's run-time overhead is ignored. This is best explained through examples. Consider a workload of n tasks, $\{\tau_i : i = 1, 2, \ldots, n\}$. Each task τ_i has a period P_i, execution time c_i, and deadline d_i. Then this workload has utilization $U = \sum_{i=1}^{n} c_i/P_i$. EDF is a dynamic-priority scheduler which gives highest priority to the earliest-deadline task [76], and can schedule all workloads with $U \leq 1$ under the ideal condition that EDF's run-time overhead is ignored. We say that $U^* = 1$ for EDF. Other schedulers such as RM (which schedules tasks according to fixed priorities based on the tightness of their P_i [76]) can have $U^* < 1$. For example, a workload with $U = 0.90$ may be schedulable under RM, but if some c_i is slightly increased so that U becomes 0.91, the workload may no longer be schedulable even under ideal conditions. We say that $U^* = 0.90$ for this workload under RM. This means that 10% of CPU time is "unusable" because of the scheduling policy, and we refer to this as the *schedulability overhead*.

EDF has zero schedulability overhead but high run-time overhead. By contrast, RM has low run-time overhead but, depending on the workload, it can cause significant schedulability overhead. In the rest of this section, we analyze the sources of these overheads and then design a mechanism to yield high schedulability with low-run-time overhead. Our goal is not to devise new scheduling theory but to use the best features of existing schedulers (EDF and RM) to get good performance in real implementations.

4.2.1 Run-time Overhead

The run-time overhead (Δt) has to do with parsing queues of tasks and adding/deleting tasks from these queues.

When a running task blocks, the OS must update some data structures to identify the task as being blocked and then pick a new task for execution. We call the overheads associated with these two steps the *blocking overhead* Δt_b and the *selection overhead* Δt_s, respectively. Similarly, when a blocked task unblocks, the OS must again update some internal data structures, incurring the *unblocking overhead* Δt_u. The OS must also pick a task to execute (since the newly-unblocked task may have higher priority than the previously-executing one), so the selection overhead is incurred as well.

Each task blocks and unblocks at least once every period: it is unblocked at the beginning of the period and then blocks itself after executing for c_i time units. This means that the minimal scheduler run-time overhead per task τ_i is $\Delta t_b + \Delta t_u + 2\Delta t_s$ incurred once every period. Overhead is even greater if τ_i uses blocking system calls during execution. This is application-dependent, but we assume that half the tasks block once during their execution. For simplicity, we assume that each task suffers a run-time overhead of $\Delta t = 1.5(\Delta t_b + \Delta t_u + 2\Delta t_s)$. Then, with the run-time scheduler overhead figured in, the workload utilization becomes $U = \sum_{i=1}^{n}(c_i + \Delta t)/P_i$ which can be significantly greater than the utilization when Δt is ignored.

Now, we calculate Δt for both EDF and RM scheduling policies. Our calculations are based on a linked list implementation of schedulers. A sorted heap can give lower run-time overhead for a large number of tasks, but linked lists are more efficient for the relatively smaller number of tasks (15–40) typically seen in real-time systems. Experimental measurements to corroborate this statement are presented in Section 4.4.

In EMERALDS, we have implemented EDF as follows. All blocked and unblocked tasks lie in a single, unsorted queue. This makes sense because task priorities change continually under EDF, so keeping the queue sorted is not worth the overhead. Tasks are blocked and unblocked by changing one variable in the appropriate task control block (TCB). To select the next task to execute, the entire list is parsed and the earliest-deadline ready task is picked. With this scheme, both Δt_b and Δt_u are $O(1)$, but Δt_s is $O(n)$, where n is the number of tasks. Since Δt_s is counted twice per task block/unblock operation, Δt for EDF increases rapidly as n increases.

The typical implementation for RM is to have a queue of ready tasks sorted by (fixed) task priorities. Blocking and unblocking involve deletion from, and insertion into, the list in sorted order. But in EMERALDS, we chose a different implementation which allows us to optimize other OS services (especially semaphores) while the run-time overhead stays about the same as for the typical implementation. All blocked and unblocked tasks are in a single queue sorted by priority, highest-priority task first. A single pointer `highestP` points to the highest-priority ready task, so Δt_s is $O(1)$ because `highestP` points to the task which should execute next. To block a task, one variable is updated in the TCB (same as in EDF), but now `highestP` has to be updated as well. The scheduler parses down the queue till it finds the next ready task in the queue, then sets `highestP` to point to that task. This is why Δt_b takes $O(n)$ time in the worst case. On the other hand, unblocking a task only involves checking if the unblocked task has higher priority than the `highestP` task. If so, `highestP` is simply reset to point to the newly-unblocked task and this takes $O(1)$ time.

For RM, $\Delta t_b = O(n)$ whereas for EDF, $\Delta t_s = O(n)$. Δt_b is counted only once every task block/unblock operation while Δt_s is counted twice, which is why $\Delta t = 1.5(\Delta t_b + \Delta t_u + 2\Delta t_s)$ is significantly less for RM than it is for EDF, especially when n is large (20 or more).

4.2.2 Schedulability Overhead

We have already mentioned that EDF has zero schedulability overhead, so if the run-time overhead is ignored, no scheduler can be better than EDF. Previous work has shown that on average, $U^* = 0.88$ for RM [67]. To see why U^* for RM is less than that for EDF, consider the workload shown in Table 4.1. Each task τ_i has deadline $d_i = P_i$. $U = 0.88$ for this workload, so it is feasible under EDF.

Figure 4.1 shows what happens if this workload is scheduled by RM. In the time interval $[0, 4)$, tasks τ_1–τ_4 execute, but before τ_5 can run, τ_1 is released again. Under RM, τ_1–τ_4 have higher priority than τ_5 (because of their shorter P_i), so the latter cannot run until all of the former execute for the second time, but by then τ_5 has missed its deadline. This makes the workload infeasible under RM and illustrates why RM has a non-zero schedulability overhead.

On the other hand, if EDF is used to schedule the same workload, τ_5 will run before

i	P_i (ms)	c_i (ms)
1	4	1
2	5	1
3	6	1
4	7	1
5	8	0.5
6	20	0.5
7	30	0.5
8	50	0.5
9	100	0.5
10	130	0.5

Table 4.1: An example task workload with $U = 0.88$. It is feasible under EDF but not under RM.

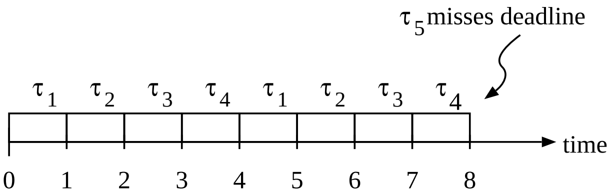

Figure 4.1: RM scheduling of the workload in Table 4.1.

τ_2–τ_4 run for the second time (because $d_5 = 8$ is earlier than the second deadlines of τ_2–τ_4) and the workload will be feasible (Figure 4.2).

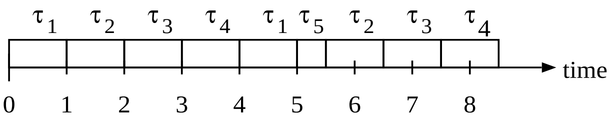

Figure 4.2: EDF scheduling of the workload in Table 4.1.

4.2.3 CSD: a Balance between EDF and RM

Going back to the workload in Table 4.1, notice that τ_5 is the "troublesome" task, i.e., because of this task the workload is infeasible under RM. Tasks τ_6–τ_{10} have relatively longer periods, so they can be easily scheduled by any scheduler, be it RM or EDF.

We used this observation as the basis of the CSD scheduler. Under CSD, tasks τ_1–τ_5 will be scheduled by EDF so that τ_5 will not miss its deadline (procedure to determine which task is the troublesome task in a given task set is discussed in Section 4.2.6). Once the troublesome task is taken care of, we can use the low-overhead RM policy to schedule the remaining tasks τ_6–τ_{10}. This way, the run-time overhead of CSD is less than that of

EDF (since the EDF queue's length has been halved) but a little more than that of RM. The schedulability overhead of CSD is the same as for EDF (i.e., zero) which is much less than that of RM. Thus, the total scheduling overhead of CSD is significantly less than that of both EDF and RM.

The CSD scheduler maintains two queues of tasks. The first queue is the *dynamic-priority* (DP) queue which contains tasks to be scheduled by EDF. The second queue is the *fixed-priority* (FP) queue which contains tasks to be scheduled by RM (or any other fixed-priority scheduler such as deadline-monotonic [69], but for simplicity, we assume RM is the policy used for the FP queue).

Given a workload $\{\tau_i : i = 1, 2, \ldots, n\}$ with tasks sorted by their RM-priority (tasks with shorter periods have lower index i), let τ_r be the "troublesome" task in this workload. Then, tasks τ_1–τ_r are placed in the DP queue while τ_{r+1}–τ_n are in the FP queue. CSD gives priority to the DP queue over the FP queue. This makes sense because all tasks in the DP queue have higher RM-priority (shorter periods) than any task in the FP queue. A single counter keeps track of the number of ready tasks in the DP queue. It is incremented when a DP task becomes ready and is decremented when a DP task blocks. When the scheduler is invoked, it first checks this counter. If it is greater than zero, the DP queue is parsed to pick the earliest-deadline ready task. Otherwise, the DP queue is skipped completely and the scheduler picks the highest priority ready task from the FP queue (pointed to by `highestP`).

4.2.4 Run-Time Overhead of CSD

We mentioned that CSD has zero schedulability overhead. Its run-time overhead depends on whether the task being blocked or unblocked is a DP or FP task. There are four possible cases:

1. <u>DP task blocks:</u> Δt_b is constant (same as for EDF), but Δt_s depends on whether any ready tasks are left in the DP queue or not. For real-time schedulability analysis, we are interested in the worst-case overhead, and this occurs when there *are* other ready tasks in the DP queue. Then, Δt_s is the time to parse the DP queue and is the same as Δt_s for EDF except that the queue length is only r instead of n. So, $\Delta t_s = O(r)$ instead of $O(n)$.

2. <u>DP task unblocks:</u> Δt_u is constant (same as for EDF). At least one ready task is definitely in the DP queue (the one that was just unblocked), so Δt_s is always the time to parse the r-long DP queue, i.e., $\Delta t_s = O(r)$.

3. FP task blocks: Δt_b is the same as for RM except the queue length is only $n - r$ so that $\Delta t_b = O(n - r)$. Regarding Δt_s, we need to know if any DP task can be ready or not. But this is not possible, because the task which just blocked is an FP task and this task could not have been executing had any DP tasks been ready. Since the DP queue has no ready tasks, the scheduler just selects `highestP` from the FP queue. This makes $\Delta t_s = O(1)$ (same as for RM).

4. FP task unblocks: Δt_u is a constant (same as for RM). The DP queue may or may not have ready tasks, but for the worst-case Δt_s, we must assume that it does, so $\Delta t_s = O(r)$. □

From this analysis, the total scheduler overhead for CSD is $\Delta t_b + \Delta t_{s\_block} + \Delta t_u + \Delta t_{s\_unblock}$ per task block/unblock operation. For DP tasks, this becomes $O(1) + O(r) + O(1) + O(r) = O(2r)$,[1] whereas for FP tasks, the overhead equals $O(n-r) + O(1) + O(1) + O(r) = O(n)$. This means that an r-long list is parsed twice for DP tasks (worst-case), while an n-long list is parsed once for FP tasks. Comparing this to EDF (n-long list parsed twice) and RM (n-long list parsed once), we see why the run-time overhead of CSD can be less than that of EDF (since r is less than n) and only slightly greater than that of RM. Considering that CSD has no schedulability overhead, it easily outperforms both EDF and RM.

4.2.5 Schedulability Test

A task set $\{\tau_i : i = 1, 2, \ldots, n\}$ with tasks sorted by their RM-priority (tasks with shorter periods have lower index i) is feasible under EDF if [76]

$$U = \sum_{i=1}^{n} \frac{c_i + \Delta t(EDF)}{P_i} \leq 1,$$

where $\Delta t(EDF)$ is Δt for EDF. The workload is feasible under RM if [67]

$$\forall i, 1 \leq i \leq n, \min_{0<t\leq d_i} \left(\sum_{j=1}^{i} \frac{c_j + \Delta t(RM)}{t} \left\lceil \frac{t}{P_j} \right\rceil \right) \leq 1.$$

In practice, this equation need only be evaluated for a finite number of t values as described in [63].

Schedulability under CSD is tested as follows. First, check if the DP tasks $\tau_1 - \tau_r$ are feasible under EDF:

[1] Strictly speaking, $O(2n) = 2O(n) = O(n)$, but we use the term $O(2n)$ to remind readers that an n-long queue is traversed twice.

$$U_{DP} = \sum_{i=1}^{r} \frac{c_i + \Delta t(DP)}{P_i} \leq 1.$$

Then, check the feasibility of the FP tasks as follows:

$$\forall i, r < i \leq n, \min_{0<t\leq d_i} \left(\sum_{j=1}^{i} \frac{c_j + \Delta t(X)}{t} \left\lceil \frac{t}{P_j} \right\rceil \right) \leq 1,$$

where X is DP or FP when j is a DP or FP task, respectively. This check is done only for FP tasks (i goes from $r+1$ to n), but it considers all the DP tasks as having higher priority than a given FP task (j goes from 1 to i).

4.2.6 Locating τ_r

The key to CSD's good performance is the proper partitioning of the workload into DP and FP tasks. Having two queues lowers run-time overhead but the low schedulability overhead of CSD depends on correctly identifying τ_r, then allocating τ_1–τ_r to the DP queue and the remaining tasks to the FP queue.

Task τ_r can be easily located through an iterative procedure by using the CSD schedulability test described above. For a given workload, start by assuming $r = 0$ and perform the schedulability test. If successful, then stop, otherwise keep increasing r until the schedulability test passes or r exceeds n in which case the workload is not feasible by CSD. This way, tasks can be partitioned between the two queues to minimize CSD's total scheduling overhead for any given workload.

4.3 Reducing Run-Time Overhead of CSD

CSD's main advantage is that even though it uses EDF to deliver good schedulable utilization, it cuts back on run-time overhead by keeping the DP queue short. But as the number of tasks in the workload increases, the DP queue's length also increases and this degrades CSD's performance. To rectify this situation, we modify CSD to keep run-time overhead under control as the number of tasks n increases.

4.3.1 Controlling DP Queue Run-Time Overhead

Under CSD, effective execution time of each task in the DP queue increases by $\Delta t(DP)$ which depends on length of the DP queue r. $\Delta t(DP)$ increases rapidly as r increases, which degrades performance of CSD.

		DP1	DP2	FP
Task	Δt_b	$O(1)$	$O(1)$	$O(n-r)$
Blocks	Δt_s	$O(\max(q, r-q))$	$O(r)$	$O(1)$
Task	Δt_u	$O(1)$	$O(1)$	$O(1)$
Unblocks	Δt_s	$O(q)$	$O(\max(q, r-q))$	$O(\max(q, r-q))$
Total run-time overhead		$O(r)$	$O(2r-q)$	$O(n-q)$

Table 4.2: Run-time overheads for CSD-3. The total values assume that the DP2 queue is longer than the DP1 queue ($\max(q, r-q) = r-q$) which is typically the case.

Our solution to this problem is to split the DP queue into two queues DP1 and DP2. DP1 has tasks with higher RM-priority (shorter periods), so the scheduler gives DP1 priority over DP2. We call this modified scheme CSD-3 because of its three queues. Properly allocating tasks to DP1 and DP2 is discussed in Section 4.3.3, but first, note that both DP1 and DP2 are expected to be significantly shorter than the original DP queue so that the run-time overhead of CSD-3 should be well below that of the original CSD scheme (which we will call CSD-2 from now on) as discussed next.

4.3.2 Run-Time Overhead of CSD-3

The run-time overheads for CSD-3 can be derived using the same reasoning as used for CSD-2 in Section 4.2.4. The overheads for different cases are shown in Table 4.2, where q is the length of the DP1 queue and r is the total number of DP tasks (so that $r-q$ is the length of DP2 queue). The table shows that the run-time overhead associated with DP1 tasks is $O(r)$ which is a significant improvement over $O(2r)$ for CSD-2. Since DP1 tasks are the shortest-period tasks in the workload, they are the ones which execute the most frequently and are responsible for most of the scheduling overhead. Reducing the run-time overhead associated with these tasks from $O(2r)$ to $O(r)$ leads to CSD-3 performing significantly better than CSD-2.

The run-time overhead of DP-2 tasks is reduced as well from $O(2r)$ in CSD-2 to $O(2r-q)$. Similarly, the overhead for FP tasks is reduced from $O(n)$ to $O(n-q)$.

4.3.3 Allocating Tasks to DP1 and DP2

If all DP tasks had the same periods, we could split them evenly between DP1 and DP2. Each queue's length will be half that of the original DP queue. This would cut the run-time overhead of scheduling DP tasks in half[2] and would give the best possible reduction in scheduler overhead. But when tasks have different periods, two factors must be considered when dividing tasks between DP1 and DP2:

- Tasks with the shortest periods are responsible for the most scheduler run-time overhead. For example, suppose $\Delta t = 0.1$ms. A task with $P_i = 1$ms will be responsible for $\Delta t/P_i = 10\%$ CPU overhead, whereas a task with $P_i = 5$ms will be responsible for only 2%. This means that only a few tasks with short periods should be kept in DP1 to keep $\Delta t(DP1)$ small. DP2 should have more tasks than DP1. This will make $\Delta t(DP2) > \Delta t(DP1)$, but this will balance out because tasks in DP2 have longer periods so that $\sum_i \Delta t/P_i$ for the two queues is approximately balanced.

- Balancing the run-time overhead between the queues cannot be made the sole criterion for allocating tasks to DP1 and DP2; the scheduling overhead must be considered as well. Once the DP tasks are split into two queues, they no longer incur zero schedulability overhead. Even though tasks within a DPx queue are scheduled by EDF, the queues themselves are scheduled by RM (all DP1 tasks have statically higher priorities than DP2 tasks), so that CSD-3 has non-zero schedulability overhead. Tasks must be allocated to DP1 and DP2 to minimize the *sum* of the run-time and schedulability overheads. For example, consider the workload in Table 4.1. Suppose the least run-time overhead results by putting tasks τ_1–τ_4 in DP1 and the rest of the DP tasks in DP2, but this will cause τ_5 to miss its deadline (see Figure 4.1). Putting τ_5 in DP1 may lead to slightly higher run-time overhead but will lower schedulability overhead so that τ_5 will meet its deadline.

At present, we use an exhaustive search (using the schedulability test described next) to find the best possible allocation of tasks to DP1, DP2, and FP queues. The search runs the schedulability test $O(n^2)$ times for three queues. This takes 2–3 minutes on a 167MHz Ultra-1 Sun workstation for a workload with 100 tasks.

4.3.4 Schedulability Test for CSD-3

As before, assume the task set $\{\tau_i : i = 1, 2, \ldots, n\}$ has tasks sorted by their RM-priority. Since DP1 tasks are scheduled by EDF, tasks $\tau_1 - \tau_q$ are feasible if:

---

[2] Increasing the number of queues also increases the overhead of parsing the prioritized list of queues, but our measurements showed this increase to be negligible (less than a microsecond) when going from two to three queues.

$$\sum_{i=1}^{q} \frac{c_i + \Delta t(DP1)}{P_i} \leq 1.$$

DP1 tasks have priority over DP2 tasks while DP2 tasks among themselves are scheduled by EDF. We modify the test for FP tasks to work for DP2 tasks. To check schedulability for a DP2 task i, the test treats all DP1 tasks as having higher priority than i (j runs from 1 to q), but checks deadlines of DP2 tasks (k runs from $q+1$ to r) to decide how many invocations of each (if any) have priority over the first invocation of i:

$$\forall i, q < i \leq r, \min_{0<t\leq d_i} \left(\sum_{j=1}^{q} \frac{c_j + \Delta t(DP1)}{t} \left\lceil \frac{t}{P_j} \right\rceil + \sum_{k=q+1}^{r} \frac{c_k + \Delta t(DP2)}{t} \left\lceil \frac{t}{P_k} \right\rceil^* \right) \leq 1,$$

where the function $\lceil x \rceil^*$ excludes the last invocation of j released before time t if its deadline exceeds d_i:

$$\left\lceil \frac{t}{P_k} \right\rceil^* = \begin{cases} \left\lceil \frac{t}{P_k} \right\rceil & \left(\left\lceil \frac{t}{P_k} \right\rceil - 1 \right) P_k + d_k \leq d_i \\ \left\lceil \frac{t}{P_k} \right\rceil - 1 & \text{otherwise} \end{cases}$$

This test for DP2 tasks uses the critical time zone assumption [76] which is valid only if all DP1 and DP2 tasks have utilization ≤ 1 ($\sum_{i=1}^{r} \frac{c_i + \Delta t(X)}{P_i} \leq 1$, X is $DP1$ or $DP2$ if i is a DP1 or DP2 task, respectively). Note that because of the check for deadlines, the critical time zone assumption is not automatically valid here as it is under rate-monotonic analysis.

The test for FP tasks is the same as for CSD-2 except for minor modifications:

$$\forall i, r < i \leq n, \min_{0<t\leq d_i} \left(\sum_{j=1}^{i} \frac{c_j + \Delta t(X)}{t} \left\lceil \frac{t}{P_j} \right\rceil \right) \leq 1,$$

where X is $DP1$, $DP2$, or FP when j is a DP1, DP2, or FP task, respectively.

4.3.5 Beyond CSD-3

The general scheduling framework of CSD is not limited to just three queues. It can be extended to have $4, 5, \ldots, n$ number of queues. The two extreme cases (one queue and n queues) are both equivalent to RM while the intermediate cases give a combination of RM and EDF.

We would expect CSD-4 to have slightly better performance than CSD-3 and so on (as confirmed by evaluation results in Section 4.4.2), although the performance gains are expected to taper off once the number of queues gets large and the increase in schedulability overhead (from having multiple EDF queues) starts exceeding the reduction in run-time overhead.

	EDF using queue (μs)	RM using queue (μs)	RM using sorted heap (μs)
Δt_b	1.6	$1.0 + 0.36n$	$0.4 + 2.8\lceil \log_2(n+1) \rceil$
Δt_u	1.2	1.4	$1.9 + 0.7\lceil \log_2(n+1) \rceil$
Δt_s	$1.2 + 0.25n$	0.6	0.6

Table 4.3: Run-time overheads for EDF and RM (n is the number of tasks). Also shows measurements for RM when a heap is used instead of a linked list. Measurements made using a 5MHz on-chip timer.

For a given workload, the best number of queues and the best number of tasks per queue can be found through an exhaustive search, but this is a computationally intensive task and is not discussed further in this chapter. This chapter demonstrates the usefulness of the general CSD scheduling framework and how it can be beneficial in real systems. Addressing issues related to optimal configuration of CSD for a given workload is part of future work.

4.4 Performance Evaluation

In this section, we evaluate the usefulness of CSD in scheduling a wide variety of workloads, by comparing CSD to EDF and RM. In particular, we want to know which is the best scheduler when all scheduling overheads (run-time and schedulability) are considered. The EDF and RM run-time overheads for EMERALDS measured on a 25MHz Motorola 68040 processor [87] with separate 4kbytes instruction and data caches are in Table 4.3. The run-time overhead of CSD is derived from these values as already discussed in Sections 4.2.4 and 4.3.2. The overhead to parse the list of queues in CSD-x (to find a queue with ready tasks) was measured at 0.55μs per queue.

Table 4.3 also shows the run-time overhead for RM when a sorted heap is used instead of a linked list to hold the tasks. The total run-time overhead Δt for a heap is more than that for a queue for $n \leq 58$. Most real-time workloads do not have enough tasks to make heaps feasible, so for the rest of this section we use the measurements for queues.

Our test procedure involves generating random task workloads, then for each workload, scaling the execution times of tasks until the workload is no longer feasible for a given scheduler. The utilization at which the workload becomes infeasible is called the *breakdown utilization* [56]. We expect that with scheduling overheads considered, CSD will have the highest breakdown utilization.

4.4.1 Results

Because scheduling overheads are a function of the number of tasks (n) in the workload, we tested all schedulers for workloads ranging from $n = 5$ to $n = 50$. For each n, we generate 500 workloads with random task periods and execution times. We scale the execution times and check feasibility using the schedulability tests in Sections 4.2.5 and 4.3.4, until the workload becomes infeasible.

