EMERALDS: A Small-Memory Real-Time Microkernel *

Khawar M. Zuberi†
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052

khawarz@microsoft.com

Kang G. Shin
Real-Time Computing Lab
Department of EECS
The University of Michigan
Ann Arbor, MI 48109-2122

kgshin@eecs.umich.edu

Abstract

EMERALDS (Extensible Microkernel for Embedded, ReAL-time, Distributed Systems) is a real-time microkernel designed for small-memory embedded applications. These applications must run on slow (15–25MHz) processors with just 32–128 kbytes of memory, either to keep production costs down in mass-produced systems or to keep weight and power consumption low. To be feasible for such applications, the OS must not only be small in size (less than 20 kbytes) but also have low-overhead kernel services. Unlike commercial embedded OSs which rely on carefully-optimized code to achieve efficiency, EMERALDS takes the approach of re-designing the basic OS services of task scheduling, synchronization, communication, and system call mechanism by using characteristics found in small-memory embedded systems such as small code size and *a priori* knowledge of task execution & communication patterns. With these new schemes, the overheads of various OS services are reduced 20–40% without compromising any OS functionality.

1 Introduction

Real-time computing systems must behave predictably even in unpredictable environments [1]. This predictability is ensured by system-level services, most important among them being the real-time operating system (RTOS). The RTOS must ensure that all real-time tasks complete by their deadlines and that no low-priority execution or communication activities are able to block higher-priority tasks for an extended period. The wide variety of real-time applications (from multimedia to industrial automation control) and the variety of hardware used in these systems (from single-board computers to distributed systems to multiprocessors) have resulted in dozens of RTOSs being designed to support these applications. Commercial RTOSs like pSOS [2], QNX [3],

*The work reported in this paper was supported in part by the NSF under Grant MIP-9203895, and by the ONR under Grants N00014-94-1-0229 and N00014-99-1-0465. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the funding agencies. An earlier version was presented at the 17th ACM Symposium on Operating Systems Principles (SOSP99).

†This work was done while the author was at the University of Michigan.

and VxWorks [4] support both stand-alone as well as distributed systems. Research RTOSs like HARTOS [5] and the Spring Kernel [6] were designed for multiprocessors, while other research OSs like Harmony [7] and RT-Mach [8] are for distributed platforms. All these RTOSs were designed with relatively powerful processors and networks in mind: processors with several megabytes of memory and networks with at least tens of Mbit/s bandwidth. In fact, just the code size of some of these RTOSs alone is in the megabytes. This is acceptable for many real-time applications such as robotics, telemetry, and multimedia. However, real-time computing today is no longer limited to high-powered, expensive applications. Many embedded real-time control applications use slow processors with small memories (tens of kilobytes) and slow fieldbus networks (with 1–2 Mbit/s bandwidth) [9, 10]. There are two main reasons for using such restricted hardware:

- To keep production costs down in mass-produced items such as home & portable electronics and automotive control systems.
- To keep weight and power consumption low in avionics and space applications.

For the first category, systems like automotive engine & ABS controllers, cellular phones, and camcorders are all produced in volumes of millions of units. Keeping per unit costs down is vital in such systems and savings of even one dollar per unit translate into millions of dollars of overall savings. For the second category, even though cost may not be the primary limiting factor, the weight and power consumption restrictions simply do not allow elaborate hardware. As a result, the only hardware feasible for such applications is cheap/slow processors with limited amounts of memory. Yet, these small-memory embedded systems must perform increasingly complex tasks. Automotive engine controllers use sophisticated control algorithms with an increasing number of sensors to precisely control the engine to minimize exhaust emissions. Camcorders can counteract a jittery human operator to stabilize the picture. Cellular phones use filtering algorithms to produce a clear sound. All these algorithms must run on slow processors with only a few tens of kilobytes of on-chip ROM and static RAM (external memory increases cost, volume, weight, and power consumption, so it is not used in our target applications).

Because of these severe hardware restrictions, the above-mentioned applications till recently did not use an OS at all. Instead, the application directly managed all hardware resources. But, as these applications became more sophisticated (due to additional functions introduced), continuing with this approach became infeasible; the application code was too difficult to modify and almost impossible to port. Moreover, the product development cycle was long and expensive because the application designer had to include OS functionality in application code. The need for an OS in these small-memory embedded applications opened up a new market for small-size, low-overhead RTOSs. These RTOSs must not only provide predictable services but must also be *efficient* and *small* in size. The maximum acceptable kernel code size is about 20 kbytes and the kernel services such as task scheduling, system calls, and interrupt handling, must incur minimal overheads. Many commercial vendors responded to this demand with products like RTXC [11], pSOS Select, and a dozen other small, real-time kernels. Their approach was to take a core set of OS services (task scheduling, semaphores, timers, interrupt handling, etc.), implement them using optimized, carefully-crafted code, and package them into an OS.

EMERALDS is an RTOS designed for small-memory embedded systems. Like the above-mentioned commercial RTOSs, EMERALDS also provides a core set of OS services in a small-sized kernel, but our approach for achieving efficiency in EMERALDS is to rely not on carefully-crafted code but on new OS schemes and algorithms. We focus on the following key OS services:

- Task scheduling
- Semaphores
- Intra-node message-passing[1]
- Memory protection and system call overhead.

For each of these areas, we have designed innovative schemes which lower OS overheads without compromising OS functionality, thus making more computational resources available for the execution of application tasks. To achieve this, we used some basic characteristics common to all small-memory embedded systems such as small kernel and application code size and *a priori* knowledge of task communication and execution patterns. Some of these characteristics are also found in other real-time applications, so some of the schemes we present (such as the task scheduler) have applicability beyond small-memory embedded systems.

In the next section, we describe the performance requirements that an RTOS must satisfy to be feasible for embedded applications. Section 3 presents a brief overview of EMERALDS. Section 4 shows how EMERALDS, as a real-time embedded OS, is different from more generalized micro-kernels like Mach [14] and SPIN [15]. Sections 5–8 give details of our scheduling, synchronization, message-passing, and system call schemes, respectively. Performance of these schemes is evaluated in Section 9 and we conclude with Section 10.

2 Embedded Application Requirements

Our target embedded applications use single-chip microcontrollers with relatively slow processing cores (such as Motorola 68000 derivatives) running at 15–25 MHz. Typical examples are the Motorola 68332 and Intel i960 controllers. All ROM and RAM is on-chip which limits memory size

[1] Inter-node networking issues are discussed in [12, 13] and are not covered in this paper.

to 32–128 kbytes. These applications are either uniprocessor (such as cellular phones and home electronics) or distributed, consisting of 5–10 nodes interconnected by a low-speed (1–2 Mbit/s) fieldbus network (such as automotive and avionics control systems).

Despite these hardware restrictions, the RTOS must still provide a comprehensive set of services:

1. Task scheduling
2. Task synchronization (semaphores)
3. Task communication (message-passing)
4. Memory protection
5. Interaction with external environment (interrupt handling)
6. Clock and timer services.

Note that disks are not used in small-memory embedded systems, so a file system is not part of the RTOS.

Of these basic services, the last two deal with hardware devices such as the on-chip timer and the processor's interrupt handling mechanism, so their overhead is dictated primarily by the hardware. Other than optimizing the kernel code, the OS designer can do little to reduce the overhead of these two services. However, the remaining services present opportunities for innovative optimizations. The thrust of EMERALDS is to come up with new optimized solutions for embedded systems for the well-known problems of scheduling, synchronization, communication, and page-table-based memory protection by using certain characteristics common to all embedded applications. The remainder of this section discusses why improvements in these four areas are important for embedded systems and what hurdles must be overcome to reduce the overhead of these OS services.

2.1 Task Scheduling

Consider a periodic task which runs once every 1 ms. For just this one task, the task scheduler must run twice every 1 ms: once when the task is released and once when the task completes. Considering that typical OS operations take 50–100 μs on the slow processors we are concerned with and that a typical task workload consists of 10–20 tasks with at least 3–5 tasks having periods less than 10 ms, the scheduler's execution alone can use up 10–15% of CPU time. This is why embedded application programmers have till now preferred cyclic time-slice scheduling techniques in which the entire schedule is calculated off-line, 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 [16], 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 embedded 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 *rate-monotonic* (RM) [17] and *earliest-deadline-first* (EDF) [17] 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. However, since they make run-time scheduling decisions, they incur overhead which can be 10–15% of the CPU time. This calls for new task scheduling schemes with lower overheads which would free up more time for application tasks as described in Section 5.

2.2 Task Synchronization

Object-oriented (OO) programming is ideal for designing real-time software. Real-time systems must deal with real-world entities, and objects are ideal for modeling these entities: the object's internal data represents the physical state of the entity (such as temperature, pressure, position, RPM, etc.) and the object's 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 simply a collection of threads of execution, each invoking the methods of various objects [18].

Conceptually, this OO paradigm is very appealing and gives benefits such as reduced software design time and software re-use. But these benefits come at a cost. The methods of an object must synchronize their access to the object's data to ensure mutual exclusion. Semaphores [19, 20] are typically used for this purpose (e.g., to provide the monitor construct [21]). Because a semaphore system call is made every time an object's method is invoked, semaphore operations of `acquire` and `release` become some of the most heavily-used OS primitives when OO design is used. This calls for new and efficient schemes for implementing semaphore locking in EMERALDS as described in Section 6.

2.3 Task Communication

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 receiver task. While this scheme is suitable for certain purposes, it has two major disadvantages:

- Passing one message may take 50–100 μs on a processor such as the Motorola 68040. Since tasks in embedded applications usually need to exchange several thousand messages per second, this overhead is unacceptable.
- If a task needs to send 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 requires new mechanisms for intertask communication. We selected the *state message* paradigm [22] which makes protected global variables available for information exchange between tasks. We optimized the basic state message scheme to reduce execution overhead and memory consumption as described in Section 7.