The run-time overhead of priority-based schedulers depends not only on the number of tasks but on the periods of tasks as well (since the scheduler is invoked every time a task blocks or unblocks). Short period tasks lead to frequent invocation of the scheduler, resulting in high run-time overhead, whereas long period tasks produce the opposite result. In our tests, we vary not only the number of tasks but the periods of tasks as well. We do this by generating a base workload (with a fixed n), then producing three workloads from it by dividing the periods of tasks by a factor of 1, 2, and 3. This allows us to evaluate the impact of varying task periods on various scheduling policies.

We generate base task workloads by randomly selecting task periods such that each period has an equal probability of being single-digit (5–9ms), double-digit (10–99ms), or triple-digit (100–999ms). Figures 4.3–4.5 show breakdown utilizations when task periods are divided by 1, 2, and 3, respectively. In Figure 4.3, task periods are relatively long (5ms–1s). The run-time overheads are low which allows EDF to perform close to its theoretical limits. Even then, CSD performs better than EDF. CSD-4 has 17% lower total scheduling overhead for $n = 15$ and this increases to more than 40% for $n = 40$ as EDF's strong dependency on n begins to degrade its performance.

Figure 4.4 is for periods in the 2.5ms–500ms range. For these moderate length periods, initially EDF is better than RM, but then EDF's run-time overhead increases to the point that RM becomes superior. For $n = 15$, CSD-4 has 25% less overhead than EDF, while for $n = 40$, CSD-4 has 50% lower overhead than RM (which in turn has lower overhead than EDF for this large n).

Figure 4.5 shows similar results. Task periods range from 1.67ms–333ms, and these short periods allow RM to quickly overtake EDF. Nevertheless, CSD continues to be superior to both.

4.4.2 CSD-x

Figures 4.3–4.5 also show a comparison between three varieties of CSD. They show that even though a significant performance improvement is seen from CSD-2 to CSD-3

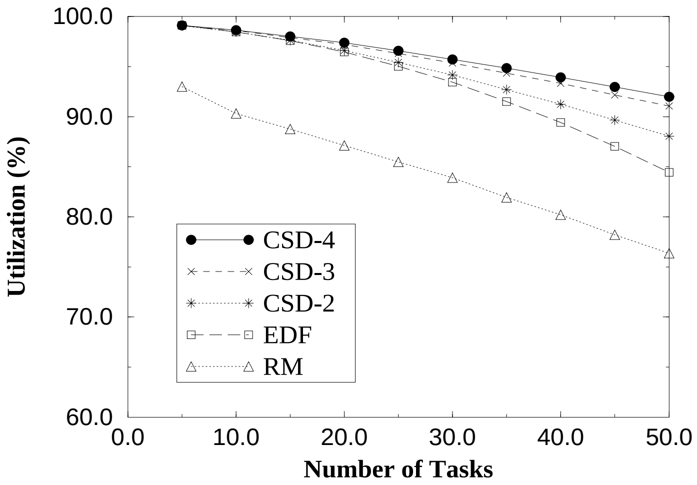

Figure 4.3: Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 1.

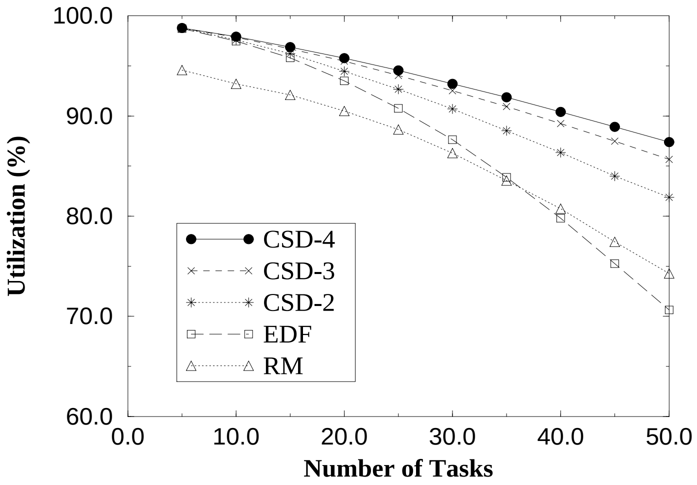

Figure 4.4: Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 2.

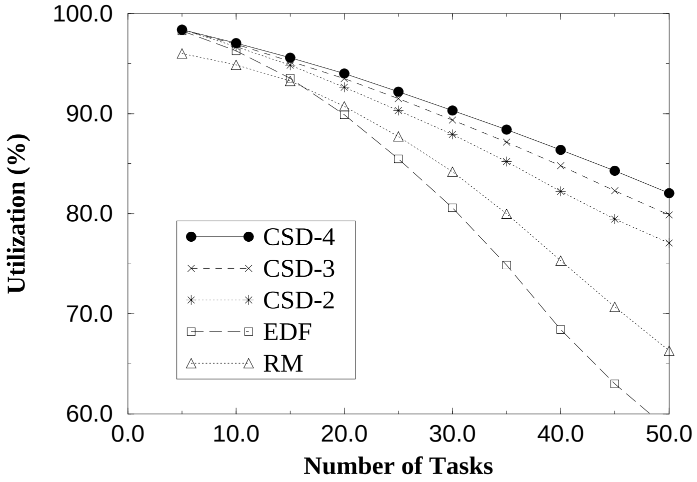

Figure 4.5: Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 3.

(specially for large n), only a minimal improvement is observed from CSD-3 to CSD-4. This is because even though the run-time overhead continues to decrease, the increase in schedulability overhead almost nullifies the reduction in run-time overhead.

CSD-4 could be expected to give significantly better breakdown utilization than CSD-3 only if workloads can be easily partitioned into four queues without increasing schedulability overhead, but this is rarely the case. DP1 tasks have statically higher priority than DP2 tasks, DP2 tasks have higher priority than DP3 tasks, and so on. As the number of queues increases, the schedulability overhead starts increasing from that of EDF to that of RM. This is why we would expect that as x increases, performance of CSD-x will quickly reach a maximum and then start decreasing because of reduced schedulability and increased overhead of managing x queues (which increases by 0.55μs per queue). Eventually, as x approaches n, performance of CSD-x will degrade to that of RM. □

The results presented here confirm the superiority of the CSD scheduling framework as compared to EDF and RM. The results show that even though CSD-2 suffers from high run-time overhead for large n, CSD-3 overcomes this problem without any significant increase in schedulability overhead. This way, CSD-3 delivers consistently good performance over a wide range of task workload characteristics. Increasing the number of queues gives some further improvement in performance, but the schedulability overhead starts increasing rapidly so that using more than three queues yields only a minimal improvement in performance.

4.5 Related Work

Using multiple scheduling queues is not a new idea. The ERCOS task scheduler [92] uses separate queues for preemptive and non-preemptive tasks. Multiple queues are used in network scheduling to combine different types of traffic on the same link in a switched network [26, 27, 54]. But in all these cases, multiple queues are used to share a single resource (CPU or network link) between tasks/messages with different quality of service (QoS) requirements. What is novel about CSD is the use of multiple queues to improve performance. We allocate tasks to different queues in a manner that reduces the task scheduling overhead, giving better performance than conventional schedulers like EDF and RM. As such, our scheme is orthogonal to scheduling schemes which handle varying QoS requirements. The two can be combined by using CSD to schedule one or more queues of a QoS scheduler.

The *rotating-priority-queues* (RPQ) scheme [73, 74] was also proposed for network scheduling and like CSD, it too attempts to find a middle ground between EDF and static priority schedulers by using multiple queues. Packets within a queue use FIFO ordering but the relative priorities between queues rotate in a fixed way. The motivation behind RPQ was to find an efficient hardware implementation for network packet scheduling. RPQ achieves this by coarse-grained deadline quantization and using FIFO ordering for all packets with the same quantized deadline. This lowers hardware costs but can degrade schedulability significantly.

Liu and Layland in their seminal paper [76] also proposed combining EDF and RM. Their motivation was to exploit fixed CPU interrupt priorities to schedule short-period tasks while using a software EDF scheduler for long-period tasks. High-priority tasks get scheduled by fixed-priority scheduling (using hardware mechanisms) while low-priority tasks are scheduled using software deadline-driven scheduling. In today's complex systems, tying the scheduler to hardware interrupt priorities is not feasible. In fact, many modern processors (such as various versions of PowerPC) do not even support multiple hardware priority levels. In such circumstances, the proposal of Liu and Layland is no longer applicable. When all scheduling is done in software, it makes much more sense to use dynamic scheduling for short-period tasks and fixed-priority scheduling for the rest of the workload as is done by CSD.

The *start-time fair queuing* (SFQ) algorithm [34] was proposed as a framework to enable use to different schedulers for different classes of applications. Conceivably, SFQ can be used to combine EDF and RM. However, origins of SFQ are also in network scheduling where it is feasible to use start time as a basis for fair queuing. This is because packets belonging to a certain stream (i.e., flow of packets) can accumulate which allows the scheduler greater flexibility in proportioning the network link bandwidth. For example, if packets for a certain stream are expected to arrive once every 2ms then they will continue to arrive regardless of whether earlier packets have been forwarded or not. The situation is completely different in CPU scheduling. If one invocation of a periodic task does not complete, the next invocation will not be released. So, if a short period task is combined with several long period tasks, SFQ will dispatch all the first invocations of the long period tasks before second and higher invocations of the long period task because the former have earlier start times. This means that the second invocation of the short period task will be considerably delayed (and no other invocations of this task will "accumulate", preventing SFQ from "catching up" later on as it can in network scheduling). This makes SFQ undesirable for scheduling tight-deadline tasks.

4.6 Conclusion

One of the most important services provided by the RTOS is real-time task scheduling. Schedulers such as RM and EDF can incur overheads of 5–15% of CPU time, leaving only 85–95% of the CPU for executing user tasks, yet little attention has been paid towards studying the sources of these overheads and even less attention towards devising schemes to reduce these overheads. In this chapter, we presented the CSD scheduler which creates a balance between static and dynamic scheduling to deliver greater breakdown utilization through a reduction in scheduling overhead of as much as 40% compared to EDF and RM. CSD is a general scheduling framework which allows the scheduler to be configured according to the workload to deliver the best possible performance.

Future work includes exploring issues related to the optimal configuration of CSD (optimal number of queues and optimal number of tasks per queue) for a given workload. Another interesting avenue of research is studying the possibility of using the low-overhead RM policy to schedule queues other than just the last queue. This can be beneficial if the tasks in a queue have only a minimal difference between RM-schedulability and EDF-schedulability. Then, using RM (instead of EDF) for such queues will reduce run-time overhead without significantly affecting schedulability overhead.

CHAPTER 5

EFFICIENT SEMAPHORES

In object-oriented programming in multi-threaded systems, updates to the state variables of objects (by the methods of the object) have to be protected through semaphores to ensure mutual exclusion. Semaphore operations are invoked each time an object is accessed, and this represents significant run-time overhead. This is of special concern in cost-conscious embedded systems. Object-oriented programming can be feasible in such applications only if the OS provides efficient, low-overhead semaphores. We present a new semaphore implementation scheme which saves one context switch per semaphore lock operation in most circumstances. Of course, an efficient semaphore scheme is useful not only for OO programming but for any application requiring synchronization between multiple threads of execution.

5.1 Introduction

In this chapter, we focus on OS support for object-oriented (OO) programming in embedded systems. The advent of Java and increasing use of C++ has made OO programming important for embedded systems. OO design gives benefits such as reduced software design time and software re-use [83]. But with these benefits comes the extra cost of ensuring mutual exclusion when an object's internal state is updated. Semaphores[1] [18, 36] are typically used to provide this mutual exclusion. Because semaphore system calls are invoked every time an execution thread enters or exits an object, it becomes essential that the RTOS provide efficient, low-overhead semaphores; otherwise, OO design will not be feasible for embedded applications because of high costs.

[1]The optimization scheme presented in this chapter applies equally well to both semaphores and mutexes. However, for simplicity, we concern ourselves only with semaphores in this chapter.

Most research in the area of reducing task synchronization overhead has focused on multiprocessors [81, 116]. But our target architectures are either uniprocessor (as in home appliances) or very loosely-coupled distributed systems (as in automotive applications). Even with the latter, threads typically do not need to access remote objects, so our concern is only with improving task synchronization performance for a single processor. Previous work in this area has focused on either relaxing the semaphore semantics to get better performance [111], coming up with new semantics and new synchronization policies [114], or putting restrictions on the application programmer to disallow certain operations (such as making blocking system calls) while holding a semaphore [89]. The problem with this approach is that these new/modified semantics may be suitable for some particular applications but usually they do not have wide applicability.

We took the approach of providing full semaphore semantics (with priority inheritance [101]), but optimizing the implementation of these semaphores by exploiting certain features of embedded applications [126]. As a result, our semaphore scheme has wide applicability within the domain of embedded applications, while significantly improving performance over standard implementation methods for semaphores.

In the next section, we give a brief overview of OO programming as it pertains to embedded real-time systems, focusing on OS support needed for OO programming. In Section 5.3, we describe our new implementation scheme. Section 5.4 discusses some limitations of the scheme and ways to overcome these limitations so that our scheme can be used in almost all embedded applications. Section 5.5 evaluates the performance of our new scheme, and we conclude with Section 5.6.

5.2 Objects and Semaphores in Embedded Real-Time Systems

An object is a collection of private state information (or data) and a set of methods which manipulate the data. Objects are ideal for representing real-world entities: the object's internal data represents the physical state of the entity (such as temperature, pressure, position, RPM, etc.) and the methods allow the state to be read or modified. These notions of encapsulation and modularity greatly help the software design process because various system components such as sensors, actuators, and controllers can be modeled by objects. Then, under the OO paradigm, real-time software is just a collection of threads of execution, each invoking various methods of various objects [48].

Conceptually, this OO paradigm is very appealing and gives benefits such as reduced software design time and software re-use. But practically speaking, these benefits come at a cost. The methods of an object must synchronize their access to the object's data to ensure mutual exclusion. Because object invocations occur very frequently, it is essential that any scheme used to achieve this synchronization must be both *memory-efficient* as well as *time-efficient*; otherwise, OO design will be infeasible for embedded systems due to high costs.

5.2.1 Active and Passive Object Models

There are two fundamentally different ways for objects and execution threads to interact with each other and this has some bearing on the type of synchronization scheme used to ensure mutual exclusion.

Under the active object model [11], one or more server threads are permanently bound to an object. When a client thread invokes a method, a server thread executes the method on behalf of the client.

With the passive object model [11], objects do not have threads of their own. To invoke a method, a thread will enter the object, execute the method, and then exit the object.

From the point of view of synchronization, the active object model has an advantage if only one thread is assigned per object. Since only one thread is in the object at any time, there is no need to worry about mutual exclusion. But the active object model has several disadvantages. First of all, having a thread per object means that there will be a large number of threads in the system (anywhere from several tens to more than a hundred depending on the application). Each thread needs its own stack, thread control block, etc., which makes the active object model very memory-inefficient. Moreover, each object invocation requires a context switch from the client thread to the server thread, so this model is time-inefficient as well.

With the passive object model, multiple threads can be inside the same object at one time, so they must synchronize their activities. Semaphores [18,36] are commonly used for this purpose (e.g., to provide the monitor construct [40]). Even though locking based on semaphores incurs time overhead, it is decidedly much more memory-efficient than the active object model.

5.2.2 OO Design Under EMERALDS

For the above stated reasons, we advocate the passive object model for embedded software design. Because a semaphore system call is made every time an object's method is invoked, semaphore operations (`sem_lock()` and `sem_unlock()` calls under EMERALDS, used to lock and unlock semaphores, respectively) become some of the most heavily used OS primitives when OO design is used. This motivated us to investigate new and efficient schemes for implementing semaphore locking in EMERALDS as described next.

5.3 An Efficient Semaphore Implementation Scheme

The first step in designing efficient semaphores is to look at the way semaphores are typically implemented in various systems, identify distinct steps involved in locking/unlocking semaphores, and try to eliminate or optimize those steps which incur the greatest overhead. To do these optimizations, we will use characteristics common to embedded applications.

5.3.1 Standard Semaphore Implementation

The standard procedure to lock a semaphore can be summarized as follows:

```
if (sem locked) {
   do priority inheritance;
   add caller thread to wait queue;
   block;   /* wait for sem to be released */
}
lock sem;
```

Priority inheritance [101] is needed in real-time systems to avoid unbounded priority inversion [114]. If a high-priority thread T_h calls `sem_lock()` on a semaphore already locked by a low-priority thread T_l, the latter's priority is temporarily increased to that of the former. Without priority inheritance, a medium priority thread T_m can get control of the CPU by preempting T_l while T_h remains blocked on the semaphore, thus causing priority inversion. With priority inheritance, T_l will keep on running until it unlocks the semaphore. At that point, its priority will go back to its original value, but now T_h will be unblocked and it can continue execution.

First of all, notice that if the semaphore is free when `sem_lock()` is called, then the semaphore lock operation has very little overhead[2]. In fact, for this case, only one counter

[2]This is especially true in EMERALDS where system call overhead is comparable to subroutine call overhead even with full memory protection between processes [124].

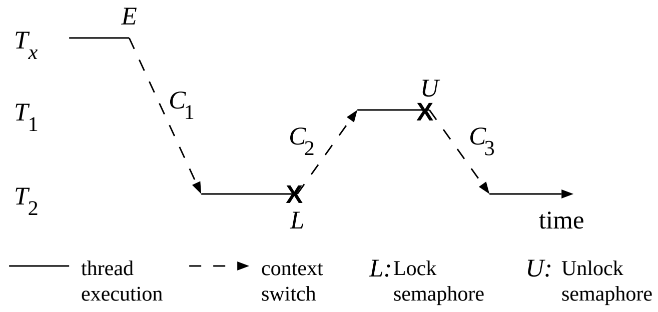

Figure 5.1: A typical scenario showing thread T_2 attempting to lock a semaphore already held by thread T_1. T_x is an unrelated thread which was executing while T_2 was blocked. Conceptually, T_x can be T_1.

> `unblock` T_2
> `context switch` C_1 `(`T_x `to` T_2`)`

(T_2 executes and calls `sem_lock()`)

> `do priority inheritance (`T_2 `to` T_1`)`
> `block` T_2
> `context switch` C_2 `(`T_2 `to` T_1`)`

(T_1 executes and calls `sem_unlock()`)

> `undo priority inheritance of` T_1
> `unblock` T_2
> `context switch` C_3 `(`T_1 `to` T_2`)`

Figure 5.2: Operations involved in locking a semaphore for the scenario shown in Figure 5.1

has to be incremented and some other variables updated.

In real-time systems, we are interested in worst-case overheads, and for semaphores, this occurs when the semaphore is already locked by thread T_1 when some thread T_2 invokes the `sem_lock()` call. Figure 5.1 shows a typical scenario for this situation. Thread T_2 wakes up (after completing some unrelated blocking system call) and then calls `sem_lock()`. This results in priority inheritance and a context switch to T_1, the current lock holder. After T_1 releases the semaphore, its priority returns to its original value and a context switch occurs to T_2. These steps are outlined in Figure 5.2.

For tasks scheduled by EDF, the context switches are responsible for the largest overhead because this is where Δt_s is incurred (which takes $O(r)$ time, see Chapter 4), whereas the remaining operations take only $O(1)$ time. For this reason, we will focus our optimization efforts on eliminating one or more context switches and this should result in good performance improvement for DP tasks.

For FP tasks, context switches incur a fixed (although significant) overhead, so eliminating one context switch is not as beneficial for FP tasks as it is for DP tasks. However, each of the two priority inheritance (PI) steps take $O(n-r)$ time because the task must be removed from the FP queue and then re-inserted in sorted order according to its new priority. All the remaining operations take $O(1)$ time, even the block operation because the PI operation preceding the block resets `highestP` so that the block operation doesn't have to. This is why, for FP tasks, we focus our optimization efforts on the PI operations.

5.3.2 Semaphore Implementation in EMERALDS

Going back to Figure 5.2, we want to eliminate context switch C_2. We also want to optimize the two PI steps. First, we deal with C_2 which occurs when T_2 in unblocked after some blocking system call (T_2 had made this call to wait for some event E such as a message arrival or timer expiry). T_2 then executes and calls `sem_lock()`, only to block again because the semaphore is locked by T_1.

The idea is that when event E occurs, instead of letting T_2 run, let T_1 execute. T_1 will go on to release the semaphore and T_2 can be activated at this point, saving C_2 (Figure 5.3). This is implemented as follows. As part of the blocking call just preceding `sem_lock()`, we instrument the code (using a code parser described later) to add an extra parameter which indicates which semaphore T_2 intends to lock (semaphore S in this case). When event E occurs and T_2 is to be unblocked, the OS checks if S is available or not. If S is unavailable, then priority inheritance from T_2 to the current lock holder T_1 occurs right here. T_2 is added to the waiting queue for S and it remains blocked. As a result, the scheduler picks T_1 to execute — which eventually releases S — and T_2 is unblocked as part of this `sem_unlock()` call by T_1. Comparing Figure 5.3 to Figure 5.1, we see that context switch C_2 is eliminated. The semaphore lock/unlock pair of operations now incur only one context switch instead of two, resulting in considerable savings in execution time overhead for DP tasks (see Section 5.5 for performance results).

For FP tasks, we want to optimize the two PI steps, each of which takes $O(n-r)$ time (Chapter 4). The first PI step (T_1 inherits T_2's priority) is easily optimized by using the observation that, according to T_1's new priority, its position in the FP queue should be just ahead of T_2's position. So, instead of parsing the FP queue to find the correct position to insert T_1, we insert T_1 directly ahead of T_2 without parsing the queue which reduces overhead to $O(1)$.

We want to reduce the overhead of the second PI step to $O(1)$ as well. In this step, T_1

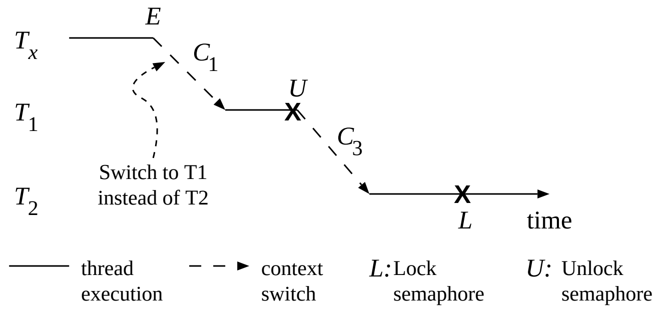

Figure 5.3: The new semaphore implementation scheme. Context switch C_2 is eliminated.

returns to its original priority. We want to do this without having to parse the entire queue. One incorrect solution is to remember T_1's neighbors from its original position in the queue in an attempt to return T_1 to that position by inserting it between these neighbors. But if these neighbors themselves undergo priority inheritance, their position in the queue will change and the scheme will not work.

The solution used in EMERALDS is to switch the positions of T_1 and T_2 in the queue as part of the first PI operation when T_1 inherits T_2's priority. This puts T_1 in the correct position according to its new priority while T_2 acts as a "place-holder" for T_1 to remember T_1's original position in the queue. Then the question is: is it safe to put T_2 in a position lower than what is dictated by its priority? The answer is yes. As long as T_2 stays blocked, it can be in any position in the queue. T_2 unblocks only when T_1 releases the semaphore, and at that time, we switch the positions of T_1 and T_2 again, restoring each to their original priorities. With this scheme, both PI operations take $O(1)$ time.

One complication arises if T_1 first inherits T_2's priority, then a third thread T_3 attempts to lock this semaphore and T_1 inherits T_3's priority. For this case, T_3 becomes T_1's place-holder and T_2 just goes back to its original position. This involves one extra step compared to the simple case described initially but the overhead is still $O(1)$.

Note that these optimizations on the PI operations were possible because our scheduler implementation keeps both ready and blocked tasks in the same queue. Had the FP queue contained only ready tasks, we could not have kept the place-holder TCB in the queue.

Code Parser:

In EMERALDS, all blocking calls take an extra parameter which is the identifier of the semaphore to be locked by the upcoming `sem_lock()` call. This parameter is set to -1 if the next blocking call is not `sem_lock()`. For embedded systems, it is possible to write a parser which examines the application code and automatically inserts the correct semaphore identifier into the argument list of blocking calls just preceding `sem_lock()` calls. Parser design issues are discussed further in Section 5.4.

Schedulability Analysis for the New Scheme:

From the viewpoint of schedulability analysis, there can be two concerns regarding the new semaphore scheme (refer back to Figure 5.3):

1. What if thread T_2 does not block on the call preceding `sem_lock()`? This can happen if event E has already occurred when the call is made.

2. Is it safe to delay execution of T_2 even though it may have higher priority than T_1 (by doing priority inheritance earlier than would occur otherwise)?