2.4 Memory Protection

Providing memory protection requires maintaining page tables and programming the memory management unit. This not only increases the size of the kernel, but also adds overhead to several kernel services, thus being contrary to our primary goal of building a small and fast kernel.

The need for memory protection in time-shared systems is indisputable. One user's processes must be protected from all other — possibly malicious — processes. But in embedded systems, all processes are cooperative and will never try to intentionally harm another process, making memory protection seem extraneous. However, bugs in application code can manifest themselves as malicious faults. For example, suppose some pointer in a C program is left uninitialized. If this pointer is used for writing, one process can easily corrupt another process or even the kernel.

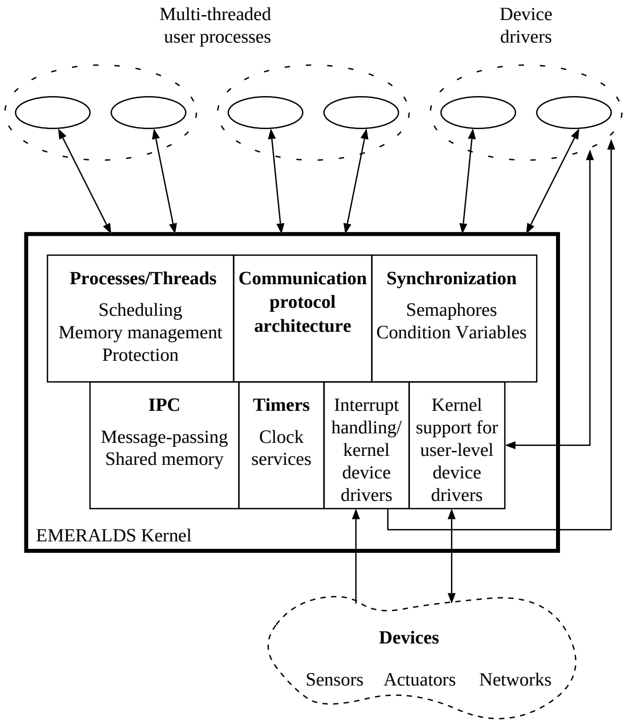

Figure 1: EMERALDS' architecture.

With memory protection, such an access will cause a TRAP to the kernel and recovery action may be taken, providing a form of software fault-tolerance. Without memory protection, such a fault may not even be detected until the CPU crashes with possibly catastrophic consequences. Also, in EMERALDS, the kernel is mapped into each user-level address space. This way, a system call reduces to a TRAP, then a jump to the appropriate kernel address, without the need to switch address spaces. Details are given in Section 8.

3 Overview of EMERALDS

EMERALDS is a microkernel real-time operating system written in the C++ language. Following are EMERALDS' salient features as shown in Figure 1.

- Multi-threaded processes:
  - Full memory protection between processes.
  - Threads are scheduled by the kernel.
- IPC based on message-passing and mailboxes. Shared-memory support is also provided.
  - Optimized local message passing.

- Semaphores and condition variables for synchronization; priority inheritance for semaphores.
- Support for communication protocol stacks [23].
- Highly optimized context switching and interrupt handling.
- Support for user-level device drivers.

To provide all these services in a small-sized (less than 20 kbytes) kernel, we make use of certain characteristics of embedded applications. First of all, our target applications are in-memory, so a file system is not needed. Moreover, embedded application designers know which resources (threads, mailboxes, etc.) reside at which node, so naming services are not necessary, allowing considerable savings in code size. Also, nodes in embedded applications typically exchange short, simple messages over fieldbuses. Threads can do so by talking directly to network device drivers, so EMERALDS does not have a built-in protocol stack. Further details regarding protocol stacks, device drivers, EMERALDS system calls, and other techniques used to reduce code size in EMERALDS can be found in [24]. With these techniques, EMERALDS provides a rich set of OS services in just 13 kbytes of code.

An embedded RTOS must not only be small but efficient as well. In the remainder of this paper, we focus on new kernel schemes for reducing the overheads of scheduling, synchronization, message-passing, and system calls.

4 What Makes EMERALDS Different ?

Microkernel optimization has been an active area of research in recent years, but little effort has been made in addressing the needs of real-time systems, let alone small-memory embedded ones. In microkernels designed for general-purpose computing such as Mach [14], L3 [25], and SPIN [15], researchers have focused on optimizing kernel services such as thread management [26, 27], IPC [28], and virtual memory management [29]. Virtual memory is not a concern in our target applications. Thread management and IPC *are* important but not for the same reasons as for general-purpose computing. The sources of OS overhead are different for embedded real-time systems than for general-purpose computing systems and this necessitates different optimization techniques.