Regarding the first concern, if T_2 does not block on the call preceding `sem_lock()`, then a context switch has already been saved. For such a situation, T_2 will continue to execute till it reaches `sem_lock()` and a context switch will occur here. What our scheme really provides is that a context switch will be saved either on the `sem_lock()` call or on the preceding blocking call. Where the savings actually occur at run-time do not really matter for calculation of worst-case execution times for schedulability analysis.

For the second concern, the answer is that yes, it is safe to let T_1 execute earlier than it would otherwise. The concern here is that T_2 may miss its deadline. But this cannot happen because under all circumstances, T_2 must wait for T_1 to release the semaphore before T_2 can complete. So from the schedulability analysis point of view, all that really happens is that chunks of execution time are swapped between T_1 and T_2 without affecting the completion time of T_2. Another similar concern is that after event E, T_2 may have to produce an output or send a message/signal to another thread (call it T_3). Delaying T_2 may cause T_3 to miss its deadline. The answer to all such scenarios is that as just discussed, T_2 completes by its deadline (even though it may be delayed). As long as T_2 completes by its deadline, no other thread that depends on T_2 will miss its deadline, so schedulability of the task workload is not adversely affected.

5.4 Applicability of the New Scheme

There can be three circumstances under which our proposed semaphore scheme may not work:

1. The code parser is unable to identify which semaphore is to be locked next due to con-

```
for (;;) {
    read sensor 1;
    read sensor 2;
    ...
    read sensor x;
    update actuator 1;
    update actuator 2;
    ...
    update actuator y;
    block till timer expiry
              or event occurrence;
}
```

Figure 5.4: A typical sensor-controller-actuator loop commonly found in embedded control applications

ditional constructs such as loops with a variable number of iterations or `if-then-else` statements.

2. The blocking call preceding an `sem_lock()` is another `sem_lock()` so that only one context switch is saved between these two calls.

3. The lock holder T_1 (Figure 5.3) blocks after event E but before releasing the semaphore. Then with standard semaphores, T_2 will be able to execute, but under our scheme it cannot which may lead to T_2 missing its deadline.

In the rest of this section, we discuss how often (if at all) these scenarios can occur in embedded real-time systems, which specific forms they can occur in, and how these problems can be resolved.

5.4.1 Code Parser Issues

Most threads in embedded systems execute sensor-controller-actuator loops as shown in Figure 5.4 (for IAs, the "sensor" can be a network device and the "actuator" can be an audio or video output device). Each device (sensor or actuator) is represented by an object protected by its own semaphore. Each device may be a real sensor/actuator or a logical one representing several devices being controlled as one group.

Note that the same devices are accessed each time the loop executes. The order in which semaphores are locked is fixed, so there is no ambiguity for the code parser. At run-time, the method which gets invoked on an object may depend on the input data:

```
if (sensorReading > A) valve.open();
else                   valve.close();
```

but this does not change the order in which semaphores are locked because all methods of an object are protected by the same semaphore. In other words, most embedded applications are structured as in Figure 5.4, and for such a structure, the parser can easily determine which semaphore is to be locked after a given blocking call.

In case a blocking call occurs inside a loop followed by `sem_lock()` outside the loop, the argument to be passed for the semaphore identifier is calculated conditionally as follows:

```
while (cond) {
   ...
   if (cond)
      sem = -1;
   else
      sem = S;
   some_blocking_call(..., sem);
   ...
}
...
sem_lock(S);
```

This way, -1 is passed as the parameter for all but the last iteration of the loop. Again, this code can be automatically inserted by the code parser without the application programmer having to make any manual modifications to the code. Note that this scheme works as long as the condition `cond` does not depend on the blocking call or code after the call. This is true for loops which execute for a fixed number of iterations which is the most common case in embedded control systems. One example is code which steps a stepper motor x number of times. Value of x may depend on sensor readings, but it stays fixed while the loop executes.

Regarding loops with a variable number of iterations, our experience shows that such loops typically do not contain blocking calls in embedded real-time systems. A variable-iteration loop is used to wait for a condition to come true (such as a spin lock), but that is what blocking calls do as well (wait for a condition). The two may be combined if the result of the blocking call is uncertain (such as for condition variables with Mesa semantics used in general-purpose computing), but such a situation rarely occurs in embedded real-time systems.

5.4.2 Consecutive `sem_lock()` Calls

Going back to Figure 5.4, the bodies of the methods invoked by the thread may contain blocking calls, especially condition variable and message-passing calls. In these calls, the parser will insert the identifier of the upcoming `sem_lock()`. But if such calls are not present, then two or more `sem_lock()` calls can occur with no other blocking call in between them. Then, only one context switch will be saved per pair of `sem_lock()` calls. This leads to an interesting avenue for future research. Our scheme can be generalized so that the blocking call at the end of the control loop will not unblock until *all* the semaphores needed by the thread for execution become available. In other words:

```
for (;;) {
   obj_1.method   // protected by sem S1
   obj_2.method   // protected by sem S2
   ...
   obj_n.method   // protected by sem Sn
   block(..., S1, S2, ..., Sn);
}
```

This is somewhat similar to the Spring kernel's notion of reserving all resources a task needs before letting the task execute [109], but with an important difference: the Spring kernel executes tasks non-preemptively while under our proposal, threads execute preemptively. This allows higher priority threads to preempt a given thread (giving good schedulable utilization) while reducing the number of context switches seen by the thread to wait for resources (giving shorter execution times). However, advance reservation of all semaphores will increase scheduler complexity and may also adversely affect task schedulability. Impact of these issues on performance must be studied to determine the viability of this extension.

5.4.3 Blocking by the Lock Holder Thread

Going back to Figure 5.3, suppose the lock holder T_1 blocks after event E but before releasing the semaphore. With standard semaphores, T_2 will then be able to execute (at least, till it reaches `sem_lock()`), but under our scheme, T_2 stays blocked. This gives rise to the concern that with this new semaphore scheme, T_2 may miss its deadline.

In Figure 5.3, T_1 had priority less than that of T_2 (call this case A). A different problem arises if T_1 has higher priority than T_2 (call it case B). Suppose semaphore S is free when event E occurs. Then T_2 will become unblocked and it will start executing (Figure 5.5). But before T_2 can call `sem_lock()`, T_1 wakes up, preempts T_2, locks S, then blocks for some

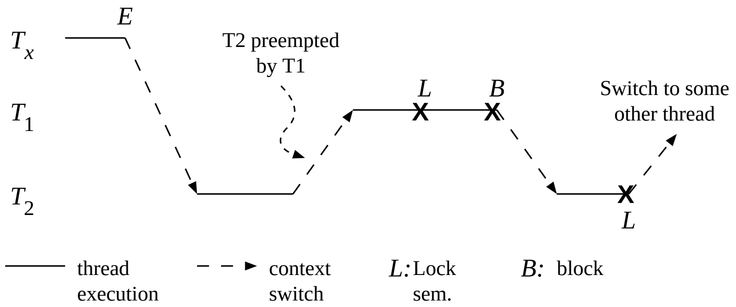

Figure 5.5: If a higher priority thread T_1 preempts T_2, locks the semaphore, and blocks, then T_2 incurs the full overhead of `sem_lock()` and a context switch is not saved.

event. T_2 resumes, calls `sem_lock()`, and blocks because S is unavailable. The context switch is not saved and no benefit comes out of our semaphore scheme.

All these problems occur when a thread blocks while holding a semaphore. To resolve these problems, we first make a small modification to our semaphore scheme to change the problem in case B to be the same as the problem in case A. This leaves us with only one problem to address. Then, by looking at the larger picture and considering threads other than just T_1 and T_2, we can show that this problem is easily circumvented and our semaphore scheme works for all blocking situations that occur in practice as discussed next.

Modification to the Semaphore Scheme:

For the situation shown in Figure 5.5, we want to somehow block T_2 when the higher-priority thread T_1 locks S, and unblock T_2 when T_1 releases S. This will prevent T_2 from executing while S is locked, which makes this the same as the situation in case A.

Recall that when event E occurs (Figure 5.5), the OS first checks if S is available or not before unblocking T_2. Now, let us extend the scheme so that the OS adds T_2 to a special queue associated with S. This queue holds the threads which have completed their blocking call just preceding `sem_lock()` but have not called `sem_lock()` yet.

Thread T_1 will also get added to this queue as part of its blocking call just preceding `sem_lock()`. When T_1 calls `sem_lock()`, the OS first removes T_1 from this queue, then puts all threads remaining in the queue in a blocked state. Then, when T_1 calls `sem_unlock()`, the OS unblocks all threads in the queue. This way, T_2 is prevented from executing while S is locked which results in the same behavior as in case A. Also, if done properly, addition and removal of threads from this queue incurs very little overhead (about 5–7 μs on a 25 MHz MC 68040 without caches and just 1–2 μs with caches).

With this modification, the only remaining concern (for both cases A and B) is: if

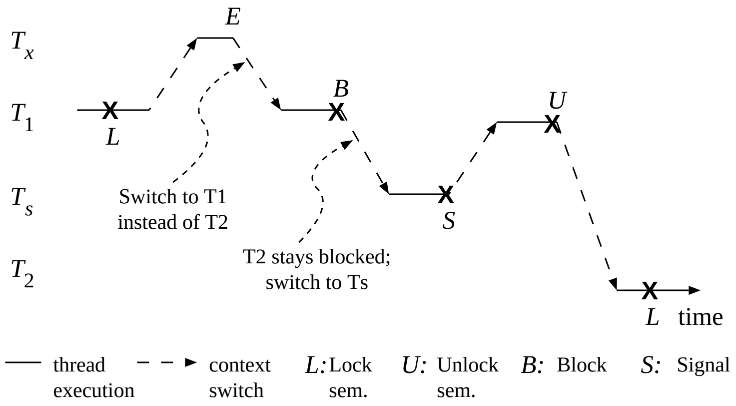

Figure 5.6: Situation when the lock holder T_1 blocks for a signal from another thread T_s.

execution of T_2 is delayed like this while other threads (of possibly lower priority) execute, then T_2 may miss its deadline. This concern is addressed next.

Applicability under Various Blocking Situations:

There can be two types of blocking:

- Wait for an *internal* event, i.e., wait for a signal from another thread after it reaches a certain point.
- Wait for an *external* event from the environment. This event can be periodic or aperiodic.

The first type of blocking is used by threads to synchronize with each other and the second type is used to interact with the environment.

Blocking for Internal Events: The typical scenario for this type of blocking is for thread T_1 to enter an object (and lock semaphore S) then block waiting for a signal from another thread T_s. Meanwhile, T_2 stays blocked (Figure 5.6). The question is: is it safe to delay T_2 like this even if T_s is lower in priority than T_2? The answer is yes, because T_2 cannot lock S till T_1 releases it, and T_1 will not release it till it receives the signal from T_s, so even though T_s may be lower in priority than T_2, it is safe to let T_s execute earlier. This leads to T_1 releasing S earlier than it would otherwise which leaves enough time for T_2 to complete by its deadline.

Blocking for External Events: External events can be either periodic or aperiodic. For periodic events, polling is usually used to interact with the environment and blocking does not occur. A common example is a periodic sensor-controller-actuator loop where sensors are read and actuator commands are updated periodically and no blocking calls are involved. One common exception is to block on a timer (usually, to wait for the current period to end), but this blocking call occurs at the end of the main loop of execution of the thread and is not inside any object and no semaphores are held by the thread when this call is made.

Blocking calls are used to wait for aperiodic events, but it does not make sense to have such calls inside an object. There is always a possibility that an aperiodic event may not occur for a long time. If a thread blocks waiting for such an event while inside an object, it may keep that object locked forever, preventing other threads from making progress. So the usual practice is to not have any semaphores locked when blocking for an aperiodic event.

In short, dealing with external events (whether periodic or aperiodic) does not affect the applicability of our semaphore scheme under the commonly-established ways of handling external events. But in case some application does require blocking for external events while inside an object, our semaphore scheme can be turned off by specifying -1 as the semaphore identifier in the blocking call just preceding `sem_lock()`. This will cause EMERALDS' semaphores to behave just like standard implementation semaphores, but we do not believe this will be needed very often, if at all.

5.5 Performance Evaluation

To measure the improvement in performance resulting from our new semaphore scheme, we implemented it under EMERALDS and measured performance on a 25 MHz Motorola 68040 processor [87].

When a thread enters an object, it first acquires the semaphore protecting the object, and when it exits the object, it releases the semaphore. The cumulative time spent in these two operations represents the overhead associated with synchronizing thread access to objects. To determine by how much this overhead is reduced when our scheme is used, we measured the time for the acquire/release pair of operations for both standard semaphores and our new scheme and then compared the two results. In the following, we first describe our evaluation procedure, then present the results.

5.5.1 The Test Procedure

We want to measure the worst-case overhead for acquire/release because this is what is used in schedulability analysis. The worst case occurs if

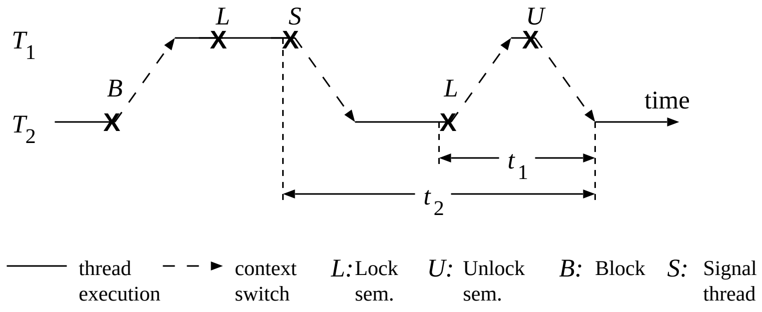

Figure 5.7: Test procedure for standard semaphores. Interval t_1 is the overhead for acquire/release operations.

- the semaphore is already locked when `sem_lock()` is called, and
- priority inheritance occurs.

To get this behavior, we use two threads in our tests, T_1 and T_2, with T_2 having higher priority. For the standard semaphore implementation, the test proceeds as shown in Figure 5.7. T_2 executes first and blocks waiting for a signal from T_1. T_1 executes, locks semaphore S, and signals T_2 which is unblocked, goes on to execute `sem_lock()`, and priority inheritance occurs. Thread T_1 then releases S, its priority goes back to its original value, and a context switch occurs back to T_2. We measure interval t_1 which is the time for an acquire plus a release and includes relevant context switches.

We repeated this test with the new semaphore scheme. Figure 5.8 shows the new sequence of events. In this case, priority inheritance is done by the OS when T_1 signals T_2, so T_1 continues after the signal and unlocks S. T_1's priority goes back to its original value, T_2 is unblocked, and it goes on to lock S without needing any more context switches. Then the difference $t_2 - t_3$ (Figures 5.7 and 5.8) represents the improvement due to the new scheme and $t_1 - (t_2 - t_3)$ is the overhead for acquire/release under the new scheme. Note that we cannot directly measure the acquire/release overhead for the new scheme because priority inheritance occurs well before the rest of the acquire operation.

5.5.2 Experimental Results

Our semaphore scheme eliminates one context switch and optimizes the priority inheritance mechanism for FP tasks, so the performance of our scheme depends on whether the relevant tasks are in the DP or FP queue, as well as on the number of tasks in the queue. Figure 5.9 shows the semaphore overheads for tasks in the DP queue as the number of tasks in the queue are varied from 3 to 30. Since the context switch overhead is a linear function

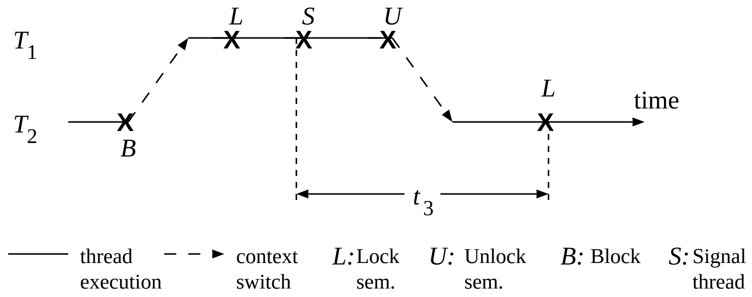

Figure 5.8: Test procedure for the new semaphore scheme.

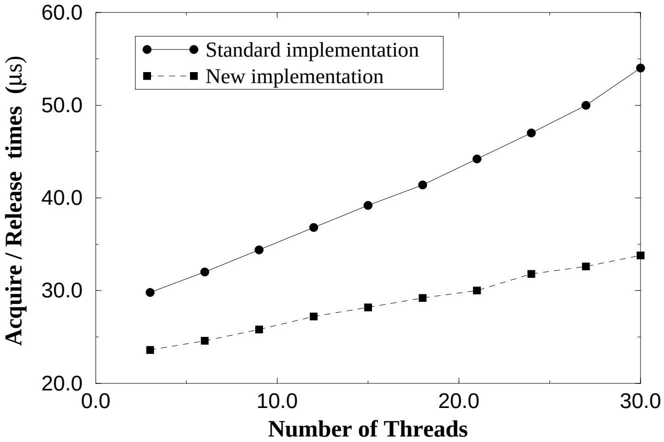

Figure 5.9: Worst-case performance measurements for DP tasks. The overhead for the standard implementation increases twice as rapidly as for the new scheme.

of the number of tasks in the DP queue (because of Δt_s), the acquire/release times increase linearly with the queue length. But the standard implementation's overhead involves two context switches while our new scheme incurs only one, so the measurements for the standard scheme have a slope twice that of our new scheme. For a typical DP queue length of 15, our scheme gives savings of 11μs over the standard implementation (a 28% improvement), and these savings grow even larger as the DP queue's length increases. Figure 5.11 shows the percentage improvement with varying number of threads.

For the FP queue, the standard implementation has a linearly increasing overhead while with the new implementation, the overhead is constant (because priority inheritance takes $O(1)$ time). Also, one context switch is eliminated. As a result, the acquire/release overhead stays constant at 29.4μs. For an FP queue length of 15, this is an improvement of 10.4μs or 26% over the standard implementation.

In general, our scheme gives performance improvements of 20–30%, depending on whether the tasks involved in locking and unlocking the semaphore are in the DP or FP queue and

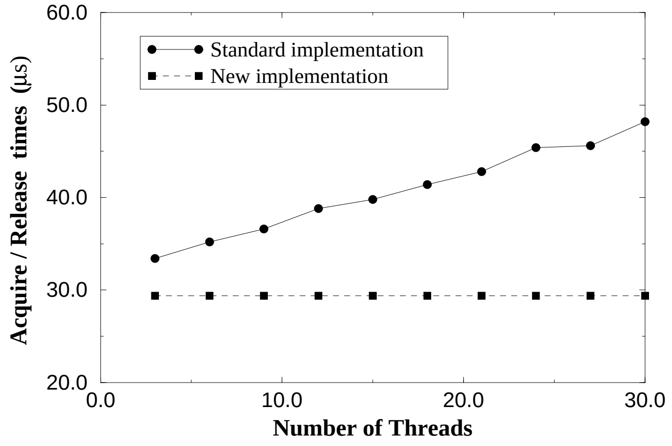

Figure 5.10: Worst-case performance measurements for FP tasks. The overhead for the standard implementation increases linearly while new scheme has a constant overhead.

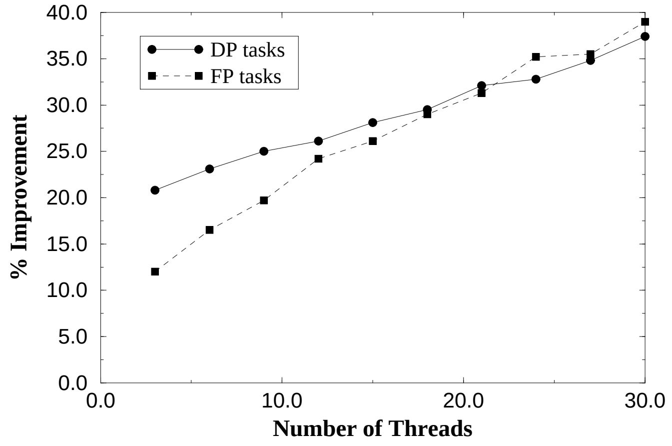

Figure 5.11: Percent improvement in performance due to our new semaphore scheme.

the length of the queue.

5.6 Conclusion

Embedded application programmers generally tend to avoid object-oriented programming, one reason being the high overhead associated with synchronizing thread access to objects. Semaphores must be used to ensure mutual exclusion when updating the state variables of objects, and this usually means a large enough overhead to make object-oriented programming infeasible for cost-conscious embedded applications.

In this chapter, we presented a new semaphore implementation scheme which saves one context switch per semaphore acquire/release pair of operations (for most scenarios found in embedded applications) and improves performance by 20–30%. We used the fact that in embedded applications, the sequence in which semaphores are to be locked can be identified at compile time. Then, during run-time, we use this known sequence to do ahead-of-time checks on the status of semaphores (whether they are available or not). If a semaphore is unavailable, we delay the execution of threads until the semaphore is released. This way, the semaphores are always available when threads actually make the `sem_lock()` system call and the call does not block, saving one context switch.

Future work includes studying the advantages and disadvantages of extending our scheme so that instead of looking ahead only to the next `sem_lock()` call, the scheduler will consider *all* the semaphores a thread may need to execute so that all resource conflict-related context switches are eliminated. Also, in this chapter we focused only on improving the semaphore lock operation. In the future, we plan to investigate optimizations related to the release operation to get further improvements in synchronization overheads.

CHAPTER 6

END-HOST PROTOCOL PROCESSING ARCHITECTURE

Information appliances (IAs) [64, 71] are single-user devices with Internet connectivity, used for specialized communication and information retrieval purposes. Currently, IAs exist as webTVs, smart cellular phones with e-mail and web browsing, PDAs, and web phones. The future of IAs is already evident in new devices such as web video phones which use the Internet for audio/video communication. With annual production volume of IAs expected to reach 48 million units by year 2001 [5], IAs are becoming an important class of embedded devices.

IAs differ from other embedded devices (such as automotive controllers) in that they communicate directly over the Internet. This means that IAs must run a full communication protocol stack. Real-time audio and video communication over the Internet is an integral part of many IAs which means that despite slow hardware, the communication subsystem within the OS must be able to efficiently handle heavy network traffic. In this chapter, we present optimizations for reducing receive-side network protocol processing overhead thus enabling efficient handling of real-time audio and video messages. (We focus on receive-side overhead since it usually exceeds send-side overhead.) In our scheme, I-cache miss overheads are minimized by safely bypassing multiple protocol layers, benefiting *short* messages such as live audio. Moreover, message data needs to be copied only once (without any hardware support from the network adapter or any restrictions on the network API) which benefits *long* messages such as video and streaming data.

The next section discusses the audio/video communication requirements of IAs. Section 6.2 gives an overview of protocol processing overheads. Section 6.3 presents our schemes for reducing these overheads, and these optimizations are then evaluated in Section 6.4. Section 6.5 discusses related work and the chapter concludes with Section 6.6.

6.1 Audio/Video Communication in IAs

IAs differ from traditional PCs and workstations in two key aspects. First, IAs are specialized devices which perform specific functions and are not meant for general-purpose computing. Second, IAs use simple, low-cost hardware to keep production costs low. For example, the current generation of *personal information managers* (PIMs) typically use processors running at 16–44 MHz [71]. These CPUs are sufficient for the specialized functions supported by IAs. Moreover, low-speed processors consume less power, thus providing longer operation with lighter batteries; both being key requirements for portable IAs.

Since audio/video communication is a primary function performed by IAs, the communication subsystem within the OS must be highly efficient to work well with the low-cost, slow hardware of IAs. The subsystem must be structured to handle both short as well as long messages with minimal overhead. Handling short messages efficiently is important for applications such as Internet telephony where live voice packets are usually just 30–50 bytes (as in the GSM audio encoding scheme [99] used in various Internet phones). On the other hand, video applications exchange long messages (10–15 kbytes [25]) and these too must be handled efficiently.

Different overheads come into play depending on whether short or long messages are being processed. *Data-touching* overheads (which include data copying and checksum overheads) tend to dominate when dealing with long messages. For short (audio) messages, the message size is just tens of bytes (so copying overheads are not important), but messages are sent once every 10–30ms [98]. With messages arriving with such high frequency, *non-data-touching* overheads (context switching, interrupt handling, I-cache miss overheads, etc.) become an important part of protocol processing.

Studies have shown that receive-side protocol processing is more complicated and has higher overhead than the send-side [55, 57, 79] and this is what limits throughput; so, here we focus on improving receive-side overhead. The send-side architecture is simpler than the receive-side because processing can occur as part of the send system call (as done in [19, 65, 79]) and this is the scheme we use as well. However, on the receive-side, protocol processing in general cannot occur at the time of the receive system call since the message may not have arrived yet. This leads to thread blocking and context switching which increases receive-side overhead, which is why we focus on improving the receive-side architecture.

We present schemes to improve both non-data-touching as well as data-touching overheads. For the former, we use application-specific knowledge to safely bypass selected layers within the protocol stack, completely avoiding all I-cache misses associated with those layers. We show that this *layer bypass* can be easily applied to live voice messages, resulting in considerable reduction in non-data-touching overheads. Regarding data-touching overheads (which affect long messages), we exploit the periodic nature of video applications. We show that the single-copy scheme presented for non-real-time systems in [15] — which requires specialized network adapter hardware to be feasible for non-real-time systems — works well without any hardware support for multimedia applications because of their periodic nature. We show that when this single-copy scheme is *combined with a real-time task scheduler*, it can be used effectively for video applications. We also show that our optimizations for real-time messages can be implemented without disrupting the handling of non-real-time messages. Moreover, our protocol architecture does not rely on any special hardware and is independent of the type of underlying network.

For the purpose of evaluation, we have implemented UDP/IP using our protocol architecture within EMERALDS. We chose UDP as the protocol to implement since it is commonly used for audio and video applications. The protocol code was taken from FreeBSD 4.4 and minor modifications were made to make it work with our protocol architecture.

6.2 Protocol Architecture Issues

Network protocol architecture has been an active area of research for many years [41, 97] and various techniques have been proposed to make protocol processing efficient. Following is an overview of I-cache and data-copying overheads, schemes proposed by other researchers to reduce these overheads, and shortcomings of these schemes.

6.2.1 Efficient I-Cache Usage

Communication protocols are designed to work with a wide variety of applications. They must accommodate varying communication patterns, error conditions, and operating modes. This is essential for a protocol to be widely accepted. The downside is that generality is achieved at the expense of performance. A large portion of the protocol code is devoted to checking for rarely-occurring errors or special message formats. These checks are usually coded as shown in Figure 6.1. Most of the time, there are no errors so that the body of the `if` statements never execute. However, code is still fetched into the I-cache, causing replacement misses. Moreover, repeated branches can cause CPU pipeline stalls. For relatively slow CPUs such as those used in IAs, this results in significant non-data-touching overhead which is important when processing short audio messages. As a result,

```
if (check1) {
    ...
}
if (check2) {
    ...
}
if (check3) {
    ...
}
```

Figure 6.1: Typical structure of error checks in protocol code.

researchers attempted to minimize such overheads by various means. In [86], techniques called *outlining* and *cloning* are presented in which frequently-executed paths through the protocol stack are identified (by studying the protocol code), and this information is passed to the compiler which places code in memory to minimize I-cache misses for these paths. In [93], *incremental specialization* is presented in which special code optimized for the common case is used whenever possible. The system includes a number of checks which cause a switch to the code which handles the fully general case if the special case no longer applies.

All these techniques can be considered as low-level optimizations. In general, they require a careful study of protocol implementation code to achieve full performance benefits and this entails considerable effort on the part of the programmer. Some methods have been proposed to partially automate these optimizations, but their full advantage is achieved only by manually applying the optimizations to protocol code. This indicates a need to develop an easier-to-use scheme to reduce I-cache misses; a scheme which does not require low-level fine-tuning of protocol code yet avoids all unnecessary I-cache misses.

6.2.2 Single-Copy Architectures

The single-copy network architecture was proposed by network adapter (NA) designers [15] to reduce data-touching overheads. The idea is to design an NA with enough buffer space so that on transmission, data is copied once from user-space directly to the NA, while on reception, data stays in the NA until the application makes a receive system call and then data is copied directly to user-space. In case NA buffers fill up, data has to be buffered within the kernel, leading to two data copies.

This architecture was proposed for general-purpose computing, and to be effective for such applications, the NA not only needs "enough" buffers, it also needs "flexible" buffers.

The Afterburner NA [15] uses linked lists to manage NA buffers. For both transmission and reception, it has two queues of in-use and free buffers, so it can continue operating as long as some free buffers are available. This is a more complicated and expensive NA design than common NAs such as LANCE [4].

LANCE[1] uses circular queues of buffers. For transmission, it transmits messages from the ring until it reaches a free buffer at which point it stops. For reception, it fills buffers in the receive ring until it reaches a filled buffer at which point it starts dropping packets. This simple design is low-cost and works perfectly with two-copy architectures. But if the single-copy architecture were to be used for an NA such as LANCE for general-purpose applications, there will be problems with both transmission and reception.

Transmission Issues: Suppose a protocol such as TCP is being used. TCP requires that message data be kept by the kernel until acknowledgment is received. Due to the circular manner in which LANCE accesses buffers, the kernel cannot keep a lock on one particular buffer for a long time because LANCE will stop once it reaches this buffer (even if other buffers are ready for transmission down the ring).

Reception Issues: Once LANCE fills some buffer b with a received packet, the kernel must remove data from this buffer before LANCE goes around the ring and comes back to this buffer. If buffer b has not been emptied, LANCE will simply stop when it reaches b (even if other buffers are free down the ring) and will start dropping packets. For general-purpose applications, this is likely since there is no bound on how long an application may take before making the receive system call.

When used for general-purpose applications, single-copy architectures work well only with hardware support. Without hardware support, they can quickly degrade in performance to the level of two-copy architectures. As we will show later, for real-time applications, the single-copy architecture is feasible *when combined with a real-time task scheduler*, and it does not require any special hardware support.

6.2.3 Our Design Goals

Our primary goal is to provide efficient audio/video communication support for IAs by lowering both data-touching and non-data-touching overheads. A secondary goal is to

[1] Current-generation IAs use modems for communication. However, as the network bandwidth requirements of IAs increase, we would expect IAs to start using simple, low-cost NAs similar to LANCE.

achieve this without hardware support from the NA. This is important for IAs since adding special features to the NA can increase hardware costs significantly. Another goal is that our architecture must be able to work with existing APIs (such as the BSD socket API) and must work for all network types, whether connection-oriented or not. In other words, we will make no assumptions about the network or the API and we will not assume existence of any special features in the NA (other than "sufficient" buffer space as discussed in detail in Section 6.3.3).

6.3 Protocol Architecture for Audio and Video

Real-time audio and video communication applications have certain characteristics not found in general-purpose applications:

- Application code executes periodically with well-known periods for both audio and video.
- Transport protocols used for communication do not use acknowledgments. UDP is commonly used over the Internet.

Acknowledgment-based transport protocols are usually not used for audio and video transmission. If a message is lost, then by the time the loss is detected and the message is retransmitted, the data contained in the message would already be too stale to be of any use. Forward error correcting codes at the application level are typically used to recover from lost messages. As such, UDP is the most commonly-used transport protocol for audio and video. We use this fact along with the periodic nature of multimedia applications to design an efficient architecture which lowers both data-touching and non-data-touching overheads.

The next subsection describes the basic structure we chose for our protocol architecture, followed by a description of our protocol processing optimizations in subsections 6.3.2–6.3.3.

6.3.1 Basic Structure

We chose *lazy receiver processing* (LRP) [19] (Figure 6.2) as our basic protocol architecture. In this scheme, the send-side processing is done by the application threads. When a send system call is made, the application thread enters the kernel, executes all relevant protocol and device driver code, and transfers the data to the NA for transmission. This is similar to the send-side schemes used in [65, 79].

The main advantage of LRP is for the receive-side processing. Under LRP, incoming

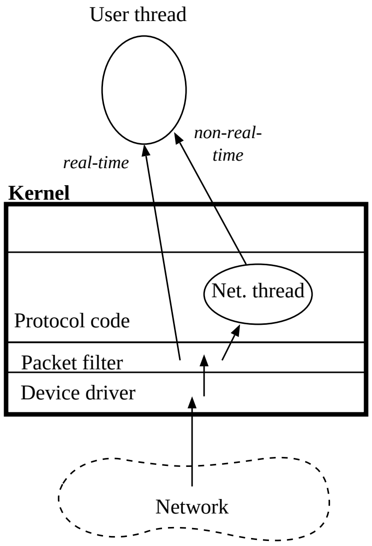

Figure 6.2: Lazy receiver processing.

packets trigger interrupts which cause the device driver to execute and it passes packets on to the packet filter [23, 84]. (The packet filter is a small piece of code which is dynamically installed in the kernel by individual applications to detect packets belonging to those applications and take appropriate actions.) The filter tries to forward packets directly to queues associated with the destination thread where packets stay unprocessed until the application makes a receive system call. This is possible for real-time messages since the application threads are usually periodic (with the period being ensured by the real-time task scheduler) so that packets are always processed within a known time interval.

For non-real-time applications, there is no bound on how long the application may take to make the receive call. As such, non-real-time packets are forwarded to a special network thread which performs protocol processing and keeps the message until the final destination thread makes a call to receive it. The network thread also acknowledges messages if needed (as in TCP). In fact, the need to send timely acknowledgments in protocols such as TCP is the primary reason for having a separate network thread.

LRP provides more predictable message handling than other protocol architectures such as the user-level architecture presented in [65] which always relies on special network threads for protocol processing. This can lead to *priority inversion* [82], i.e., handling of a high-priority real-time message (such as live voice) being unnecessarily delayed by the processing of lower-priority or non-real-time messages. Priority inversion can occur if the network thread executes at a priority different from the destination thread (this can happen if one network thread handles traffic for multiple application threads). For example, if the network thread runs at a priority lower than that of the destination thread, an intermediate-priority thread can preempt the network thread, preventing messages from reaching the higher-priority destination thread. In LRP, all protocol processing is done by the destination thread itself, which prevents any priority inversion and ensures predictable processing.

Another advantage of LRP is that it saves one context switch for real-time messages compared to architectures which always use intermediate network threads for protocol processing. This makes LRP ideally suitable for use in real-time systems.

Next, we describe our optimizations for protocol processing. Note that these optimizations do not depend on LRP and would work equally well with other protocol architectures.

6.3.2 Reducing Non-Data-Touching Overheads

For lowering non-data-touching overheads — especially those related to I-cache misses — we present a new scheme called *layer bypass*. It is easier to apply than low-level optimizations like outlining and specialization. It does not require any in-depth analysis of protocol code, and can be used effectively for short messages, which is where lower non-data-touching overheads are most beneficial.

Layer bypass relies on the observation that most of the functionality implemented by various protocol layers is simply not needed when processing short messages. The few operations that are needed are either already duplicated in the packet filter or can be easily migrated there. This allows various protocol layers to be bypassed, completely avoiding all I-cache misses these layers may have caused.

Layer bypass can be applied as follows. The protocol specification for various protocol layers is first studied to determine which aspects of the protocol are not needed for a given message stream (such as live audio messages). Those layers are identified which perform little or no functions. The few useful operations these layers do perform can all be placed together in the packet filter or some other such small module appropriately inserted in the protocol stack. Then the redundant layers can be bypassed completely.

Note that layer bypass is similar in spirit to specialization, i.e., it makes the common case fast. But it differs fundamentally from specialization in that it deals with macro operations, not small functions. If small portions of the code were to be bypassed, the resulting code will look just like in Figure 6.1 and performance may actually degrade. This is why specialization relies on a completely separate piece of code to implement the fast case, but this leads to increased code size which is a disadvantage in small-scale systems such as IAs. Layer bypass envisages functionality implemented in a large chunk of code (an entire layer) to be bypassed. This not only results in greater reduction in I-cache misses but also allows a single piece of code to exist for all cases with the protocol architecture providing the mechanism to bypass unnecessary layers. With layer bypass, all protocol layers exist and there is only one implementation of each layer. Layers not useful for certain applications are bypassed for those applications while the remaining applications can use the full protocol stack. Outlining, cloning, specialization, and other low-level optimization schemes can still be applied to protocol layers (if appropriate) since these schemes are orthogonal to layer bypass.

Next, we show that layer bypass can be used very effectively for audio messages. It can also be used for handling other types of short messages such as web server requests, and we present a small illustration as proof-of-concept.

Layer Bypass for Live Audio Messages:

To bypass a protocol layer for messages for a certain application, the application designer must consider the operations performed by that layer and decide whether these operations are needed or not for messages for that application. We illustrate this methodology by applying layer bypass to live audio messages for receive-side processing. Let's first consider the IP layer which performs the following major functions:

IP: Routing, fragmentation/reassembly, IP address checks, IP header checksum, checks for malformed headers and packets, IP option processing.

A receiving host does not perform any routing. Short live audio messages need not be fragmented/reassembled. IP options are used for network testing, so they do not apply here. The packet filter checks the destination IP address, so this function of the IP layer is already being handled by the packet filter. This leaves the various error checks which are related to rarely-occurring error conditions. It is safe to bypass the IP header checksum and other error checks for the following reason: the packet filter examines the various fields in the header and will recognize the packet as an audio message only if the fields contain expected values. If the header has been corrupted, the filter will not recognize the packet; it will be forwarded to the IP layer, normal processing will occur, the errors will be detected, and the packet will be discarded.

Next, we look at the feasibility of bypassing the UDP layer which performs the following major functions:

UDP: UDP datagram checksum, cooperate with IP layer in handling IP options, multicast a datagram to multiple local sockets (if requested), port address checks, generate ICMP messages if destination port does not exist, pass incoming datagrams to correct socket.

The UDP checksum is usually turned off for live audio messages since audio applications use their own forward error correcting codes (if they use error codes at all). This is because of the soft real-time nature of the application. Local multicast is not needed for audio applications, and the packet filter already checks the destination port address. If this check fails, the packet is routed through the full protocol stack where error handling (if needed) can occur. This shows that UDP can be bypassed safely for live audio messages.

This example shows how a high-level description of a protocol layer is all that's needed to determine feasibility for bypass. It also shows the usefulness of layer bypass for handling live audio messages. The IP layer can be bypassed because no reassembly is required, and UDP can be bypassed because the checksum is not needed. The packet filter checks the destination IP and port addresses, strips headers, and forwards messages directly to the socket layer, saving considerable I-cache misses. This results in a significant reduction in protocol processing overheads on the slow CPUs used in IAs (see Section 6.4 for measurements).

Layer Bypass for Web Servers:

To show that layer bypass is not just limited to live audio messages but can also be used for other short messages, we give an example of its application to web servers. Since it is not the main topic of this chapter, we omit details and only present the general framework.

Layer bypass can be used for HTTP request messages. A server may receive thousands of such messages per second. These messages are short, so the IP layer can be bypassed for the same reasons as for audio messages. This leaves the TCP layer. The major functions performed by TCP are message sequencing and reliable delivery (using acknowledgments). Regarding sequencing, if web document request messages get re-ordered, it makes no real difference. For reliability, no separate acknowledgment messages are needed since the ack will be piggybacked on the server reply message. However, TCP keeps track of the sequence numbers of incoming messages to perform acknowledgments. So, the TCP layer can be bypassed provided that the packet filter can update the TCP connection's sequence numbers by invoking the appropriate routines in the TCP implementation. This is a violation of the layering concept, but packet filters violate layering anyway. This leaves the issue of the TCP checksum. Feasibility of bypassing the checksum has to be determined on an application-by-application basis. However, bypassing it should be safe for most web servers since MAC layers already provide one level of error checking. □

The above examples demonstrate the usefulness and applicability of layer bypass. Layer bypass has one drawback: it is most effective when the layers being bypassed are at the top (for outgoing messages) or bottom (for incoming messages) of the protocol stack. To bypass middle layers, a filter will have to be inserted between layers and this will increase processing overhead for messages which do not utilize layer bypass. However, for end-host receive-side processing, the layers feasible for bypass are usually the bottom layers. For example, layer bypass is most useful for short messages where non-data-touching overheads matter the most, and for such messages, the IP layer can almost always be bypassed since message reassembly is not needed.

6.3.3 Improving Data-Touching Overheads

We now show that the single-copy scheme — which, without hardware support, has limited value for non-real-time systems — can be used effectively for video communication with no special hardware support. The key to the effectiveness of the single-copy scheme in real-time systems is a real-time scheduler which guarantees that the application executes at its period and does not face unpredictable delays. (Note that such a scheduler is needed anyway — not just for the single-copy scheme — for the purpose of meeting timing constraints.) Hence, incoming packets will stay in the NA buffers for no longer than the period of the application. This is in contrast to non-real-time applications where no such bound exists on how long an application may take to retrieve its packets from the NA.

In our scheme, packet arrivals trigger interrupts. The device driver executes and forms an mbuf chain of the packets. Mbufs are linked lists of buffers which allow easy addition and removal of headers (see [66] for details). Data is left in the NA and the mbufs are made to point to that data. The packet filter then enqueues the mbuf chain in the appropriate socket, and the associated user thread is signaled. If the thread had already made a receive call then IP, UDP, and socket layer processing occurs as soon as the thread is scheduled for execution. Otherwise, packets are processed when the receive call is made.

Before the device driver exits the interrupt service routine, it checks if a free buffer is available for more packets. For LANCE, this means checking if the next buffer in the ring has been processed and relinquished by the kernel. If not, data is copied from that buffer into kernel buffers and the NA buffer is freed to avoid dropping packets. Obviously, when this happens, performance is the same as the two-copy scheme. Then, the important question is: how often can this happen under the condition that both real-time as well as non-real-time applications (such as telnet and web browsing) are receiving packets?

Real-time audio and video applications run with some period T. T for audio is quite short, usually 10–30ms [98], so audio is not a problem. Video applications usually run at 30 frames/s [96] but to conserve CPU and network bandwidth — which is important in IAs — some may run at a slower rate of 20 or even 10 frames/s, giving a T as large as 0.1s. This is the maximum time messages for a video application have to stay in NA buffers. If NA buffers are about to overflow and the video messages have not been processed, packets for these messages have to be copied out of the NA into kernel buffers to make room for incoming packets. This can occur if a burst of non-real-time packets arrive, filling NA buffers in a short period of time. Following is an analysis of how frequently this might happen when both real-time and non-real-time packets are being received through the NA.

Estimating Non-Real-Time Packet Arrivals:

In the past, Poisson processes have been used to model packet arrivals [29]. This reduces the complexity of network traffic analysis because of the simplicity of Poisson processes. However, various studies have shown that wide-area network traffic is too bursty to be correctly modeled by a Poisson process [16, 90]. Telnet arrivals have been modeled by a Pareto distribution [90], but only empirical models exist for FTP [16] and web browsing [13, 14]. This precludes any closed-form derivation of non-real-time packet arrival distributions. Added to this is yet another difficulty that network traffic characteristics may change from time to time or place to place. These characteristics depend on factors such as network congestion, speed at which servers can transmit data, etc. This means that any calculation of packet arrival distributions will be an estimate at best.

Accurate modeling of network traffic is not our intent. All we want is to show that receiving even 10–20 non-real-time packets within time interval T is highly improbable, and get some idea of how improbable that is. Since network adapters usually have 128–256 receive buffers [4, 15] — and this is likely to increase even further as memory densities increase and cost decreases — receiving even 10–20 packets within T seconds is not enough to disrupt the handling of real-time packets. An engineering approximation of packet arrival rates is all that is needed to get an idea whether the single-copy scheme can be used successfully in IAs or not.

For evaluation purposes, we chose to use web browsing as a representative non-real-time application. Measurements of web traffic have shown that retrieval of even small web pages take more than 2 seconds [14]. This is the time needed to look up the remote host's DNS entry and establish TCP connections. After this initial phase data transfer begins at the rate of 1 byte per 90–100μs [14]. Most web pages are relatively small-sized. Measurements in [14] show most pages to be 256–512 bytes, but with the increasing use of in-line images, this is likely to increase. Even then, the trend of favoring small-sized pages will persist, especially considering the small display screens that IAs have. As such, we assume a 10kbyte page size.

We know of no study which correlates size of a web page to the number of network packets needed to transfer the page, so, instead, we use some common-sense approximations. An Ethernet packet can carry up to 1500 bytes, so 10kbyte require a minimum of 7 packets. However, each in-line image is usually sent as a separate message, so we will conservatively assume 20 packets to carry every 10kbyte of data. With these assumptions and using 90μs as the per-byte transfer time (which is faster than the wireless link speeds available to most IAs today), we get a packet arrival rate of 22 packets/s or 2.2 packets/0.1s. Even if due to burstiness, five times as many packets arrive within $T = 0.1s$, we still get only 11 packets/T. Burstiness is more likely when downloading large documents, but after downloading such a document, the user will take more time to read the document (maybe several minutes) which increases the gap between downloads and spaces the series of bursts further and further apart.

Considering both connection and reading delays, downloads are separated by at least several seconds. Even if 5 bursts of 11 packets/T occur during download (highly unlikely since the majority of web pages fit in fewer than 30 packets) and the user spends just one second reading the document, the probability of getting a burst of 11 packets/T is only 5 times in $35T$ seconds or 0.143. This is negligible considering that NAs typically have more than a hundred buffers.

In summary, lack of characterization of network traffic prevents an accurate calculation of the pattern of non-real-time packet arrivals. We have used available data regarding web traffic to show that even under highly-exaggerated network use conditions, probability of receiving a large number of non-real-time packets within T is still small, so that our protocol processing optimization should be useful for real-time applications most of the time. $\Box$

Note that the above analysis is true even for the NAs with the simplest buffer-management policies, such as LANCE. As such, the single-copy scheme for real-time applications does not require any special/expensive hardware support which is an important consideration for IAs.

6.3.4 Non-Real-Time Messages

Non-real-time traffic can co-exist with real-time traffic, but because of unpredictable execution of non-real-time applications, no statistical analysis as one presented above is possible for such applications. If the application responds quickly enough, only one copy will be needed, otherwise packets will have to be copied and stored in kernel buffers. At the device driver level, when a non-real-time message arrives, it will be left in the NA buffer till it is either processed or it has to be copied to the kernel to free up the NA buffer. At the protocol stack level, special network threads perform protocol processing for non-real-time messages (Figure 6.2). The packet filter can be configured to forward non-real-time messages to an appropriate network thread instead of sending them to the application thread. Applications can specify if certain communication end-points are to receive real-time or non-real-time messages, and the packet filter can use this information to forward messages accordingly.

6.4 Evaluation Results

We want to evaluate the effectiveness of our architecture in handling both short audio and long video messages. For short messages, we measure receive-side overheads both with and without layer bypass to show the effectiveness of layer bypass in reducing protocol processing overheads. For long messages, we compare the single-copy and standard two-copy schemes.

6.4.1 Platform

We implemented our protocol architecture within EMERALDS on a 25MHz Motorola 68040 processor with separate 4kbyte data and instruction caches. EMERALDS features highly optimized context switching, interrupt handling, and memory usage [124]. The 68040 is typical of CPUs used in many IAs today. (We will later discuss the results on a faster processor.)

We use two processors in our experiments, connected by a 10Mb/s private Ethernet using the LANCE network adapter. LANCE uses DMA to transfer data between its buffers and the network. In some cases, the memory contention between DMA and CPU can distort overhead measurements. In all measurements in this section, we have attempted to minimize this distortion so that measurements depend only on the protocol architecture and not on the NA.

Operation	*Overhead (μs)*
Context switch	9.2
Interrupt handling (with 1 context switch and device-driver code)	40.0
Packet Filter	6.9

Table 6.1: Measurement of some non-data-touching overheads.

For evaluation, we implemented UDP/IP using our architecture. The protocol and LANCE device driver code was taken from FreeBSD 4.4 and minor modifications were made to make it work with EMERALDS. We also added our own packet filter rather than using the high-overhead BSD packet filter. For simplicity, we implemented a UDP/IP-specific filter. Interested readers are referred to [23] for more generalized high-performance packet filters.

6.4.2 Performance Improvements

We sent datagram messages from one processor to another and measured the total overhead of receive-side protocol processing including interrupt handling and all relevant context switches. The total receive overhead was measured by noting the increase in the execution time of a delay loop of known duration running on the receive-side host with message reception being the only other on-going activity on that host. Measurements were made using a 5MHz on-chip timer. For each data point (fixed message size) we repeated this experiment 100 times and averaged the results. Further increase in number of samples did not result in any significant change in averaged results. We also measured various non-data-touching components of the receive-side overhead by instrumenting the kernel to measure execution times of relevant operations as shown in Table 6.1.

Short Messages:

Figure 6.3 shows the total receive overhead for short message sizes. This figure presents measurements for the cases when processing is done by a special network thread (labeled “standard architecture”), regular LRP, and LRP when layer bypass is used as well. In all cases, the UDP checksum is turned off. From the figure, we see the benefit of bypassing IP and UDP layers. Performance is improved 20% (beyond that of LRP). Note that the sharper variations in the plots are a result of BSD’s mbuf allocation scheme [57] and are not related to the protocol architecture.