Thread management is a concern in typical microkernels because either the kernel itself has a large number of threads (one for each user-level thread) so that switching overhead and stack use by these threads must be minimized [26], or, in case of user-level threads, the kernel must export the correct interface to these threads [27]. Neither of these concerns apply to EMERALDS. EMERALDS has kernel-managed threads but the kernel itself has no threads. To make a system call, a user thread enters protected kernel mode and simply calls the appropriate kernel procedure (see Section 8). What *is* important in a real-time environment is that these threads be scheduled properly to ensure timely completion of real-time tasks. So, in EMERALDS, optimizing thread management takes the form of ensuring low-overhead transition between user and kernel modes and providing efficient real-time scheduling of threads.

IPC is important in most microkernels because RPC is used to communicate with user-level servers. Frequently-accessed services such as file systems and virtual memory are implemented in servers. But embedded systems do not need these services. In EMERALDS, only inter-node networking is implemented at the user level and even this server is accessed only infrequently (because nodes are loosely-coupled). Instead, IPC is important in embedded systems for intra-node inter-task communication and this is what we address in EMERALDS.

Task synchronization does not receive much attention in the design of most microkernels, but it is of crucial importance in embedded systems. The little research that has been conducted in this area has focused primarily on multiprocessors [30, 31], whereas we are interested in uniprocessor locking.

In summary, design of an optimized OS for small-memory real-time embedded applications is a largely under-explored area of research. With embedded systems quickly becoming part of everyday life, designing OSs targeted specifically toward embedded applications has become important (as witnessed by the emergence of many commercial RTOSs, see Section 9.5), and EMERALDS is a first step in this direction.

5 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, an execution time c_i, and deadline d_i. In this paper, we assume $d_i = P_i$ unless stated otherwise (see [32] for methods to derive schedulability conditions with this restriction relaxed). Note that inside the kernel, tasks are represented by threads. Then, this workload has a utilization $U = \sum_{i=1}^{n} c_i/P_i$. Obviously, no scheduler can schedule a workload with

$U > 1$. EDF is a dynamic priority scheduler which gives the highest priority to the earliest-deadline task [17], and can schedule all workloads with $U \leq 1$ under the ideal condition that EDF's run-time overhead is ignored. $U^* = 1$ for EDF, because this is the utilization that EDF can schedule under ideal conditions. Other schedulers such as the static priority RM scheduler (which schedules tasks according to fixed priorities based on the tightness of their P_i [17]) can have $U^* < 1$. For example, a workload with $U = 0.80$ may be schedulable under RM, but if some c_i is slightly increased so that U becomes 0.81, the workload may no longer be schedulable even under ideal conditions. $U^* = 0.80$ for this workload under RM. This means that 20% of CPU time is wasted because of the scheduling policy, and we refer to this as the *schedulability overhead*.

EDF has zero schedulability overhead but high run-time overhead. 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 devise a scheduler with low schedulability and run-time overheads which gives better performance than both EDF and RM.

Note that both static and dynamic priority schedulers have advantages/disadvantages other than those mentioned above. For example, dynamic schedulers can in general handle aperiodic tasks better than static schedulers [12]. On the other hand, static schedulers may provide better guarantees for completion of critical tasks under processor overload situations. Detailed discussion of these issues is beyond the scope of this paper. Interested readers are referred to [32, 33] for comparisons between various scheduling methodologies. In this paper, we focus on schedulability and run-time overhead properties of EDF and RM schedulers.

5.1 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 a half of the tasks block once during their execution, waiting for a message or a signal to be sent by one of the other half of the tasks. For simplicity, we can say 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. 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 (especially when using priority inheritance semaphores which cause repeated changes in thread priorities as described in Section 6), 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 semaphores (as discussed in Section 6) 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` is 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. 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 for 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).

5.2 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, $U^* =$

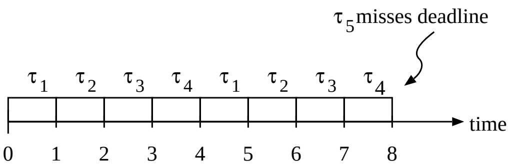

Figure 2: RM scheduling of the workload in Table 1.

0.88 for RM [34]. To see why U^* for RM is less than that for EDF, consider the workload shown in Table 1. Each task τ_i has deadline $d_i = P_i$. $U = 0.88$ for this workload, so it is feasible under EDF.

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 1: A typical task workload with $U = 0.88$. It is feasible under EDF but not under RM.

Figure 2 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 τ_1–τ_4 run for the second time (because $d_5 = 8$ is earlier than the deadlines of second invocations of τ_1–τ_4) and the workload will be feasible.

5.3 CSD: a Balance between EDF and RM

Going back to the workload in Table 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 much 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 combined static/dynamic (CSD) scheduler. Under CSD, tasks τ_1–τ_5 will be scheduled by EDF so that τ_5 will not miss its deadline. 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 the 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* [35], 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`).