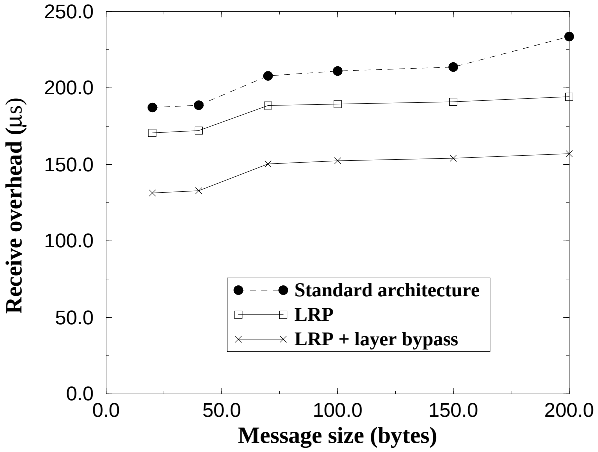

Figure 6.3: Receive overhead for short messages.

The above measurements were for a relatively slow processor typical of CPUs used in IAs. As already mentioned, layer bypass can also be applied to web servers which use much faster processors. To evaluate the usefulness of layer bypass for such applications, we emulated our protocol architecture on a 167 MHz Sparc Ultra-1 workstation (16 kB I/ 16 kB D caches) running Solaris 2.5.1. We use two threads which execute all the protocol code at the user level to send messages to one another. LANCE is emulated using shared memory and interrupt overhead is emulated through context switches between the threads. All the device driver, packet filter, protocol, and socket code executed in the previous experiment is also executed in this experiment. The results are shown in Figure 6.4. Even though protocol processing overhead is dominated by heavy-weight Solaris context switches (two switches cost about 40μs), still layer bypass delivers a 14% improvement in performance.

Long Messages:

Figure 6.5 plots the receive overhead for messages ranging from 20 to 6000 bytes. It shows the overheads for the two-copy and single-copy schemes. For short messages, the savings from the single-copy scheme are not significant since non-data-touching overheads account for most of the receive overhead. But as message size increases and copying costs begin to dominate, the benefit of eliminating one copy becomes more apparent and the percentage reduction in receive overhead (compared to the standard two-copy architecture) increases steadily until it reaches 22% for a message size of 1.4kbytes. Further increase in message size results in messages being fragmented into two packets (Ethernet has an MTU of 1500 bytes). This causes sharp increases in overhead every 1458 bytes as shown

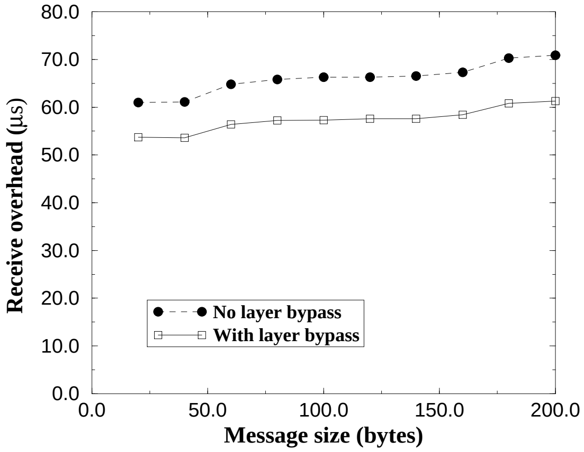

Figure 6.4: Receive overhead for short messages on an Ultra-1 Sparc station.

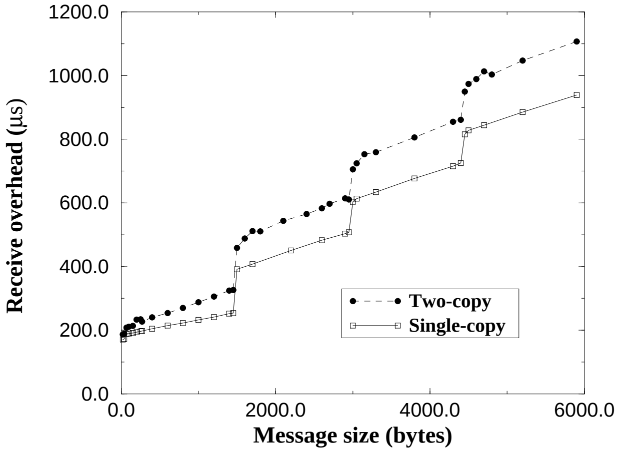

Figure 6.5: Measurement of receive overheads for long messages.

in the figure since each packet generates a separate interrupt. The increase in going from one packet to two is large since this is also the point that data starts to overflow the small 4kbyte data cache. These data cache misses cause the percent overhead reduction to level out at about 15% for messages larger than 1.4kbyte.

6.4.3 Improved Predictability

Figures 6.6–6.7 show the effectiveness of LRP in improving predictability. These figures show the variations in execution periods of a high-priority thread which executes an infinite loop to receive 1kbyte messages which arrive at a period of 1.5ms (100 message receptions shown per figure). A low-priority thread is also receiving messages on the same node. These

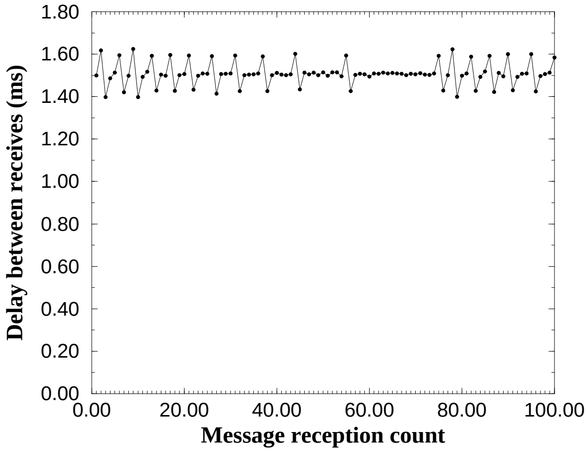

Figure 6.6: Timeline plot of delays between consecutive receptions for the standard protocol architecture.

messages arrive 4ms apart. We kept these messages short (10 bytes) to minimize variation in protocol processing times of high-priority messages due to background DMA of low-priority messages.

Figure 6.6 depicts the case when the standard architecture is used and the thread with the 1.5ms period has higher priority than the network thread. Protocol processing for the low-priority thread interferes with that of the high-priority thread, leading to variations in the period of the latter of as much as 0.12ms

Figure 6.7 shows the case for LRP. Variations are significantly smaller, maximum being just 0.08ms — 33% smaller than that for the standard architecture. The variations are primarily due to interrupts and limited priority inversion which occurs when the low-priority receive thread is in a critical section and cannot be preempted by the high-priority thread. These measurements justify our choice of LRP for ensuring predictability.

6.5 Related Work

Many researchers attempted to reduce data-touching overheads in various ways. Virtual memory page re-mapping is commonly used to reduce data-copying costs when crossing protection boundaries [20]. This technique works well to a certain extent, especially when combined with the mbuf mechanism of BSD [66]. Mbufs are linked lists of buffers. To add a header, all that needs to be done is allocate a memory buffer (anywhere in the address space), put the header data in it, and link this buffer at the head of the mbuf chain. Then,

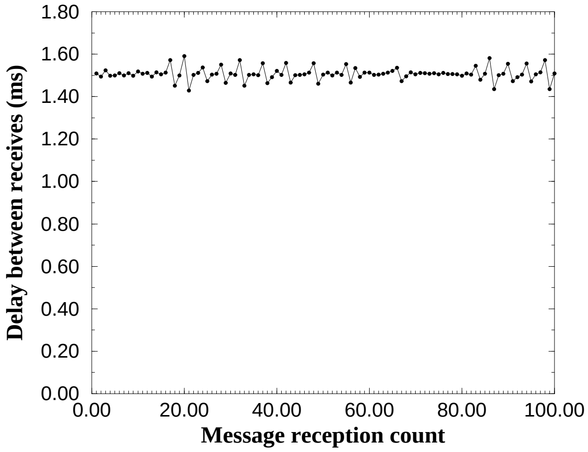

Figure 6.7: Timeline plot of delays between consecutive receptions for LRP.

as the message passes from one process to another, the mbuf chain is mapped out of the former process' address space and into that of the latter. For message transmission, if the message can be transmitted immediately over the network, data needs to be copied only once: from host memory to network adapter (NA). But if the kernel must retain the message (possibly for retransmission later on), then two copies must occur unless the application is modified so that its outgoing message data is aligned at page boundaries. The kernel can then map the pages containing the message out of the user's address space, replace them with other pages, and unblock the caller.

The situation is even more complicated on the receive side. Once a message arrives, it is copied from NA to host memory, and protocol processing occurs. But transferring the message from kernel to user will require another copy unless the API is such that it allows the kernel to place messages anywhere in the user's address space and then inform the user of the location. This is not the case with the BSD socket interface, where the location for incoming messages is specified by the user, not by the kernel, resulting in two data copies. Yet another problem with memory remapping is the cost of remapping pages from one domain to another. In [20] an optimization is presented in which mappings are cached to reduce overhead significantly. This works reasonably well for transmission, but works for reception only if the network protocol allows packet-level demultiplexing at the device driver level. Low-level *message* demultiplexing is quite common by means of packet filters [84] but it requires the first packet of a message to arrive before the message's final destination can be determined. If packets get re-ordered, then *packet*-level demultiplexing is possible only if the network is connection-based such as ATM. In short, caching page mappings cannot work over connectionless networks such as Ethernet.

Other protocol architecture optimizations have been proposed, such as giving applications direct (but controlled) access to the NA [21], but these schemes usually require hardware support from the NA.

Regarding non-data-touching overheads, a batch processing technique was proposed in [8]. The idea is to wait till several small messages have been received; then process them as a batch, thereby reducing I-cache misses. This scheme is useful for handling bursts of short messages but is not effective for live audio messages because they are spaced at regular intervals in time.

An innovative protocol architecture was presented in [115] which also aims to optimize the fast path. The packet filter compares headers of incoming packets against pre-computed expected values. In case of a match, the packet is forwarded immediately to the application, thereby reducing the latency of processing. The actual protocol stack is invoked later to update state and pre-compute the expected header for the next packet. This scheme reduces latency but has no effect on throughput since the entire protocol stack still executes.

6.6 Conclusion

Information appliances (IAs) are an emerging class of embedded devices which are used for specialized communication tasks such as audio/video communication over the Internet. The Internet-centric nature of IAs combined with the need to keep costs low in these mass-produced devices (which results in the use of slow/cheap processors) dictates that the communication subsystem with the OS be highly efficient to enable audio/video communication despite slow hardware. In this chapter we presented a protocol architecture which focuses on improving message reception overhead (which usually exceeds transmission overhead) for real-time audio and video. Our design not only adapts existing optimizations for use in real-time communication but also includes new optimizations to reduce overhead and increase throughput.

Video applications are characterized by large message sizes while live audio messages are short but frequent. This presents a challenge for the protocol architecture designer. Both data-touching and non-data-touching overheads must be reduced. The former is important for large video messages and the latter for short audio messages. For non-data-touching overheads, we presented the layer-bypass scheme under which entire layers within the protocol stack are bypassed by transferring the few useful functions they perform to the packet filter. This way, I-cache misses associated with protocol processing are minimized, which reduces overhead by 14–20% for short messages. To reduce data-touching overheads, we used the single-copy scheme [15], showing that for real-time messages, it can be used effectively without any hardware support from the NA, and improves overhead by 15–22%.

CHAPTER 7

MESSAGE SCHEDULING FOR CONTROLLER AREA NETWORK (CAN)

The previous chapter dealt with communication requirements of embedded devices connected to the Internet. In this chapter, we address communication issues related to another important class of embedded systems which are fieldbus-based such as automotive and factory automation systems. These systems consist of multiple computational nodes, sensors, and actuators interconnected by a low-speed LAN called a field bus [94]. Of the multiple field bus protocols available for such use (including SP-50 FieldBus [44], MAP [80], TTP [61], etc.), the Controller Area Network (CAN) [49, 104] has gained wide-spread acceptance in the industry [118].

CAN is a contention-based multi-master network which has the potential to efficiently handle both periodic as well as sporadic messages. It is currently being used in a wide range of embedded real-time control applications [118] including automotive control, industrial automation, and medical monitoring. Its main attraction is its low cost (a CAN interface chip costs about $5) and reliability features like atomic multicasts and fault confinement. It provides prioritized bus access and fast response times for high-priority messages, making it ideal for use in real-time control applications.

Control networks must carry both periodic and sporadic real-time messages, as well as non-real-time messages. All these messages must be properly scheduled on the network so that real-time messages meet their deadlines while co-existing with non-real-time messages (we limit the scope of this chapter to scheduling messages whose characteristics like deadline and period are known *a priori*). Previous work regarding scheduling such messages on CAN includes [112, 113], but they focused on fixed-priority scheduling. Shin [102] considered *earliest-deadline first* (EDF) scheduling, but did not consider its high overhead which makes

EDF impractical for CAN. In this chapter, we present a scheduling scheme for CAN called the *mixed traffic scheduler* (MTS) [122, 125, 127] which increases schedulable utilization and performs better than fixed-priority schemes while incurring less overhead than EDF. We also describe how MTS can be implemented on existing CAN network adapters. We address the problem of how to control priority inversion (low-priority message being transmitted ahead of a higher-priority one) within CAN network adapters and evaluate different solutions for this problem.

We measure various execution overheads associated with MTS by implementing it within EMERALDS. Using an emulated CAN network device (another 68040 acting as a CAN network adapter and connected to the main node through a VME bus), we present detailed measurements of all execution, interrupt handling, task scheduling, and context switching overheads associated with MTS to show the feasibility of using MTS for control applications.

In the next section we give an overview of the CAN protocol. Section 7.2 describes the various types of messages in our target application workload. They include both real-time and non-real-time messages. Section 7.3 gives the MTS algorithm. Section 7.4 discusses issues related to implementation of MTS, focusing on the priority inversion problem. Section 7.5 evaluates the network schedulability performance of MTS (compared to EDF and DM) and presents implementation overhead measurements. The chapter concludes with Section 7.6.

7.1 Controller Area Network (CAN)

CAN [49, 104] is an advanced serial communication protocol for distributed real-time control systems. It is a contention-based multi-master network whose timeliness properties come from its collision resolution algorithm. Some other salient features of CAN are prioritized bus access, reconfiguration flexibility, high reliability in noisy environments through CRC checks and bit stuffing, and network speeds up to 1Mb/s (in the rest of the chapter we assume this speed). The CAN specification defines the physical and data link layers (Layers 1 and 2 in the ISO/OSI reference model). Both layers are implemented in a network adapter (NA) chip which connects the processing element (like microprocessor or smart sensor) to the bus. Such chips are available from various vendors with a variety of features. Recently, some vendors have introduced microcontrollers with on-chip CAN modules which allow these microcontrollers to interface directly with the CAN bus.

SOF	Identifier	Control	Data	CRC	Ack	EOF
1	12	6	0-64	16	2	7

SOF: Start of Frame
CRC: Cyclic Redundancy Code
EOF: End of Frame

Figure 7.1: Various fields in the standard CAN data frame along with the length of each field in bits (including field delimiter bits).

7.1.1 CAN Data Frame

A data message in CAN has seven fields as shown in Figure 7.1. CAN allows two message formats to coexist on the same bus. They differ in the length of the *identifier* (ID) field: the *standard* format has an 11-bit ID, whereas the *extended* format has a 29-bit ID. The ID field (i) controls bus arbitration, and (ii) describes the meaning of the data (message routing).

Here we describe message routing in CAN (bus arbitration is described laster in this section). The ID, instead of containing some destination address, contains a code identifying the meaning of the data. CAN allows all or part of the ID field to be used for this purpose. For example, if the ID is 11 bits long, periodic messages from a temperature sensor may have a binary code xxxxxx10110, where an x denotes a bit not being used for identification. All nodes desirous of knowing the current temperature will set *filters* in their CAN network adapters to match the above code. Then, whenever a message with this ID code is sent on the bus, the NA will automatically receive it and notify the processing element of the node. This scheme is called *message filtering.*

Each CAN message can contain 0 to 8 bytes of data in the *data* field. The minimal CAN data frame has 47 bits when no data bytes are sent (44 bits for the frame plus a 3-bit inter-frame space). When 8 bytes of data is sent, the frame is 111 bits long.

For safety of data transfer, a 15-bit CRC check is sent with each message. This CRC is calculated over the SOF, ID, control, and data fields.

The *control* field contains the *data length* field plus one bit which identifies the frame as standard or extended. The data length field is 4 bits wide and specifies the number of bytes in the data field, from 0 to 8.

The *ack* field is used to acknowledge correct reception of a message. This is a single bit which is sent recessive by the transmitter. As receivers receive the SOF, identifier, control, and data fields of a message, they locally calculate the CRC over these bits. Next they receive the CRC field which they compare with their locally-calculated value. If they match, the receiver overwrites the ack bit on the bus with a dominant bit, signaling correct reception of the message. But if an error is detected through a CRC mismatch, the node signals an error as described next.

7.1.2 Active Error Detection and Atomic Multicast

Once a receiver detects a corrupted message, it not only discards that message but also transmits an *error flag.* This flag is a special sequence of bits which purposely violate the bit stuffing rules of CAN. The error flag is transmitted by the receiver while the sender of the message is still transmitting the last few bits of the original message. The error flag overwrites the last portion of the message, ensuring that if other nodes did not detect a CRC failure, they will at least detect the violation of bit stuffing rules. This ensures that if one node detects an error, all nodes are notified of it and they too discard that message. This results in atomic multicast, i.e., either all receivers receive a message or none do.

7.1.3 Bus Arbitration Mechanism

CAN makes use of a wired-OR (or wired-AND) bus to connect all the nodes (in the rest of the chapter we assume a wired-OR bus). Two logical bit representations are defined: *dominant* and *recessive*. If even a single node transmits a dominant bit, the bus will reflect a dominant bit, else it will reflect a recessive bit.

When a processor has to send a message it first calculates the message ID which may be based on the priority of the message. The ID for each message must be unique to prevent a tie. Let each ID be b bits long.

The bus acquisition algorithm works as follows. Processors pass their messages and associated IDs to their CAN NAs. The NAs wait till the bus is idle, then transmit the SOF which is a single dominant bit. All NAs synchronize to the leading edge of the SOF sent by the station starting transmission first. Following this synchronization, the NAs write their respective IDs on the bus, one bit at a time, starting with the most significant bit. After writing each bit, each NA waits long enough for signals to propagate along the bus, then it reads the bus. If a node had written a recessive bit but reads a dominant one, it means that another node has a message with a higher priority. If so, this node drops out of contention. After b such rounds, there is only one winner and it can use the bus.

7.2 Workload Characteristics

In control applications, some devices exchange periodic messages (such as automotive ABS controller) while others are more event-driven (such as smart sensors). Moreover, operators may need status information from various devices, thus generating messages which do not have timing constraints. To cover all these varieties, we classify messages into three broad categories:

1. Hard-deadline periodic messages.
2. Hard-deadline sporadic messages.
3. Non-real-time (best effort) aperiodic messages.

7.2.1 Periodic Messages

A common example of this type of messages is automotive ABS control. The controller must periodically sample the current rotational velocity and slip of the wheel and then send appropriate corrections to the braking actuators. Such messages have hard deadlines, because if the update message to the actuators is delayed beyond its deadline, the car may skid out of control. In general, most sensor-controller-actuator loops have hard-deadline periodic messages. Such systems include all robots and industrial cutting tools as well as various automotive actuators.

A periodic message i has period T_i, length C_i, and relative deadline D_i, the last being defined relative to the release time of the message. We also define the absolute deadline d_i of a message as its deadline relative to the global time frame (defined in terms of a system of synchronized clocks). Also, $C_i \leq D_i \leq T_i$.

Note that a single periodic message will have multiple invocations, each one period apart. So, whenever we use the term *message stream* to refer to a periodic message, we are referring to *all* invocations of that periodic message.

7.2.2 Sporadic Messages

Strictly speaking, all events in the real world are aperiodic in nature. If these events are expected to occur frequently enough, periodic monitoring can be used to detect them and take appropriate action (as in ABS control). There are other events which are not as frequent, such as temperature of a process exceeding a critical threshold. In fact, maximum interval between two such events is unbounded (event may never occur again). In such cases, using periodic messages is a waste of network bandwidth and CPU cycles because there is nothing to say most of the time.

Smart sensors [9] are most suitable for detecting such events. These sensors have DSP capabilities to recognize events on their own, so they signal the controller only when required. If these messages are treated as purely aperiodic, then we are assuming that they may be released at any time — even in rapid succession. If so, we will not be able to guarantee their delivery by their deadlines. Fortunately, in most real-world situations, there exists a minimum interarrival time for aperiodic events. In the temperature hazard example mentioned above, once the hazard is detected, the process will probably be shut down and restarted later on — during which time there cannot be any more temperature hazard messages. This corresponds to a minimum interarrival time (MIT) for such messages. Such aperiodic messages which have a MIT are called *sporadic messages* [85]. Knowing the MIT of a sporadic message makes it possible to guarantee its delivery even under the worst possible situation.

7.2.3 Non-Real-Time Messages

In automotive systems, a monitoring process needs to collect status information from various controllers for on-board diagnostic purposes. Similarly, in manufacturing and process control applications, an operator must be able to monitor the status of every device in the system. Such messages are non-real-time because they do not have timing constraints. Any communication protocol for control applications must be able to accommodate such messages while guaranteeing the deadlines of real-time traffic.

7.2.4 Low-Speed vs. High-Speed Real-Time Messages

Messages in a real-time control system can have a wide range of deadlines. For example, messages from a controller to a high-speed drive may have deadlines of few hundreds of microseconds. On the other hand, messages from devices such as temperature sensors can have deadlines of a few seconds because the physical property being measured (temperature) changes very slowly. Thus, we further classify real-time messages into two classes: *high-speed* and *low-speed*, depending on the tightness of their deadlines. As will be clear in Section 7.3.2, the reason for this classification has to do with the number of bits required to represent the deadlines of messages.

Note that "high-speed" is a relative term — relative to the tightest deadline D_0 in the workload. All messages with the same order of magnitude deadlines as D_0 (or within one order of magnitude difference from D_0) can be considered high-speed messages. All others will be low-speed.

7.3 The Mixed Traffic Scheduler

As stated earlier, access to the CAN bus is controlled by the IDs of competing messages. This necessitates that messages be assigned IDs in a proper manner to ensure their transmission by their deadlines. In other words, message scheduling on CAN corresponds to the proper assignment of IDs to messages.

To see the difficulties faced in scheduling messages on CAN, we must first consider a typical CAN network adapter (NA). Various CAN NAs usually have memory space for one or more messages. When a processor has to send a message, it will calculate the ID and transfer the message (with its ID) to the NA. From then on, the NA will function autonomously: it will compete for the bus with the message ID, and upon getting access, it will transmit the message (there may be an option to notify the processor once a message has been sent).

Once a message has been transferred to the NA for transmission, its ID will stay fixed unless the processor comes and updates it. If the ID is to be derived from the message's priority, that priority should stay fixed (at least for reasonably long periods of time). For this reason, fixed-priority scheduling is a natural fit for CAN. Each message will have a unique priority which will form its ID. However, in general, fixed-priority schemes give lower utilization than other schemes such as non-preemptive EDF. Non-preemptive scheduling under release time constraints is NP-hard in the strong sense [50], meaning that there is no polynomial time scheduler which will always give the maximum schedulable utilization. However, the authors of [120] showed that EDF performs better than other simple heuristics. This is why several researchers have used EDF for network scheduling [26, 54, 121]. This motivates us to use EDF to schedule messages on CAN, but EDF incurs high overhead (as discussed later) which makes it impractical for CAN.

In this section, we first describe the problems associated with fixed-priority and EDF scheduling, then in Section 7.3.4, we present MTS as a solution for these problems. MTS combines EDF and fixed-priority scheduling to overcome the problems of EDF, making MTS practical for CAN.

priority	deadline	uniqueness

Figure 7.2: Structure of the ID for EDF scheduling.

7.3.1 Fixed-Priority Scheduling — Low Utilization

As already mentioned, fixed-priority scheduling is the natural choice for currently available CAN bus interface chips. The most popular form of fixed-priority real-time scheduling is *rate monotonic* (RM) [76]. In this scheme, messages with a shorter period get higher priority than those with longer periods. RM assumes that deadline equals period, which is not always true in reality. Instead of RM, we can use its close relative, *deadline monotonic* (DM) scheduling [69]. With DM, messages with tighter relative deadlines are assigned higher priorities and these priorities form the ID for each message [112, 113].

DM is a simple scheme and is easily implementable on CAN. However, to get greater schedulable utilization, we would like to use EDF to schedule messages on CAN.

7.3.2 Earliest-Deadline Scheduling — Deadline Encoding Problems

EDF works by giving higher priority to messages with earlier deadlines-to-start transmission at the scheduling instant. Our goal is therefore to make the IDs reflect the deadlines of messages. Moreover, each message must have a unique ID (which is a requirement of CAN). This can be done by dividing the ID into three fields [102], as shown in Figure 7.2. The *deadline* field is derived from the deadline of the message. Actually, it is the logical inverse of the deadline because we want the shortest deadline to have the highest priority. To deal with the case when two messages have the same deadline, each message has a unique code which forms its *uniqueness* field. If two messages have the same deadline, the one with the higher uniqueness code will win. This uniqueness code also serves to identify the message for reception purposes. For EDF scheduling, messages may be assigned codes arbitrarily as long as they are unique for each message [102]. However, as we will see later, the question of assigning uniqueness codes will be critical in MTS.

In Figure 7.2, the *priority* field is a single bit used to distinguish real-time and non-real-time messages. It is 1 for real-time messages and 0 otherwise. This ensures that real-time messages always have higher priority than non-real-time ones.

As time progresses, absolute message deadlines (i.e., logical inverse of the actual deadlines) get larger and larger. Eventually, they will require more bits than are available in the CAN ID field. The obvious solution is to use *slack time* [102] (time to deadline) instead of the deadline itself, but this introduces two other problems:

P1. Remaining slack time of a message changes with every clock tick. This will require IDs of all messages to be updated continually (at the start of each arbitration round). This will put too much burden on the local CPU.

P2. A typical communication workload in a real-time control system may have messages with vastly different deadlines. This means that we must encode a wide range of laxities, and there may not be enough bits in the CAN ID field to do this.

Problem **P1** can be solved by using a wrap-around scheme which we describe in Section 7.3.3. The main reason EDF is impractical for CAN is because of **P2**: too many bits are required for the deadline field in the ID. In a typical workload, messages associated with high-speed drives may have deadlines in the hundreds of microseconds range (high-speed messages). Other messages, such as those related to temperature sensors, may have deadlines of several seconds (low-speed messages). If we represent deadlines at the granularity of, say, a microsecond, then more than 20 bits will be required to represent deadlines of several seconds.

One may say that if the extended format of CAN is used with its 29-bit IDs, then there will be enough bits to represent deadlines with enough left over for the uniqueness field. Unfortunately, if this scheme is used, each message will be 20 bits longer compared to the standard 11-bit format of CAN (the extended format uses two more framing bits than the standard format). This means that 20–30% bandwidth will be wasted just because of using the longer ID format, and the loss in schedulability (because of blocking effects of longer message frames) will be even greater. This makes EDF impractical for CAN.

7.3.3 Time Epochs

Here we discuss a solution to problem **P1**. We will face the same problem with MTS and we will use the same solution.

One simple way to solve **P1** will be to redesign the bus interface chips to have programmable counters in appropriate positions of the ID. This way, the slack time will be updated automatically at every clock tick. However, at present such chips are not commercially available. Even if they were, they would be more expensive than chips without counters. This motivates us to investigate a software solution (a cost/performance tradeoff).

In a software solution, the CPU will still have to update the ID, but we want to reduce the frequency of these updates, i.e., spend less CPU-time on updates. Our solution uses actual deadlines (instead of slack time) but expresses them relative to a periodically-increasing reference called the *start of epoch* (SOE). The time between two consecutive SOEs is called the *length of epoch*, ℓ. Then, the deadline field for message i will be the logical inverse of $d_i - \text{SOE} = d_i - \lfloor \frac{t}{\ell} \rfloor \ell$, where d_i is the absolute deadline of message i and t is the current time (it is assumed that all nodes have synchronized clocks [31]). Value of ℓ depends on what fraction of CPU-time the designer is willing to allow for ID updates. Let this fraction be x. Let M be the MIPS of the CPU and n be the number of instructions required to do the update. Since each update must be ℓ seconds apart, $\ell \geq \frac{n}{xM \times 10^6}$.

This ID update scheme is needed for MTS as well. Next, we describe the details of MTS, then in Section 7.3.5, we present a protocol to implement the ID update scheme.

7.3.4 MTS

MTS attempts to give high utilization (like EDF) while using the standard 11-bit ID format (like DM). MTS can be thought of as a cross between EDF and DM.

In DM, an 11-bit ID can represent 2048 messages. No realistic system will have this many different message streams, so that a few ID bits will remain unused. The goal is to use these bits to enhance schedulability. Suppose there are an equal number of high-speed and low-speed messages in a workload. If the high-speed ones have a ten times faster rate, they will use ten times more bandwidth than low-speed messages. This means that if we can increase the schedulability of just high-speed messages, we will get a large improvement in overall schedulability.

The idea behind MTS is to use EDF for high-speed messages and DM for low-speed ones. First, we give high-speed messages priority over low-speed and non-real-time ones by setting the most significant bit to 1 in the ID for high-speed messages (Figure 7.3a). This protects high-speed messages from all other types of traffic. If the uniqueness field is to be 5 bits [127] (allowing 32 high-speed messages), and the priority field is 1 bit, then the remaining 5 bits are still not enough to encode the deadlines (relative to the latest SOE). Our solution is to quantize time into *regions* and encode deadlines according to which region they fall in. To distinguish messages whose deadlines fall in the same region, we use the DM-priority of a message as its uniqueness code. This makes MTS a hierarchical scheduler. At the top level is EDF: if the deadlines of two messages can be distinguished after quantization, then the one with the earlier deadline has higher priority. At the lower level is DM: if messages have deadlines in the same region, they will be scheduled by their DM priority.

We can calculate length of a region (l_r) as $l_r = \frac{\ell + D_{max}}{2^m}$ where D_{max} is the longest relative

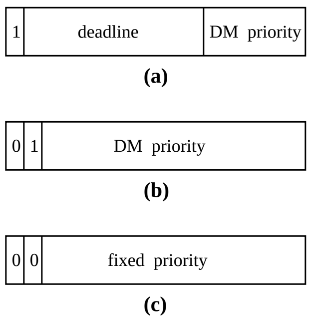

Figure 7.3: Structure of the ID for MTS. Parts (a) through (c) show the IDs for high-speed, low-speed, and non-real-time messages, respectively.

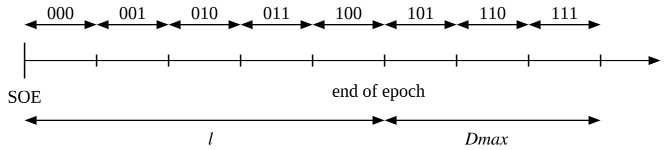

Figure 7.4: Quantization of deadlines (relative to start of epoch) for $m = 3$.

deadline of any high-speed message and m is the width of the deadline field (5 bits in this case). This is clear from Figure 7.4 (shown for $m = 3$). The worst-case situation occurs if a message with deadline D_{max} is released just before the end of epoch so that its absolute deadline lies $\ell + D_{max}$ beyond the current SOE. The deadline field must encode this time span using m bits leading to the above expression for l_r.

We use DM scheduling for low-speed messages and fixed-priority scheduling for non-real-time ones, with the latter being assigned priorities arbitrarily. The IDs for these messages are shown in Figures 7.3 (b) and (c) respectively. The second-most significant bit gives low-speed messages higher priority than non-real-time ones.

This scheme allows up to 32 different high-speed messages (periodic or sporadic), 512 low-speed messages (periodic or sporadic), and 480 non-real-time messages[1] — which should be sufficient for most applications.

7.3.5 ID Update Protocol

The IDs of all high-speed messages have to be updated at every SOE. One way to do this is to have a periodic (timer-driven) process at every node which wakes up every ℓ seconds and updates IDs of all ready messages. Because clocks are not perfectly synchronized, ID updates on different nodes may occur at slightly different times. This can cause priority inversion if the ID of a low-priority message is updated before that of a high-priority one.

[1]CAN disallows consecutive zeros in the six most significant bits of the ID. This means that 32 codes for non-real-time messages are illegal which leaves $512 - 32 = 480$ legal codes.

Then, for a small window of time, the low-priority message will have a higher priority ID than the high-priority message. To avoid this problem, we must use an agreement protocol to trigger the ID update on all nodes. The CAN clock synchronization algorithm [31] synchronizes clocks to within 20μs, so that the ID update processes on various nodes will wake up within 20μs of each other. A simple agreement protocol can be that one process is designated to broadcast a message on the CAN bus. This message will be received by all nodes at the same time (because of the nature of the CAN bus) and upon receiving this special message, all nodes will update the IDs of their local messages. But this protocol has two disadvantages. First of all, too much CAN bandwidth is wasted transmitting the extra message every ℓ seconds. Moreover, a separate protocol must be run to elect a new leader in case the old leader fails. Instead, we use the following protocol which is not only robust but also consumes less bandwidth. Upon activation, each ID update process takes the following actions:

1. Set a flag to inform the CAN device driver that the ID update protocol has begun.
2. Configure the CAN network adapter to receive all messages (by adjusting the receive filter).
3. Increment the data length (DL) field of the highest-priority ready message on that node.

After taking these actions, the process blocks on a timer till the next SOE.

The first incremented-DL message to be sent on the CAN bus will serve as a signal to all nodes to update the IDs of their messages. If the original DL of the message is less than 8, then incrementing the DL will result in transmission of one extra data byte (device drivers on receiving nodes strip this extra byte before forwarding the message to the application as described later). If the DL is already 8, CAN adapters allow the 4-bit DL field to be set to 9 (or higher) but only 8 data bytes are transmitted.

Now, each node starts receiving all messages transmitted on the CAN bus. The device driver on each node has a table listing the IDs of all message streams in the system along with their data lengths. As messages arrive, the device driver compares their DL field to the values in this table until it finds a message with an incremented DL field. All nodes receive this message at the same time and they all take the following actions:

1. Restore the receive filter to re-enable message filtering in the NA.
2. If the local message whose DL field was incremented by the periodic process has not been transmitted yet, then decrement the DL field back to its original value.

3. Update message IDs to reflect the new SOE.

Each node receives the incremented-DL message at the same time, so the ID update process on each node starts at the same time. After the first incremented-DL message completes, the next-highest-priority message begins transmission. As long as all nodes complete their ID updates before this message completes (a window of at least 55μs since this message contains at least one data byte), all messages will have updated IDs by the time the next bus arbitration round begins and no priority inversion will occur. In case one or more nodes are slow and cannot complete the ID update within this window of time, all nodes can be configured to do the update while the n^{th} message after the first incremented-DL message is in transmission, where n is a small number large enough to allow the slowest node to calculate all new IDs and then just write these to the NA while the n^{th} message is in transmission.

This protocol incurs a network overhead of 16 bits every ℓ seconds (compared to 47 bits per epoch for the simple leader-based agreement protocol). Reception of the first incremented-DL message causes the device drivers to set the DL fields of their local messages back to their original values, but before this can complete, the next transmission (also with an incremented DL field) has already started. These two messages have 8 extra data bits each (worst-case) which leads to the 16-bit overhead. On the CPU side, the periodic process incurs some overhead. Moreover, while the network adapter's filter is disabled, the device drivers must process two messages which may or may not be meant for that node. The device drivers must perform filtering in software and discard messages not meant for their node. Measurements of these various CPU overheads are in Section 7.5.3.

7.3.6 Schedulability Conditions

For MTS, we want off-line schedulability conditions which, when satisfied, will guarantee that all real-time messages will meet their deadlines. We will first review similar conditions for non-preemptive DM, and then develop those for MTS.

Deadline Monotonic

For the non-preemptive case, a message i is feasible if all higher-priority messages are feasible and message i finds an opportunity to start transmission sometime during $[0, D_i - C_i]$. If messages are numbered according to their priority with $j = 1$ being the highest-priority message, then i is schedulable [54] if:

$\exists t \in S, \quad \sum_{j=1}^{i-1} \lceil t/T_j \rceil C_j + C_p \leq t$, where S ={set of all release times of messages $1, 2, \cdots, i-1$ through time $D_i - C_i\} \cup \{D_i - C_i\}$; T_j, C_j, and D_j are the period, length,

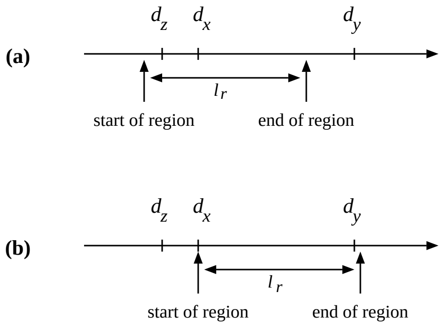

Figure 7.5: Suppose y has higher DM-priority than x but z does not. Then in (a), x has the highest priority, whereas in (b), it has the lowest.

and relative deadline of message j; and C_p is the length of the longest possible packet.

MTS

First, we will discuss the schedulability check for high-speed messages and then look at low-speed ones. The worst-case loading conditions for a high-speed message invocation x result when there is

1. worst possible traffic congestion, and

2. worst possible deadline encoding.

The first situation is created by releasing all messages at the same time $t = 0$ as long as $\sum_{j=1}^{n} C_j/T_j \leq 1$. The second occurs when deadline-to-start of x falls at the start of a region as illustrated in Figure 7.5.

Now, we can draw a parallel between schedulability conditions for MTS and those for DM. From the above discussion, note that invocation y has higher priority than invocation x (belonging to separate message streams) if y satisfies one of the following two conditions:

1. $(d_x - C_x) > (d_y - C_y)$, or

2. (a) $(d_x - C_x) < (d_y - C_y) \leq (d_x - C_x + l_r)$, and

   (b) DM priority of y is greater than that of x, and

   (c) y is released before $d_x - C_x$.

We can use these conditions to determine schedulability of some high-speed message stream i. We identify all invocations of message streams j which have higher priority than the first invocation of i and schedule them according to MTS. If the bus ever becomes idle during interval $[0, D_i - C_i]$, then stream i's first invocation will get a chance to run and i is feasible. Formally, a high-speed message stream i is schedulable under MTS if and only if $\sum_{j=1}^{n} C_j/T_j \leq 1$ and it satisfies the following condition:

$\exists t \in S, \ \ \sum_j \lceil t/T_j \rceil^{\star} C_j + C_p \ < \ t,$
where $j = \{1, 2, \ldots, i-1, i+1, \ldots, n\}$, S ={set of release times of each j through time $D_i - C_i$} $\cup\{D_i - C_i\}$, C_p is the size of a longest possible packet, and function $\lceil t/T_j \rceil^{\star}$ equals $\lceil t/T_j \rceil$ if last invocation of j released before t has higher priority than the first invocation of i; and equals $\lceil t/T_j \rceil - 1$ otherwise.

Checking schedulability of low-speed messages is simple — just check DM schedulability for each. Since high-speed messages have shorter deadlines than low-speed ones, they will automatically have higher DM priority (which is exactly what we want).

7.4 Implementation

In this section, we present schemes to implement MTS on Motorola's TouCAN module [88] which features 16 message buffers and internal arbitration between transmission buffers based on message ID. As such, TouCAN is representative of modern CAN NAs. We explore several inter-related implementation issues:

- Managing buffer space in the network adapter.
- Queuing and sequencing messages in host CPU.
- Controlling priority inversion.

In the following, we present a brief description of TouCAN, the problems faced when implementing real-time scheduling on CAN, and our solution to these problems for MTS.

7.4.1 Motorola TouCAN

TouCAN is a module developed by Motorola for on-chip inclusion in various microcontrollers. TouCAN lies on the same chip as the CPU and is interconnected to the CPU (and other on-chip modules) through Motorola's intermodule bus. Motorola is currently marketing the MC68376 [88] microcontroller which incorporates TouCAN with a CPU32 core.

TouCAN has 16 message buffers. Each buffer can be configured to either transmit or receive messages. When more than one buffers have valid messages waiting for transmission, TouCAN picks the buffer with the highest-priority ID and contends for the bus with this ID. In this respect TouCAN differs from older CAN network adapters such as the Intel 82527 [45] which arbitrate between buffers using a fixed-priority, daisy-chain scheme which forces the host CPU to sort messages according to priority before placing them in the network adapter buffers. This was one of the main reason we picked TouCAN for implementing MTS.

7.4.2 TouCAN Device Emulation

At this time, TouCAN is available only with the MC68376 microcontroller. To implement MTS within EMERALDS on TouCAN, we would first have to port EMERALDS to the MC68376 microcontroller. To avoid this, we instead used device emulation [42] under which a general-purpose microcontroller is made to emulate a network adapter. This emulator interfaces to the host CPU through an I/O bus. The emulator presents the host CPU the same interface that the actual network adapter would. The emulator receives commands from the host CPU, performs the corresponding actions, and produces the same results that the actual network adapter would.

Network adapter emulation presents two advantages over using the actual network hardware. First, design time is reduced significantly since software does not have to be ported across platforms. Moreover, software testing is easier with an emulator than with an actual network adapter because the emulator can be easily made to emulate network conditions that may be unpredictable or difficult to reach with a real network (such as various transmission and reception related events, error conditions, etc.).

The disadvantage of network emulation is that overhead measurements are not exact. The MTS implementation overheads we are interested in are:

1. Interrupt handling, context switching, and other associated operating system overheads.

2. Message queuing on host CPU.

3. Data transfer between host CPU and network adapter.

With network emulation, the first two overheads can be measured exactly since they depend solely on the host CPU and the operating system. The third element will have some inaccuracy, but since for CAN, only 6–14 bytes per message (including message ID) are transferred between the host and the adapter, this inaccuracy should be insignificant.

We use a 68040 board to emulate the TouCAN module and connect it to the host CPU (another 68040) through a VME bus. A TouCAN-emulating C++ program runs on the emulator. Through the VME bus, it presents the same memory-mapped interface to the host that a TouCAN module would. The emulator software executes an infinite loop to check various emulated command "registers." Whenever the host modifies any of these registers, the emulator takes the appropriate actions and interrupts the host through the VME bus if needed.

7.4.3 Problems In Implementing Message Scheduling On CAN

In implementing MTS on CAN, our goal is to minimize the average overhead suffered by the host node for transmitting a message. This overhead has the following components:

1. Queuing/buffering messages in software if network adapter buffers are unavailable.

2. Transferring messages to network adapter.

3. Handling interrupts related to message transmission.

We do not consider overheads related to protocol processing because applications using CAN usually do not need to use protocol stacks; instead they communicate directly with the CAN device driver for efficiency reasons. Moreover, such overheads are heavily OS-dependent.

To understand the difficulties involved in reducing message transmission overhead, it serves to compare CAN to other LAN protocols such as token-based schemes (token-ring, FDDI, etc.). In the latter, the network becomes available periodically (whenever the local node has the token) at which time, the network adapter can transmit messages in its buffer. Priority inversion can occur if adapter buffers are filled with low-priority messages. If a high-priority message arrives at this point, it has to be buffered in software to wait for one or more messages already in the adapter to be sent. But this priority inversion is bounded by the token rotation time and number of buffers in the adapter [43].

In CAN, priority inversion can be unbounded. If the adapter buffers contain low-priority messages, these messages will not be sent as long as there are higher-priority messages anywhere else in the network. Consequently, a high-priority message can stay blocked in software for an indeterminate period of time, causing it to miss its deadline. Because of this priority inversion problem, any network scheduling implementation for CAN (regardless of which scheduling policy — DM or MTS — is being implemented) has to ensure that adapter buffers always contain the highest-priority messages and only lower-priority messages are queued in software.

7.4.4 MTS on TouCAN

Suppose B buffers are allocated for message transmission (usually B is about two-thirds of the total number of buffers; see Section 7.5.3). If the total number of outgoing message streams is B or less, then MTS's implementation is straight-forward: assign one buffer to each stream. Whenever the CAN device driver receives a message for transmission, it simply copies that message to the buffer reserved for that stream. In this case, no buffering is needed within the device driver which also means that there is no need for the CAN adapter to generate any interrupts upon completion of message transmission[2], and this leads to the lowest-possible host CPU overhead.

When number of message streams exceeds B, some messages have to be buffered in software. To reduce host CPU overhead, we want to buffer the fewest possible messages while avoiding priority inversion. Just as MTS treats low-speed and high-speed messages differently for scheduling purposes, we treat these messages differently for implementation purposes as well. Our goal is to keep the overhead for frequent messages (those belonging to high-speed periodic streams) as low as possible to get a low average per-message overhead. In our implementation, if the number of periodic high-speed message streams N_{Hp} is less than B, then we reserve N_{Hp} buffers for high-speed periodic streams and treat them the same as before (no buffering in software).

The remaining $L = B - N_{Hp}$ buffers are used for high-speed sporadic, low-speed, and non-real-time messages. As these messages arrive at the device driver for transmission, they are inserted into a priority-sorted queue. To avoid priority inversion, the device driver must ensure that the L buffers always contain the L messages at the head of the queue. So, if a newly-arrived message has priority higher than the lowest-priority message in the buffer, it "preempts" that message by overwriting it. This preemption increases CPU overhead but is necessary to avoid priority inversion. The preempted message stays in the device driver queue and is eventually transmitted according to its priority.

Among these L buffers, the buffer containing the $I + 1^{th}$ lowest priority message is configured to trigger an interrupt upon message transmission (I is defined later). This interrupt is used to refill the buffers with queued messages. I must be large enough to ensure that the bus does not become idle while the interrupt is handled and buffers are refilled. Usually an I of 1 or 2 is enough (which can keep the bus busy for 47–94 μs minimum). Note that this puts a restriction on L that it must be greater than I. Making L less than or equal to I can lead to the CAN bus becoming idle while the ISR executes, but makes more buffers available for high-speed periodic messages. This can be useful if low-speed messages make up only a small portion of the workload and high-speed sporadic messages are either non-existent or very few. This tradeoff is discussed in more detail in Section 7.5.4.

If $N_{Hp} \geq B$ then we must queue even high-speed periodic messages in software. Then we have a single priority-sorted queue for all outgoing messages and all B buffers are filled from this queue.

[2]The CAN adapter must be programmed to generate interrupts if messages are queued in software waiting for adapter buffers to become available, which is not the case here.

Overheads

For streams with dedicated buffers, the CPU overhead is just the calculation of the message ID and transferring the message data and ID to the network adapter. Note that message data can be copied directly from user space to the network adapter to keep overhead to a minimum.

For messages which are queued in software, there is an extra overhead of inserting the message in the queue (including copying the 8 or fewer bytes of message data from user space to device driver space before inserting in the queue), plus the overhead for handling interrupts generated upon message transmission. This interrupt overhead is incurred once every $Q - I$ message transmissions, where Q is the number of buffers being filled from the queue (Q can be B or L depending on whether high-speed periodic messages are buffered or not). Also, each message will potentially have to preempt one other message. The preempted message had already been copied to the network adapter once and now it will have to be copied again, so the preemption overhead is equivalent to the overhead for transferring the message to the network adapter. Table 7.1 summarizes the overheads for various types of messages. Measurements of these overheads are in Section 7.5.

Message type	*Overhead*
Not queued	Calculate ID + copy to NA
....	....
Queued	Calculate ID + insert in priority queue + copy to NA + preempt + interrupt/$(Q - I)$

Table 7.1: Summary of overheads for MTS's implementation on TouCAN.

Note that DM scheduling also incurs similar overheads. The only difference is that the ID of message streams under DM is fixed, so a new ID does not have to be calculated each time. Other than that, implementing DM on TouCAN is no different than implementing MTS.

7.4.5 Preemption as a Mechanism for Controlling Priority Inversion

In CPU scheduling, priority inheritance [101] is a well-known mechanism for handling priority inversion between threads of execution. If a low-priority thread holds a resource needed by a high-priority thread, the former's priority is temporarily increased to that of the latter until the resource is released.

For CPU scheduling, priority inheritance is feasible because resources (such as a critical section) are usually held for short periods of time compared to the overall execution times of threads. Temporarily elevating thread priorities for these brief durations does not hurt schedulability much.

But for network message transmission, a message holds the resource under contention (i.e., the network adapter buffer) until it completes transmission. Using priority inheritance in this situation can lead to a significant schedulability degradation (see Section 7.5.5).