5.4 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. <u>FP task blocks:</u> 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 $O(1) + O(r) + O(1) + O(r) = 2O(r)$, 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 significantly less than that of EDF (considering that the median r is about $n/2$; see Section 9) and only slightly greater than that of RM. Considering that CSD has no schedulability overhead, it easily outperforms both EDF and RM. This is corroborated by performance measurements in Section 9.

5.5 Schedulability Test

A task set $\{\tau_i : i = 1, 2, \ldots, n\}$ with tasks sorted by their priority (tasks with shorter periods have lower index i) is feasible under EDF if [17]

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

where $\Delta t(EDF)$ is Δt for EDF. The workload is feasible under RM if [34]

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

In practice, this equation need only be evaluated for a finite number of t values as described in [32].

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

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

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

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

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).

The best possible length of the DP queue for a given workload is found iteratively. 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.

5.6 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.

5.6.1 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.

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 5.6.3, 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.

5.6.2 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 5.4. The overheads for different cases are shown in Table 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.

		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 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.

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 $O(n-q)$.

5.6.3 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 1. Suppose the least run-time overhead results by putting tasks τ_1–τ_4 in

[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.

DP1 and the rest of the DP tasks in DP2, but this will cause τ_5 to miss its deadline (see Figure 2). 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.

5.7 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:

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

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 [17] 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.

5.8 Beyond CSD-3

The general scheduling framework of CSD is not limited to just three queues. It can be extended to have $4, 5, \ldots, n$ 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 9.1), 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.

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 paper. We demonstrated the usefulness of the general CSD scheduling framework and how it can be beneficial in real systems.

6 Efficient Semaphore Implementation

Previous work in lowering the overhead of semaphore operations has focused on either relaxing the semaphore semantics to get better performance [36] or coming up with new semantics and new synchronization policies [37]. The problem with this approach is that such new/modified semantics may be suitable for some particular applications, but usually do not have wide applicability.

We took an approach of providing full semaphore semantics (with priority inheritance [38]), but optimizing the implementation of these semaphores by exploiting certain features of embedded applications.

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 by using characteristics common in small-memory embedded applications.

6.1 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 */
}
```

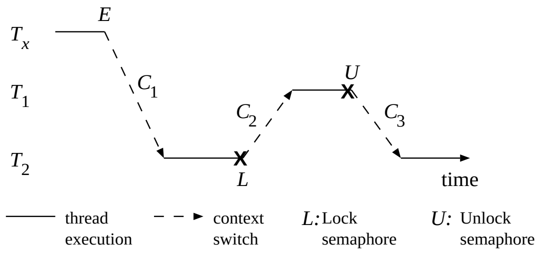

Figure 3: A typical scenario showing thread T_2 attempting to lock a semaphore already held by thread T_1. T_x is an unrelated thread which was executing while T_2 was blocked.

```
lock sem;
```

Priority inheritance [38] is needed in real-time systems to avoid unbounded priority inversion [37]. If a high-priority thread T_h calls `acquire_sem()` 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 `acquire_sem()` is called, then the semaphore lock operation has very little overhead. In fact, for this case, only one counter 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 `acquire_sem()` call. Figure 3 shows a typical scenario for this situation. Thread T_2 wakes up (after completing some unrelated blocking system call) and then calls `acquire_sem()`. 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 4.

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), 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, albeit 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