Another technique for tackling priority inversion in CPU scheduling is preempt-and-restart [114] in which the resource holder is preempted and forced to restart later from the beginning of the critical section. The disadvantage of this scheme is that once a thread is preempted, all CPU work it had done since entering the critical section is lost.

Our network preemption scheme is similar to preempt-and-restart, but unlike CPU scheduling, no "network work" is lost due to preemption (although some extra CPU overhead is incurred). This makes preempt-and-restart more attractive for network scheduling than for CPU scheduling because the only cost is the preemption overhead.

Preemption Overheads in Various Networks

Priority inversion within the network adapter is a problem in all shared-access networks. In a network such as CAN where message sizes are small, preemption costs are low, so preemption-and-restart can be used effectively. But for networks such as FDDI and Ethernet which have large packet data units in the kilobytes, it would appear that preemption is infeasible, but in reality that is not the case. Most network adapters (such as LANCE [4] for Ethernet) do not keep message data in network adapter memory. Instead, data is kept in host memory and the network adapter is provided with pointers to this data and it uses DMA to transfer data as needed. If this is the case, preemption can be used to address the priority inversion problem in these networks as well. Previous solutions to reduce priority inversion for such networks include schemes which limit the number of FIFO buffers used for transmission [43]. This reduces priority inversion but also increases the frequency of interrupts. If preemption is used, the maximum allowable FIFO queue length can be used which reduces interrupt overhead to a minimum; this at the low cost of overwriting a few memory locations containing pointers to message data.

7.5 Results

In this section we first evaluate the schedulability performance of MTS as compared to ED* and DM. ED* is an imaginary scheduling policy which works the same as EDF but requires only an 11-bit ID. We would expect MTS's performance to lie between those of ED* and DM. To check schedulability under ED*, we use the schedulability check for non-preemptive EDF in [121].

Our measurements show that performance of MTS depends upon various workload characteristics. We identify the conditions under which MTS performs well and show that these conditions are typical of control applications.

We also measure the various overheads related to implementation of MTS and present results to justify certain implementation-related design decisions.

7.5.1 Workload Model

To compare the schedulability performance of ED*, MTS, and DM, we generate workloads with different characteristics and test their schedulability under each of the three scheduling policies. Unless stated otherwise, workloads are generated as follows.

Each workload has 8–15 high-speed periodic streams, 2 high-speed sporadic streams (13–25% of the number of periodic streams), 25 low-speed periodic streams, and 4 low-speed sporadic streams. Deadlines of high-speed messages are set randomly in the 0.5–2ms range while those for low-speed messages are set randomly between 2–100ms. Periods of periodic messages are calculated by adding a small random value to the deadline, while MIT of sporadic streams is set to 2s (for both low-speed and high-speed sporadic streams). Length of epoch is 2ms.

Workloads for a particular experiment are generated by changing one of the above parameters (such as deadlines of low-speed messages, number of high-speed sporadic streams, and ℓ). For each experiment, different data points are obtained by varying the number of high-speed periodic streams from 8 to 15 which leads to a variation in workload utilization roughly in the 50–100% range. For each data point, we generate 1000 workloads (with a fixed number of high-speed periodic streams) and measure the percentage found feasible under ED*, MTS, and DM. Increasing the number of workloads beyond 1000 did not produce any significant variation in measured results.

All results for MTS include the overhead resulting from 16 extra bits per epoch for ID updates.

7.5.2 Schedulability Comparisons

MTS was designed based on the premise that certain messages have relatively short periods/deadlines (high-speed messages) while others have relatively long periods/deadlines (low-speed messages). Figures 7.6–7.8 show that MTS goes from being close in performance to DM (Figure 7.6) to being close in performance to ED* (Figure 7.8) as deadlines of high-speed and low-speed messages become more and more differentiated.

In Figure 7.6, low-speed messages have relatively tight deadlines (2–50ms), so MTS's performance is closer to that of DM than to that of ED*. Our measurements showed that

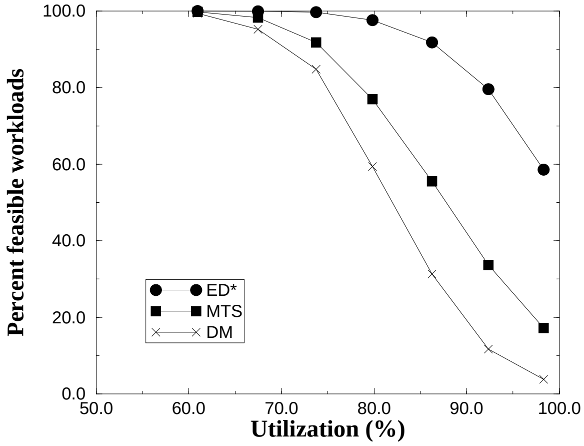

Figure 7.6: Schedulability when deadlines of low-speed messages are set in the 2–50ms range.

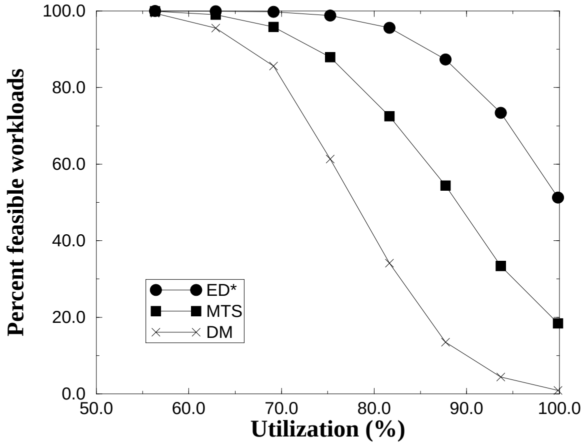

Figure 7.7: Schedulability when deadlines of low-speed messages are set in the 2–100ms range.

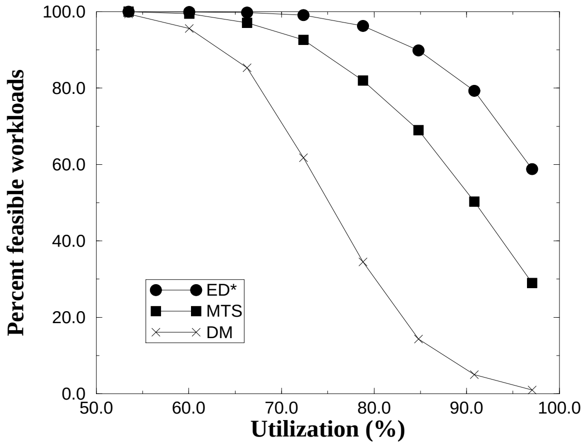

Figure 7.8: Schedulability when deadlines of low-speed messages are set in the 2–200ms range.

at the 86% utilization mark, 89% of the workloads infeasible under MTS were infeasible because of low-speed messages. In other words, when the low-speed portion of the workload is "difficult" to schedule (because of tight deadlines or any other reason), MTS's performance is significantly worse than that of ED*, although it is still much better than DM because MTS schedules high-speed messages more successfully than DM. In fact, MTS feasibly schedules about 20 percentage points more workloads than DM for workload utilizations of 80–100%.

In Figure 7.7, low-speed messages have deadlines in the 2–100ms range. This means that low-speed messages are now a smaller portion of the workload (utilization-wise) which results in a corresponding increase in performance of MTS. It can now feasibly schedule 25–40 percentage points more workloads than DM for workload utilizations of 75–100%. This trend continues in Figure 7.8 in which low-speed messages have deadlines ranging from 2ms to 200ms and performance of MTS is close to that of ED*. For these workload characteristics, at the 85% utilization mark, 63% of the workloads infeasible under MTS are infeasible because of low-speed messages. Because of this improved schedulability of low-speed messages, MTS feasibly schedules 30–45 percentage points more workloads than DM for workload utilizations of 70–100%.

Clearly, performance of MTS relative to ED* and DM is workload-dependent. MTS performs well when all (or most) low-speed messages have deadlines several times larger than those of high-speed messages. When we look at workloads of typical real-time control applications, we find that there is indeed a great variation between periods of various tasks (and a corresponding variation in periods/deadlines of messages sent by these tasks). The well-known avionics task workload [77, 78] is accepted as typifying real-time control applications and also has been used by others for research on real-time scheduling [56]. The workload is reproduced in Table 7.2. It has 6 tasks with deadlines in the 5–50ms range (i.e., high-speed tasks) and 11 tasks with deadlines in the 59–1000ms range (low-speed tasks). On this basis, we would expect most real-world communication workloads to conform to the basic premise behind the design of MTS (i.e., high-speed messages have relatively tight deadlines and low-speed messages have relatively long deadline), leading to good performance in practical real-time control applications.

Execution time (ms)	*Deadline (ms)*	*Period/MIT (ms)*
3	5	200
2	25	25
5	25	25
1	40	40
3	40	40
5	50	50
8	59	59
9	80	80
2	80	80
5	100	100
1	200	200
1	200	200
1	200	200
3	200	200
3	200	200
1	1000	1000
1	1000	1000

Table 7.2: Avionics task workload.

Dependence on Sporadic Streams

For the remaining tests, we use workloads with low-speed messages having deadlines in the 2–100ms range. Figures 7.9–7.10 show the effect of varying the number of high-speed sporadic streams. Figure 7.9 is based on workloads with no high-speed sporadic streams while the workloads in Figures 7.10 contain 4 high-speed sporadic streams (27–50% of the number of high-speed periodic streams). These figures show that as the sporadic component of the workload increases, DM suffers a sharp decline in performance. MTS's performance also drops but not as much as that of DM. This affirms that deadline-based schemes are more capable of handling sporadic messages. Increasing the number of sporadic messages increases the load at the critical instant (using Liu and Layland's terminology [76]), even though overall workload utilization is almost unchanged. With so many tight-deadline messages released at the same time, prioritizing messages based on deadlines results in a

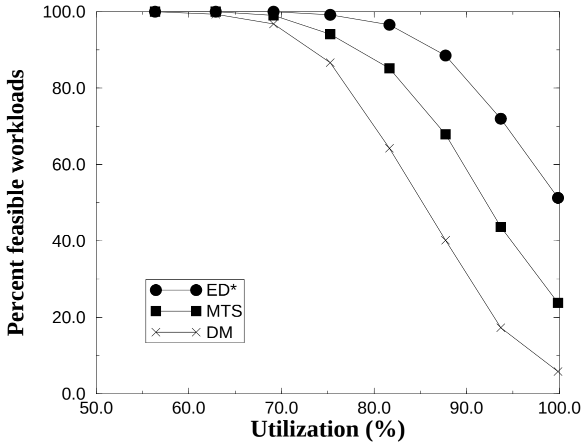

Figure 7.9: Schedulability when number of high-speed sporadic streams is 0.

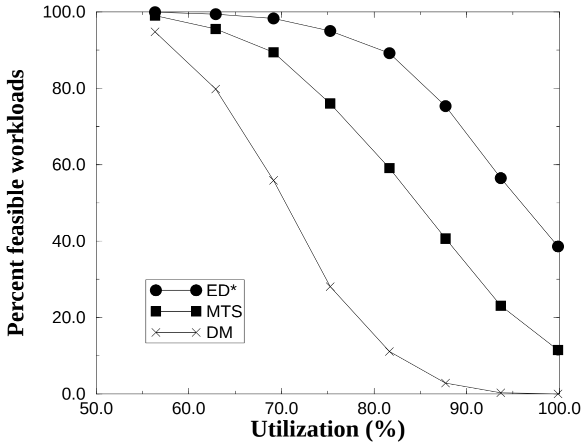

Figure 7.10: Schedulability when number of high-speed sporadic streams is 4.

significant improvement in performance over fixed-priority scheduling.

7.5.3 CPU Overheads

The overhead measurements for implementation of MTS on a 25MHz Motorola 68040 with the EMERALDS RTOS are in Table 7.3. These measurements are with caches turned off since automotive controllers typically do not use caches (to reduce cost, increase predictability, and to fit more I/O devices on-chip).

From this data, we see that high-speed messages with dedicated network adapter buffers incur an overhead of

$$\text{ID calculation} + \text{transfer to NA} + \text{misc.} = 16.8\mu\text{s/msg}.$$

Operation	*Overhead* (μs)
Calculate ID (high-speed messages)	3.0
Insert in priority queue (including copying to device driver memory)	$6.3 + 1.55 l_Q$
Transfer message to NA (8 data bytes)	7.8
Preempt message	7.8
Interrupt handling and dequeuing of transmitted messages	42.4
Miscellaneous (parameter passing, etc.)	6.0

Table 7.3: CPU overheads for various operations involved in implementing MTS.

If high-speed periodic messages are queued, then average per-message overhead depends on the number of buffers used for transmission (Q). TouCAN has 16 buffers. Of these, 5–6 are usually used for message reception and their IDs are configured to receive the various message streams needed by the node. This leaves about 10 buffers for message transmission. Then, under worst-case scenario, message transmission incurs an average overhead (assuming $I = 2$):

$$\text{ID calculation} + \text{queuing} + \text{preempt} + \text{transfer to NA} + \frac{\text{interrupt}}{Q - I} + \text{misc.} = \\ 36.2 + 1.55 l_Q \ \mu\text{s/msg},$$

where the worst-case l_Q is the total number of message streams using that queue. Low-speed and non-real-time messages have fixed IDs, so they incur an overhead of $33.2 + 1.55 l_Q$ μs/msg if all low-speed and high-speed messages share the same queue.

If high-speed messages are using dedicated buffers, then $Q - I$ is smaller for low-speed messages. Assuming only 3 buffers are available and $I = 2$, then low-speed and non-real-time messages incur overheads of $70.3 + 1.55 l_Q$ μs/msg while high-speed sporadic messages have overheads of $73.3 + 1.55 l_Q$ μs/msg.

From these numbers we see that if a certain node has 7 high-speed periodic streams, 1 high-speed sporadic stream, 10 streams of low-speed and non-real-time messages, and if the high-speed periodic messages make up 90% of the outgoing traffic while $Q - I = 1$ for high-speed sporadic/low-speed/non-real-time messages, then average per-message overhead comes to $(16.8)(0.9) + (70.3 + 1.55(11))0.1 = 23.9\mu$/msg. Overhead is significantly higher if the number of high-speed periodic streams is large enough that high-speed messages have to be queued. In that case, per-message overhead can be twice as much as the overhead when high-speed periodic streams have dedicated buffers. Fortunately, real-time control applications do not have more than 10–15 tasks per node (the avionics task workload is an example). Not all tasks send inter-node messages and those that do typically do not send more than 1–2 messages per task. This indicates that for most applications, dedicated buffers should be available for high-speed message streams, resulting in a low per-message overhead in the 20–25μs range.

We used a simple linked list to sort messages in the priority queue. This works well for a small number of messages (5–10) that typically need to be in the queue. For larger number of messages, a sorted heap will give lower overhead.

Note that these overheads are applicable to DM as well. Only difference is that under DM, the ID does not have to be calculated, so per-message overhead will be 3μs less than for MTS.

ID Re-adjustment at End of Epoch

Table 7.4 lists the CPU overheads incurred during the ID update protocol. Overhead for the periodic task includes all context switching and CPU scheduling overheads. One context switch occurs when the task wakes up and another when the task blocks. Both of these are included in the overhead measurements.

Operation	*Overhead (μs)*
Periodic task	68.0
Device driver interrupt (message arrival)	40.4
Read message from NA (8 data bytes)	7.8
Software filtering and DL lookup	3.0
ID update	2.8 per message

Table 7.4: CPU overheads for various operations involved in updating message IDs.

During each ID update, the device driver receives two messages (each incurring an overhead of $40.4 + 7.8 + 3.0 = 51.2\mu s$ including all context switching overheads). After receiving the first message, IDs of high-speed messages are updated. Assuming IDs of 5 messages need to be updated, the total overhead per epoch becomes 184.4μs. If ℓ = 2ms, the ID update takes up about 9% of CPU time. This motivates us to increase ℓ.

Increasing ℓ increases the level of quantization of deadlines which results in reduced schedulability for high-speed messages. But on the other hand, the network overhead associated with ID updates (16 bits per epoch) decreases, leading to increased schedulability. For ℓ = 2ms, 16 extra bits per epoch consume only 0.8% of the network bandwidth for a 1Mb/s bus, but their impact on network schedulability (due to their blocking effect) is much higher. Our measurements showed that with this extra overhead, about 2–3 percentage points fewer workloads are feasible under MTS (for the same workload utilization)

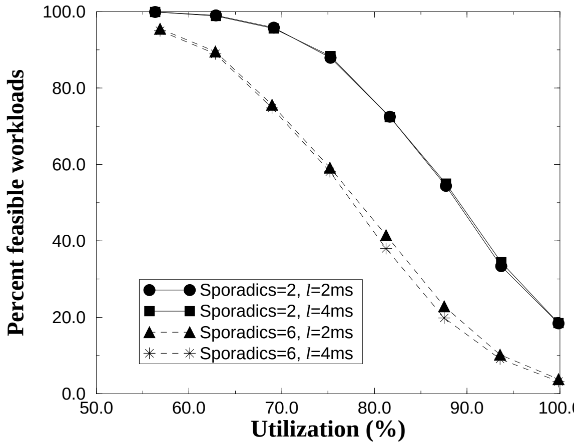

Figure 7.11: Impact of changing ℓ on MTS schedulability.

than without this overhead. As such, increasing ℓ can result in a sizeable improvement in schedulability due to reduced ID update overhead which can offset the loss in schedulability due to coarser quantization.

Figure 7.11 shows the effect of increasing ℓ on schedulability. It shows that when ℓ is doubled from 2ms to 4ms, network schedulability is actually improved slightly when two high-speed sporadic streams are in the workload. But when six sporadic streams are used, loss in schedulability from coarser quantization is more than the gain from reduced ID update overhead, so that 1–2 percentage points fewer workloads are feasible. These results show that for light-to-moderate high-speed sporadic loads, increasing ℓ to 4ms continues to give good performance, and even for heavy high-speed sporadic loads, ℓ = 4ms results in only a slight degradation in performance.

If ℓ is increased to 3ms, then the ID update CPU overhead reduces to about 6% of CPU time, whereas for ℓ = 4ms, it becomes 4.6% of CPU time.

7.5.4 Varying L

In Section 7.4.4 we mentioned that reducing the number of buffers used for low-speed messages (L) to less than $I + 1$ can be beneficial since it makes more buffers available for high-speed periodic messages. If L is less than $I + 1$, then the CAN bus can become idle while the CPU refills the network adapter buffers. Under worst-case scenario, the bus becomes idle after the transmission of each high-speed sporadic and low-speed message. We measure the impact of this effect by increasing the length of each of these messages by the duration by which the bus was idle. The results are in Figure 7.12. We use $I = 2$ for these experiments, meaning that L should be at least 3 to prevent the bus from becoming idle.

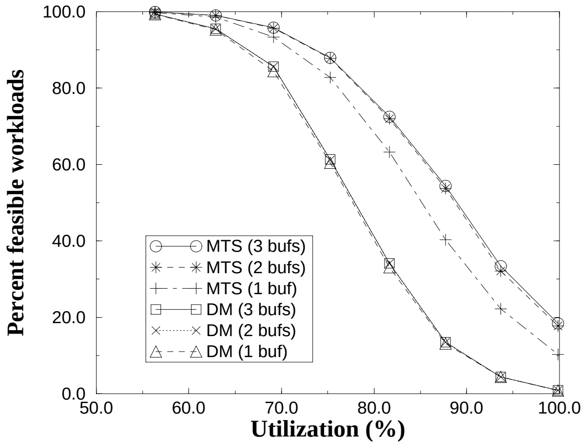

Figure 7.12: Schedulability when number of buffers for low-speed messages is decreased below $I + 1$ ($I = 2$ for these experiments).

Reducing L to 2 can idle the bus for one message length (47μs) worst-case and reducing it to just 1 will make the bus idle for 94μs.

The results show that reducing L by one has no significant impact on schedulability, but performance of MTS drops significantly when L is reduced any further. DM is more robust in this matter with no significant change in performance even when L is reduced to just 1.

7.5.5 Using Priority Inheritance

It is difficult to determine what the schedulability conditions for MTS will be if priority inheritance is used (instead of preemption) to control priority inversion between outgoing messages on the same node. The difficulty is in determining the worst-case scenario. Releasing all messages at the same time may not lead to the worst-case situation since only those messages in the network adapter at the time of the release will inherit priorities. Releasing messages with some phase offsets will cause more messages to inherit priorities but will not lead to the worst-case loading of the bus.

Actually, we do not need to know the exact worst-case situation to show that priority inheritance does not perform as well as preemption for CAN scheduling. It is clear that the actual worst-case situation must be same as or worse than:

- All messages released at the same time.
- Only the highest-priority message suffers any blocking due to priority inheritance. It can stay blocked for as much as 111μs (longest message transmission time).

Figure 7.13 shows the results under this situation. It clearly shows that priority inheritance

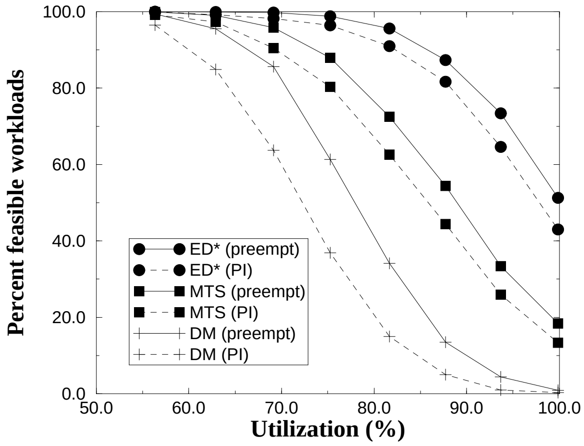

Figure 7.13: Impact on schedulability of using priority inheritance (PI) instead of preemption.

is not a suitable policy for handling priority inversion in CAN network adapters because as much as 10 percentage points fewer workloads are feasible under MTS when priority inheritance is used, and even worse for DM.

7.6 Conclusion

The CAN standard message frame format has an 11-bit ID field. If fixed-priority scheduling (such as DM) is used for CAN, some of these bits go unused. The idea behind MTS is to use these extra bits to enhance network schedulability. MTS places a quantized form of the message deadline in these extra bits while using the DM-priority of messages in the remaining bits. This enhances schedulability of the most frequent messages in the system (high-speed messages) so that MTS is able to feasibly schedule 20–40 percentage points more workloads than DM.

Since message IDs are based on deadlines, they must be periodically updated. We presented a protocol to perform this update without any priority inversion. This protocol consumes about 5–6% of CPU time, but considering the large improvements in network schedulability that MTS displays over DM, this extra overhead is justified.

We also presented a scheme to implement MTS on the TouCAN network adapter which is representative of modern CAN network adapters. The biggest challenge in implementing CAN scheduling (be it MTS or DM) is controlling priority inversion within the network adapter. We showed that because of CAN's characteristics (short message size), preemption of a message in the adapter by a newly-arrived, higher-priority outgoing message is an effective method for avoiding priority inversion. The alternative — priority inheritance —

is not feasible because the extra blocking suffered by messages causes a sizeable loss in network schedulability.

Performance of MTS depends in part on the value of ℓ. A large ℓ reduces both CPU overhead as well as network overhead related to ID updates, but increasing ℓ too much can hurt schedulability when deadline quantization becomes too coarse. For future work, a software tool needs to be designed which can analyze a particular workload and determine the best value of ℓ for it, so that MTS can deliver optimal performance by giving high network schedulability while keeping CPU overhead to a minimum.

Another avenue of research is to study message reception issues for CAN to try to reduce the average per-message reception overhead. Unlike message transmission, message reception does not depend on which network scheduling policy (DM or MTS) is used. Message reception overheads can be reduced by optimizing interrupt handling, using polling (instead of interrupts) to detect message arrival, or using a combination of interrupts and polling.

CHAPTER 8

CONCLUSIONS

The computing world is no longer limited to expensive desktops, workstations, and PCs. Fairly sophisticated consumer item embedded systems are becoming a part of our everyday life. These devices run our cars, control our smart information gadgets, and automate our homes. With annual production volumes reaching tens of millions of units, these embedded systems are now an important class of computation devices.

Whereas embedded systems of the past were simple microcontrollers running a few tasks written in assembly or C, the embedded systems of today tend to be networked, run application code written in object-oriented (OO) languages such as Java, execute an increasing number of complex tasks, and need real-time OS (RTOS) support — either to handle audio/video or to interact with the environment. The challenge is to provide all these OS services while keeping overheads to a minimum and without putting extra burden on the application programmer. An efficient OS enables low-cost hardware to be used in consumer products which lowers per-unit costs and makes the product attractive for customers. This thesis dealt with developing low-overhead solutions for high-utilization task scheduling, support for OO programming (in the form of efficient semaphores), and real-time networking (protocol architecture and network scheduling) for embedded systems. We were able to show that these OS services can be made efficient without sacrificing flexibility, restricting OS APIs, or relying on any special hardware features.

We now summarize the primary contributions of this dissertation and suggest avenues for future research.

8.1 Research Contributions

This dissertation focused on key OS services which must be optimized for good overall performance in embedded systems. Primary contributions made in this regard are as follows:

- As a first step, we designed the EMERALDS RTOS to serve as the platform for the rest of our research. EMERALDS has highly optimized context switching, interrupt handling, and system call mechanism. The kernel is mapped into each address space, so that system calls are reduced to a TRAP followed by a subroutine call. This is done without any hardware support (other than a simple page-table MMU) by using characteristics of embedded systems. EMERALDS also features optimized local (intra-node) message passing using state messages. We used EMERALDS as a platform for implementing and evaluating the various optimizations we developed for scheduling, semaphores, and communication as discussed next.

• Task scheduling can take up 5–15% of CPU time (especially for relatively slow automotive controllers). This overhead has two components: run-time overhead and CPU utilization being less than 100%. Dynamic schedulers like earliest-deadline first (EDF) give high utilization but incur high run-time overhead. Static schedulers like rate-monotonic (RM) have low run-time overhead but give low utilization. We designed the *combined static/dynamic* (CSD) scheduler which splits tasks into two groups: one scheduled by EDF and the other by RM. Critical to good performance of CSD is proper assignment of tasks to the two groups. We developed an iterative method for partitioning tasks. With this grouping of tasks, CSD incurs run-time overhead comparable to RM while delivering schedulable utilization comparable to EDF. We implemented CSD in EMERALDS and experimental measurements show that it can feasibly schedule more workloads than EDF or RM through a reduction in scheduling overhead of as much as 40% compared to EDF and RM.

• Networked embedded systems are able to download code (such as Java applications) and execute them on-demand. The advent of Java has made OO programming important for embedded systems. In OO programming, updates to the state variables of objects have to be protected through semaphores to ensure mutual exclusion, and this represents significant run-time overhead. We developed a new priority inheritance semaphore implementation scheme which saves one context switch per semaphore lock operation in most circumstances, thus reducing semaphore overheads by 20–30%. This is achieved by a combination of OS mechanisms and an off-line instrumentation of application code by an automatic code parser. The parser inserts hints for the OS in the application code which allow elimination of context switches at run-time.

• Many embedded applications today require Internet connectivity (such as Internet telephones and webTV). Network bandwidth delivered to these applications is limited by host protocol processing overheads, especially on the receive side. We designed an architecture for reducing receive-side overhead for processing real-time audio and video messages by exploiting the periodic nature of such messages. All protocol processing for real-time traffic is performed by the application threads which ensures predictability. Moreover, overhead for short messages (such as live voice) is reduced by safely bypassing multiple protocol layers, greatly reducing I-cache misses. Also, message data is left in the network adapter buffers until the application makes a receive call so that data needs to be copied only once (without any hardware support from the network adapter or any restrictions on the network API). This is possible since the real-time scheduler guarantees an execution period for the audio/video applications which in turn ensures that data is not left in the network adapter for more than a known maximum time interval. We implemented UDP/IP using this architecture within EMERALDS and demonstrated its ability to efficiently and predictably handle short messages (such as live voice) as well as long ones (such as video and streaming data). Processing overhead for short messages was reduced 14–20% while that for long messages was reduced 15–22%.

• For embedded applications which require multiple controllers within a system (such as a car or a manufacturing workcell) to be interconnected by a LAN, we designed a network scheduling scheme for the Controller Area Network (CAN), which is a popular LAN for automotive and factory automation applications. Pure EDF scheduling of messages is not useful for CAN: packets have only 8 bytes of payload so that including a deadline with each packet results in unacceptable overhead. Fixed-priority deadline-monotonic (DM) scheduling needs fewer bits to express priorities but it yields relatively low utilization. We designed a scheduler called the *mixed traffic scheduler* (MTS) which combines EDF and DM using quantized deadlines and a moving time frame of reference. Packets are scheduled based on deadlines if deadlines are distinguishable after quantization; otherwise they are scheduled using DM priorities. Not only is MTS feasible for CAN (as demonstrated by its implementation within EMERALDS), it can also feasibly schedule 20–40 percentage points more workloads than DM.

8.2 Future Work

As use of embedded systems (specially IAs) becomes more widespread, several important problems will have to be addressed as briefly discussed next.

Resource redistribution: One avenue of future research is to explore mechanisms to allow the OS to reclaim resources (such as CPU time and memory) granted by the OS to applications. This should be done without causing the application to crash or malfunction and with minimal change in the way applications are written right now (i.e., application developer should not be overly burdened). The need for such reclamation mechanisms will arise in network-connected IAs which handle diverse functions such as telephony, web browsing, e-mail, etc. These small, portable devices will have comparatively scarce hardware resources.

The idea is to give applications all the resources they need if resources are available. Then, if an urgent task needs to be started (such as handling a phone call), the OS should be able to redistribute resources to allow the new task to execute. At first, reclaiming CPU time seems easy, but this is not true for real-time applications which require guaranteed CPU resources to function correctly. Burdening applications with having to constantly re-negotiate resource usage with the OS is an unreasonable solution. But if the OS can control the execution time of real-time applications (e.g., by controlling the network data flowing to audio/video applications) then CPU time can be easily reclaimed. Similar techniques can be used to reclaim memory and other resources.

Communication support for portable devices: The current paradigm for a portable device (such as a PDA) to connect to the Internet is to use a cellular phone to connect to a fixed Internet host which provides access to the rest of the Internet. Even mobile IP [105] relies on a *home agent* (which is an IP host residing on the home subnetwork of the mobile host) to forward IP packets to the mobile host. These extra hops increase network traffic. But if the Internet and the telephone worlds are to merge, then cell stations themselves will be Internet hosts. Then, as the mobile device moves and switches cells, the new cell should now be its gateway to the rest of the Internet. It would be interesting to investigate protocols to re-route data streams so that data is automatically directed to the correct cell/host as a portable device moves from one cell to another. For audio/video streams, the re-routing must also be performed in real-time.

Application Programming Interfaces for IAs: The mechanisms employed in hard real-time control systems to allow applications to communicate their execution and communication requirements to the RTOS do not fit the needs of IAs. For example, the traditional task scheduling paradigm requires the application to provide its worst-case execution time, period, and deadline to the scheduler. But in the dynamic operating environment of IAs, the worst-case execution time of applications will probably not be known. This is specially true for Java code because the application designer has no idea which processors the code may run on, so no *a priori* worst-case execution time measurement is possible. As such, new APIs must be developed which allow the applications to state their resource requirements in a simple way and maybe even omit certain pieces of information (which is then dynamically determined as the application executes).

Scalable servers for IAs: Currently, servers are designed to maximize throughput they can supply to relatively few concurrent clients. As IAs begin to proliferate, the servers must be designed to scale well not only in delivered bandwidth but also in the ability to concurrently support millions of clients. One example would be an Internet radio station. Such a server may have to service millions of clients although bandwidth requirements of each client may not be that high. Techniques which can be used to enable such services include aggregation of flows, multicast, and hierarchical organization of the server.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight multicast for real-time process groups," in *Proc. Real-Time Technology and Applications Symposium*, pp. 250–259, June 1996.

[2] T. Abdelzaher and K. G. Shin, "Optimal combined task and message scheduling in distributed real-time systems," in *Proc. Real-Time Systems Symposium*, pp. 162–171, 1995.

[3] F. Adelstein and M. Singhal, "Real-time causal message ordering in multimedia systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 36–43, 1995.

[4] *Am79C90 CMOS Local Area Network Controller for Ethernet (C-LANCE)*, Advanced Micro Devices, Inc., 1994.

[5] *Appliance war could make web less open.* News Briefs, IEEE Computer, vol. 30, no. 10, pp. 20–25, October 1997.

[6] A. C. Audsley, A. Burns, and A. J. Wellings, "Deadline monotonic scheduling theorey and application," *Control Engineering Practice*, vol. 1, no. 1, pp. 71–78, 1993.

[7] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the SPIN operating system," in *Proc. Symp. Operating Systems Principles*, pp. 267–284, 1995.

[8] T. Blackwell, "Speeding up protocols for small messages," in *Proc. SIGCOMM*, pp. 85–95, August 1996.

[9] J. Brignell and N. White, *Intelligent Sensor Systems*, Bristol, Philadelphia, 1994.

[10] G. Cena, L. Durante, and A. Valenzano, "Standard field bus networks for industrial applications," *Computer Standards and Interfaces*, vol. 17, no. 2, pp. 155–167, January 1995.

[11] R. S. Chin and S. T. Chanson, "Distributed object-based programming systems," *ACM Computing Surveys*, vol. 23, no. 1, pp. 91–124, March 1991.

[12] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," in *Proc. Int'l Symposium on Fault-Tolerant Computing*, pp. 200–206, June 1985.

[13] M. Crovella and A. Bestavros, "Self-similarity in world wide web traffic evidence and possible causes," in *Proc. SIGMETRICS*, pp. 160–169, May 1996.

[14] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of WWW client-based traces," Technical Report BU-CS-95-010, Boston University, Computer Science Department, 1995.

[15] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner," *IEEE Network*, vol. 7, no. 4, pp. 36–43, July 1993.

[16] P. B. Danzig, S. Jamin, R. Caceres, D. Mitzel, and D. Estrin, "An empirical workload model for driving wide-area TCP/IP network simulations," *Journal of Internetworking*, vol. 3, no. 1, pp. 1–26, March 1992.

[17] A. S. Debelack, J. D. Dehn, L. L. Muchinsky, and D. M. Smith, "Next generation air traffic control automation," *IBM Systems Journal*, vol. 34, no. 1, pp. 63–77, 1995.

[18] E. W. Dijkstra, "Cooperating sequential processes," Technical Report EWD-123, Technical University, Eindhoven, the Netherlands, 1965.

[19] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. Operating Systems Design and Implementation*, October 1996.

[20] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proc. Symp. Operating Systems Principles*, pp. 189–202, December 1993.

[21] P. Druschel, L. L. Peterson, and B. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *Proc. SIGCOMM*, pp. 2–13, August 1994.

[22] D. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Extensibility, safety and performance in the SPIN operating system," in *Proc. Symp. Operating Systems Principles*, pp. 251–266, December 1995.

[23] D. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *Proc. SIGCOMM*, pp. 53–59, August 1996.

[24] Wu-chang Feng, D. Kandlur, D. Saha, and K. Shin, "On providing minimum rate guarantees over the internet," Technical report, IBM Research report RC 20618, version 2, November 1997.

[25] Wu-chi Feng and F. Jahanian, "Providing VCR functionality in a constant quality video-on-demand transportation service," Technical Report CSE-TR 271-95, University of Michigan, EECS Dept., December 1995.

[26] D. Ferrari and D. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368–379, April 1990.

[27] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking*, vol. 3, no. 4, pp. 365–386, August 1995.

[28] *FIP bus for exchange of information between transmitters, actuators, and programmable controllers, NF C46 601-607*, French Association for Standardization, 1990.

[29] V. Frost and B. Melamed, "Traffic modeling for telecommunications networks," *IEEE Communications Mag.*, vol. 33, pp. 70–80, March 1994.

[30] J. G. Ganssle, *The Art of Programming Embedded Systems*, Academic Press, 1992.

[31] M. Gergeleit and H. Streich, "Implementing a distributed high-resolution real-time clock using the CAN-bus," in *1st International CAN Conference*, September 1994.

[32] *PROFIBUS standard part 1 and 2, DIN 19 245*, German Institute of Normalization, April 1991.

[33] R. Gopalakrishnan and G. Parulkar, "Bringing real-time theory and practice closer for multimedia computing," in *SIGMETRICS*, pp. 1–12, May 1996.

[34] P. Goyal, X. Guo, and H. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. Operating Systems Design and Implementation (OSDI)*, October 1996.

[35] R. K. Gupta, *Co-Sysnthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, 1995.

[36] A. N. Habermann, "Synchronization of communicating processes," *Communications of the ACM*, vol. 15, no. 3, pp. 171–176, March 1972.

[37] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, S. Mullender, editor, pp. 97–145, Addison Wesley, New York, second edition, 1993.

[38] H. Haertig, M. Hohmuth, J. Liedtke, S. Schoenberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Proc. Symp. Operating Systems Principles*, pp. 66–77, October 1997.

[39] D. Hildebrand, "An architectural overview of QNX," in *Proc. Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[40] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, October 1974.

[41] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[42] A. Indiresan, A. Mehra, , and K. G. Shin, "The END: An emulated network device for evaluating adapter design," in *Proc. 3rd Intl. Workshop on Performability Modeling of Computer and Communication Systems*, 1996.

[43] A. Indiresan, *Exploring Quality-of-Service Issues in Network Interface Design*, PhD thesis, University of Michigan, 1997.

[44] *Industrial Automation Systems — Systems Integration and Communication — Fieldbus (draft) (ISA/SP50–93)*, Instrument Society of America, 1st edition, 1993.

[45] *82527 Serial Communications Controller Architectural Overview*, Intel Corporation, 1993.

[46] *Electrical Equipment of Industrial Machines — Serial Data Link for Real-Time Communication between Controls and Drives*, International Electrotechnical Commission, 1994. Revision 8.

[47] *Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication. ISO 11898*, International Standards Organization, 1st edition, 1993.

[48] Y. Ishikawa, H. Tokuda, and C. W. Mercer, "An object-oriented real-time programming language," *IEEE Computer*, vol. 25, no. 10, pp. 66–73, October 1992.

[49] *Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication*, ISO 11898, 1993.

[50] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proc. Real-Time Systems Symposium*, pp. 129–139, 1991.

[51] E. D. Jensen and J. D. Northcutt, "Alpha: a non-proprietary operating system for large, complex, distributed real-time systems," in *Proc. IEEE Workshop on Experimental Distributed Systems*, pp. 35–41, 1990.

[52] M. B. Jones, D. Rosu, and M.-C. Rosu, "CPU reservations and time constraints: Efficient, predictable scheduling of independet activities," in *Proc. Symp. Operating Systems Principles*, pp. 198–211, October 1997.

[53] M. F. Kaashoek and et. al, "Application performance and flexibility on Exokernel systems," in *Proc. Symp. Operating Systems Principles*, pp. 52–65, October 1997.

[54] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[55] D. Kandlur, D. Saha, and M. Willebeek-LeMair, "Protocol architecture for multimedia applications over ATM networks," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1349–1359, September 1996.

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

[57] J. Kay and J. Pasquale, "Measurement, analysis, and improvement of UDP/IP throughput for the DECstation 5000," in *Proc. Winter USENIX*, pp. 249–258, January 1993.

[58] K. Kettler, D. Katcher, and J. Strosnider, "A modeling methodology for real-time/multimedia operating systems," in *Proc. Real-Time Technology and Applications Symposium*, pp. 15–26, 1996.

[59] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 460–467, June 1992.

[60] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: the MARS approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, February 1989.

[61] H. Kopetz and G. Grunsteidl, "TTP – a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[62] H. Kopetz and J. Reisinger, "The non-blocking write protocol NBW: a solution to a real-time synchronization problem," in *Proc. Real-Time Systems Symposium*, pp. 131–137, 1993.

[63] C. M. Krishna and K. G. Shin, *Real-Time Systems*, McGraw-Hill, 1997.

[64] G. Lawton, "Dawn of the Internet appliance," *IEEE Computer*, vol. 30, no. 10, pp. 16–18, October 1997.

[65] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable communication protocol processing in Real-Time Mach," in *Proc. Real-Time Technology and Applications Symposium*, pp. 220–229, June 1996.

[66] S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, Inc., 1989.

[67] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proc. Real-Time Systems Symposium*, 1989.

[68] G. W. Lenhart, "A field bus approach to local control networks," *Advances in Instrumentaion and Control*, vol. 48, no. 1, pp. 357–365, 1993.

[69] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, December 1982.

[70] H. Levy and R. Eckhouse, Jr., *Computer Programming and Architecture: The VAX-11*, Digital Press, 1980.

[71] T. Lewis, "Information appliances: Gadget netopia," *IEEE Computer*, vol. 31, no. 1, pp. 59–70, January 1998.

[72] J.-P. Li and M. W. Mutka, "Priority based real-time communication for large scale wormhole networks," in *Proc. International Parallel Processing Symposium*, pp. 433–438, April 1994.

[73] J. Liebeherr and D. E. Wrege, "Versatile packet multiplexer for quality-of-service networks," in *Proc. IEEE International Symposium on High Performance Distributed Computing*, pp. 148–155, 1995.

[74] J. Liebeherr, D. E. Wrege, and D. Ferrari, "Exact admission control for networks with a bounded delay service," *IEEE/ACM Trans. Networking*, vol. 4, no. 6, pp. 885–901, December 1996.

[75] J. Liedtke, "On μ-kernel construction," in *Proc. Symp. Operating Systems Principles*, pp. 237–250, December 1995.

[76] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[77] C. D. Locke, D. Vogel, L. Lucas, and J. Goodenough, "Generic avionics software specification," Technical Report CMU/SEI-90-TR-8, Carnegie Mellon University, 1990.

[78] C. D. Locke, D. Vogel, and T. Mesler, "Building a predictable avionics plarform in Ada: A case study," in *Proc. Real-Time Systems Symposium*, pp. 181–189, 1991.

[79] C. Maeda and B. Bershad, "Protocol service decomposition for high-performance networking," in *Proc. Symp. Operating Systems Principles*, pp. 244–255, December 1993.

[80] *Manufacturing Automation Protocol (MAP) 3.0 Implementation Release*, MAP/TOP Users Group, 1987.

[81] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, February 1991.

[82] C. Mercer and H. Tokuda, "An evaluation of priority consistency in protocol architectures," in *Proc. IEEE Conf. Local Computer Networks*, pp. 386–398, October 1991.

[83] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[84] J. Mogul, R. Rashid, and M. Accetta, "The packet filter: An efficient mechanism for user-level network code," in *Proc. Symp. Operating Systems Principles*, pp. 39–51, November 1987.

[85] A. K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," *Ph.D thesis*, 1983.

[86] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Analysis of techniques to improve protocol processing latency," in *Proc. SIGCOMM*, pp. 73–84, August 1996.

[87] *M68040 User's Manual*, Motorola Inc., 1992.

[88] *MC68336/376 User's Manual*, Motorola Inc., 1996.

[89] *OSEK/VDX Operating System Specification 2.0*, OSEK Group, 1997. Revision 1.

[90] V. Paxson and S. Floyd, "Wide area traffic: The failure of poisson modeling," *IEEE/ACM Trans. Networking*, vol. 3, no. 3, pp. 226–244, June 1995.

[91] P. Pleinevaux and J. D. Decotignie, "Time critical communication networks: field buses," *IEEE Network*, vol. 2, no. 3, , May 1988.

[92] S. Poledna, T. Mocken, and J. Schiemann, "ERCOS: an operating system for automotive applications," in *SAE International Congress and Exposition*, pp. 55–65, February 1996. SAE Technical Paper Series 960623.

[93] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang, "Optimistic incremental specialization: Streamlining a commercial operating system," in *Proc. Symp. Operating Systems Principles*, pp. 314–324, December 1995.

[94] R. S. Raji, "Smart networks for control," *IEEE Spectrum*, vol. 31, no. 6, pp. 49–55, June 1994.

[95] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, January 1994.

[96] A. Reibman and A. Berger, "Traffic descriptions for VBR video teleconferencing over ATM networks," *IEEE/ACM Trans. Networking*, vol. 3, no. 3, pp. 329–339, June 1995.

[97] D. C. Schmidt and T. Suda, "Transport system architecture services for high-performance communications systems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 489–506, May 1993.

[98] H. Schulzrinne, "RTP profile for audio and video conferences with minimal control," RFC 1890, January 1996.

[99] J. Scourias, "Overview of the Global System for Mobile Communications," http://ccnga.uwaterloo.ca/~jscouria/GSM/gsmreport.html.

[100] M. Seltzer, Y. Endo, C. Small, and K. Smith, "Dealing with disaster: Surviving misbehaving kernel extensions," in *Proc. Operating System Design and Implementation*, October 1996.

[101] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. on Computers*, vol. 39, no. 3, pp. 1175–1198, 1990.

[102] K. G. Shin, "Real-time communications in a computer-controlled workcell," *IEEE Trans. Robotics and Automation*, vol. 7, no. 1, pp. 105–113, February 1991.

[103] 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, January 1994.

[104] *SAE Handbook*, Society of Automotive Engineers, 1995. pp. 23.560–23.573.

[105] J. D. Solomon, *Mobile IP, the Internet Unplugged*, Prentice Hall, 1998.

[106] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, June 1989.

[107] J. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a bidding algorithm for real-time distributed systems," *IEEE Trans. on Computers*, vol. C.34, no. 12, , December 1985.

[108] J. Stankovic, "Misconceptions about real-time computing," *IEEE Computer*, vol. 21, no. 10, pp. 10–19, October 1988.

[109] J. Stankovic and K. Ramamritham, "The Spring Kernel: a new paradigm for real-time operating systems," *ACM Operating Systems Review*, vol. 23, no. 3, pp. 54–71, July 1989.

[110] W. R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.

[111] H. Takada and K. Sakamura, "Experimental implementations of priority inheritance semaphore on ITRON-specification kernel," in *11th TRON Project International Symposium*, pp. 106–113, 1994.

[112] K. Tindell, A. Burns, and A. J. Wellings, "Calculating Controller Area Network (CAN) message response times," *Control Engineering Practice*, vol. 3, no. 8, pp. 1163–1169, 1995.

[113] K. W. Tindell, H. Hansson, and A. J. Wellings, "Analyzing real-time communications: Controller Area Network (CAN)," in *Proc. Real-Time Systems Symposium*, pp. 259–263, December 1994.

[114] H. Tokuda and T. Nakajima, "Evaluation of real-time synchronization in Real-Time Mach," in *Second Mach Symposium*, pp. 213–221. Usenix, 1991.

[115] Robbert van Renesse, "Masking the overhead of protocol layering," in *Proc. SIGCOMM*, pp. 96–104, August 1996.

[116] C.-D. Wang, H. Takada, and K. Sakamura, "Priority inheritance spin locks for multiprocessor real-time systems," in *2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 70–76, 1996.

[117] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Engineering*, vol. 19, no. 2, pp. 139–154, 1993.

[118] H. Zeltwanger, "An inside look at the fundamentals of CAN," *Control Engineering*, vol. 42, no. 1, pp. 81–87, January 1995.

[119] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new Resource ReSerVation Protocol," *IEEE Network*, vol. 7, no. 9, pp. 8–18, September 1993.

[120] W. Zhao and K. Ramamritham, "Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints," *Jounal of Systems and Software*, vol. 7, pp. 195–205, 1987.

[121] Q. Zheng and K. G. Shin, "On the ability of establishing real-time channels in point-to-point packet-switched networks," *IEEE Trans. Communications*, vol. 24, no. 2/3/4, pp. 1096–1105, February/March/April 1994.

[122] K. M. Zuberi and K. G. Shin, "Non-preemptive scheduling of messages on Controller Area Network for real-time control applications," in *Proc. Real-Time Technology and Applications Symposium*, pp. 240–249, May 1995.

[123] K. M. Zuberi and K. G. Shin, "A causal message ordering scheme for distributed embedded real-time systems," in *Proc. Symposium on Reliable and Distributed Systems*, pp. 210–219, October 1996.

[124] K. M. Zuberi and K. G. Shin, "EMERALDS: A microkernel for embedded real-time systems," in *Proc. Real-Time Technology and Applications Symposium*, pp. 241–249, June 1996.

[125] K. M. Zuberi and K. G. Shin, "Real-time decentralized control with CAN," in *Proc. IEEE Conference on Emerging Technologies and Factory Automation*, pp. 93–99, November 1996.

[126] K. M. Zuberi and K. G. Shin, "An efficient semaphore implementation scheme for small-memory embedded systems," in *Proc. Real-Time Technology and Applications Symposium*, pp. 25–34, June 1997.

[127] K. M. Zuberi and K. G. Shin, "Scheduling messages on Controller Area Network for real-time CIM applications," *IEEE Trans. Robotics and Automation*, pp. 310–314, April 1997.

[128] K. M. Zuberi and K. G. Shin, "An efficient end-host protocol processing architecture for real-time audio and video traffic," to appear in *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, July 1998.