```
    unblock T2
    context switch C1 (Tx to T2)
(T2 executes and calls acquire_sem())
    do priority inheritance (T2 to T1)
    block T2
    context switch C2 (T2 to T1)
(T1 executes and calls release_sem())
    undo priority inheritance of T1
    unblock T2
    context switch C3 (T1 to T2)
```

Figure 4: Operations involved in locking a semaphore for the scenario shown in Figure 3

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.

6.2 Implementation in EMERALDS

Going back to Figure 4, we want to eliminate context switch C_2 [39]. We also want to optimize the two PI steps. First, we deal with C_2 which occurs when T_2 is 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 `acquire_sem()`, 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). This is implemented as follows. As part of the blocking call just preceding `acquire_sem()`, 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 `release_sem()` call by T_1. Comparing Figure 5 to Figure 3, 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 9 for performance results).

For FP tasks, we want to optimize the two PI steps, each of which takes $O(n-r)$ time. The

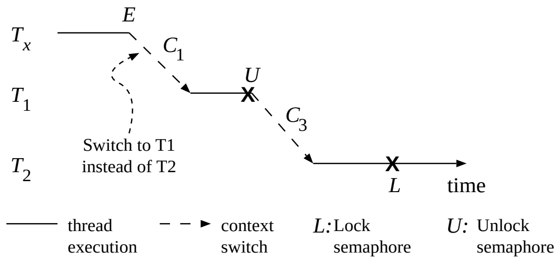

Figure 5: The new semaphore implementation scheme. Context switch C_2 is eliminated.

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 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.

6.2.1 Code Parser

In EMERALDS, all blocking calls take an extra parameter which is the identifier of the semaphore to be locked by the upcoming `acquire_sem()` call. This parameter is set to -1 if the next blocking call is not `acquire_sem()`.

Semaphore identifiers are statically defined (at compile time) in EMERALDS as is commonly the case in OSs for small-memory applications, so it is trivial to write a parser which examines the application code and inserts the correct semaphore identifier into the argument list of blocking calls just preceding `acquire_sem()` calls. Hence, the application programmer does not have to make any manual modifications to the code.

6.2.2 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):

1. What if thread T_2 does not block on the call preceding `acquire_sem()`? 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 `acquire_sem()`, then a context switch has already been saved. For such a situation, T_2 will continue to execute till it reaches `acquire_sem()` and a context switch will occur there. What our scheme really provides is that a context switch will be saved either on the `acquire_sem()` call or on the preceding blocking call. Where the savings actually occur at run-time do not really matter to the 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.

6.3 Applicability of the New Scheme

Going back to Figure 5, 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

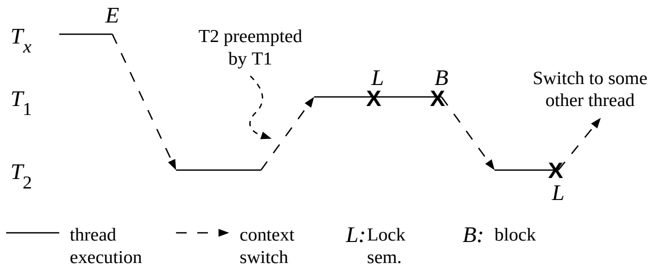

Figure 6: If a higher priority thread T_1 preempts T_2, locks the semaphore, and blocks, then T_2 incurs the full overhead of `acquire_sem()` and a context switch is not saved.

`acquire_sem()`), 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, T_1 had priority lower 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 6). But before T_2 can call `acquire_sem()`, T_1 wakes up, preempts T_2, locks S, then blocks for some event. T_2 resumes, calls `acquire_sem()`, 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. These problems can be resolved as follows. First, by making a small modification to our semaphore scheme, we can 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. By looking at the larger picture and considering threads other than just T_1 and T_2, we can then show that this problem is easily circumvented and our semaphore scheme works for all blocking situations that occur in practice, as discussed next.

6.3.1 Modification to the Semaphore Scheme

The problem illustrated in Figure 6 necessitates a small modification to our scheme. 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 6), the OS first checks if S is available or not, before unblocking T_2. Now, let's 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 `acquire_sem()` but have not yet called `acquire_sem()`.

Thread T_1 will also get added to this queue as part of its blocking call just preceding `acquire_sem()`. When T_1 calls `acquire_sem()`, the OS first removes T_1 from this queue, then puts all threads re-

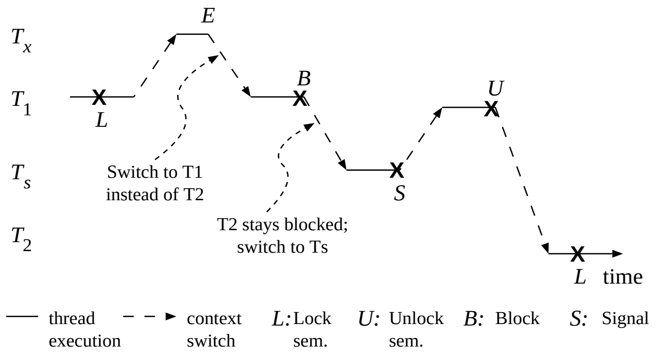

Figure 7: Situation when the lock holder T_1 blocks for a signal from another thread T_s.

maining in the queue in a blocked state. Then, when T_1 calls `release_sem()`, the OS unblocks all threads in the queue.

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

6.3.2 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.

Blocking for Internal Events, i.e., Handshake between Threads: 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 7). But it is perfectly safe to delay T_2 like this (even if T_s is lower in priority than T_2) because T_2 cannot lock S until T_1 releases it, and T_1 will not release it until it receives the signal from T_s. Letting T_s execute earlier 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. 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 for a long time, preventing other threads from making progress. This is why 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.

7 State Messages for Intertask Communication

From the performance point of view, global variables are ideal for sharing information between tasks, but if reading from and writing to global variables are not regulated, subtle bugs can crop up in the application code. State messages [22] 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.

7.1 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.

7.2 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 τ_1 reads a sensor and supplies the reading to task τ_2. If τ_1 sends two such messages to τ_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 τ_2 must first read the old sensor reading before it can get the new one. Moreover, if multiple tasks need the same sensor reading, τ_1 must send a separate message to each.

State messages streamline this entire process. An SMmailbox $SM1$ will be associated with τ_1 and it will be known to all tasks that $SM1$ contains the reading of a certain sensor. Every time τ_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 τ_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 τ_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.

7.3 Previous Work

State messages were first used in the MARS OS [22] and have also been implemented in ERCOS [40]. The state message implementation used in these systems as described in [41] 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 [41] 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 what follows, we describe our implementation for state messages including the calculation of N for the case when both readers and writers are residing on the same CPU. Then, we describe a system call included in EMERALDS to support state messages.

7.4 Implementation of State Messages 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. The tool **MessageGen** 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. In this way readers always get the most recent consistent copy of the message.

Calculating Buffer Depth N: Now, we address the issue of how to set N, the depth of the buffer. It is possible 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:

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

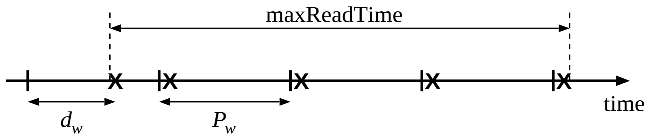

Figure 8: Calculation of x_{max}. Write operations are denoted by **X**. Excluding the first write, there are $\lfloor (maxReadTime - (P_w - d_w))/P_w \rfloor = 4$ writes, so $x_{max} = 5$.

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 8 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.

8 Memory Protection and System Calls

The benefits of memory protection mentioned earlier in Section 2.4 will not be of much practical use if the implementation of memory protection was not efficient and small-sized. In fact, many small

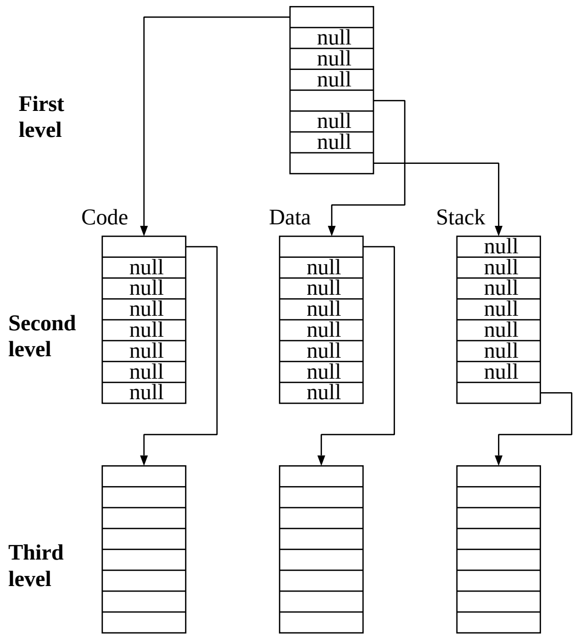

Figure 9: A typical page table in EMERALDS.

RTOSs (specially commercial ones) do not include memory protection because of the belief that it's negative impact on performance would be unacceptable. We show that this belief is unfounded.

To get an efficient and small-sized implementation of memory protection, 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 without relying on any special hardware features. In virtual memory systems with disk backing stores, there is a need to distinguish unmapped pages from those which have been swapped out to disk. 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. 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 9.

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

Other OSs with virtual memory with disk backing stores also have small page tables, but they achieve this with hardware support. Linux uses segment registers present in x86 processors to

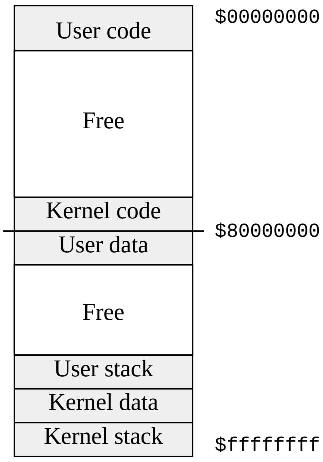

Figure 10: A typical address space in EMERALDS. Area labeled *kernel stack* is used for interrupts and area labeled *user stack* is used by both the user and the kernel.

distinguish unmapped and swapped out pages. VAX/VMS uses the page table length register of the VAX-11 to achieve the same goal [42]. But depending on such hardware support in embedded systems is not feasible since it increases hardware costs.

The small size of page tables not only saves memory, but also enables other optimizations like mapping the kernel into every address space. A typical 32-bit EMERALDS address space is shown in Figure 10. 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), 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 9).

Note that 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 directly mapping it into each address space is not feasible. Some OSs like Windows NT [43] use a lazy mapping scheme in which portions of the kernel's data segment are dynamically mapped into user address spaces as needed, but this incurs the high (and unpredictable) overhead of manipulating page tables when a system call needs to access some data and discovers that the page is not mapped into the current address space. Such schemes are not appropriate for embedded systems because of their high overhead and unpredictability. In EMERALDS, mapping the kernel in every address space 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. Moreover, once an address space is set up, no more page table mnipulation is needed, leading to low overhead. Also, the kernel areas are protected from corruption by buggy 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.

9 Performance Evaluation

EMERALDS has been implemented on the Motorola 68040 processor [44]. It has a code size of just 13 kbytes. EMERALDS has also been ported to the PowerPC 505, the Super Hitachi 2 (SH-2), and the Motorola 68332 microcontroller [45], the last two of which are popular in automotive control applications. In this section, we evaluate the benefits of the scheduling, semaphore, message-passing, and system call schemes presented thus far. These evaluations are performed on a 25MHz MC 68040 with 4 kbyte I- and D-caches. All measurements are made using a 5MHz on-chip timer.

EMERALDS is also being evaluated by the Scientific Research Laboratory, Ford Motor Company for use in automotive engine control. Their initial testing focused on basic OS overheads related to interrupt handling, context switching, event signaling, and timer services. Their evaluation covered nine commercial RTOSs in addition to EMERALDS, and their evaluation results are presented at the end of this section.

9.1 CSD Scheduler

In this subsection, 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 [44] with separate 4kbytes instruction and data caches are in Table 3. The run-time overhead of CSD is derived from these values as already discussed in Sections 5.4 and 5.6.2. 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 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

	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 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.

utilization at which the workload becomes infeasible is called the *breakdown utilization* [46]. We expect that with scheduling overheads considered, CSD will have the highest breakdown utilization.

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 5.5 and 5.7, 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 11–13 show breakdown utilizations when task periods are divided by 1, 2, and 3, respectively. In Figure 11, 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 12 is for periods in the 2.5–500ms 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 13 shows similar results. Task periods range from 1.67–333ms, and these short periods allow RM to quickly overtake EDF. Nevertheless, CSD continues to be superior to both.

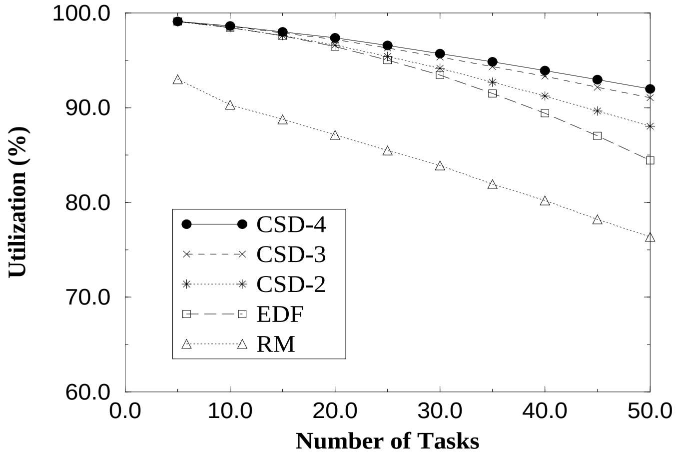

Figure 11: Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 1.

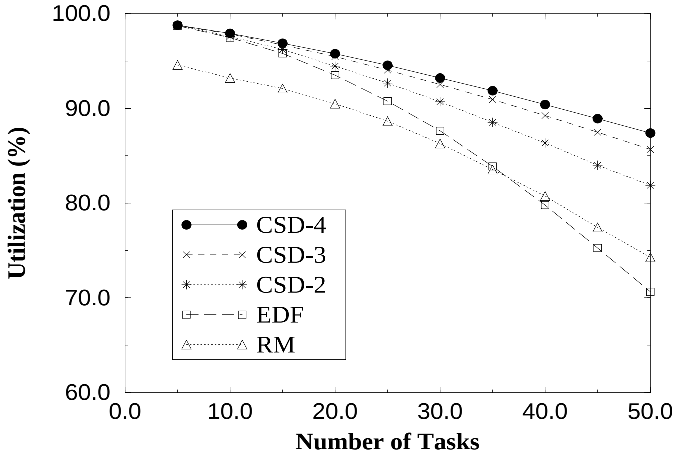

Figure 12: Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 2.

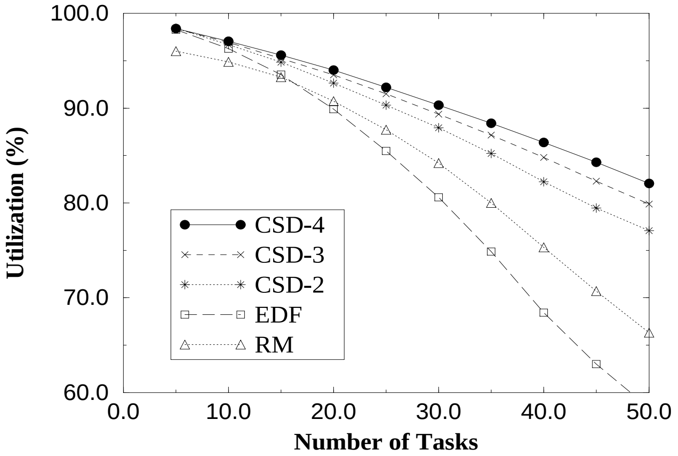

Figure 13: Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 3.

Figures 11–13 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 (especially 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.

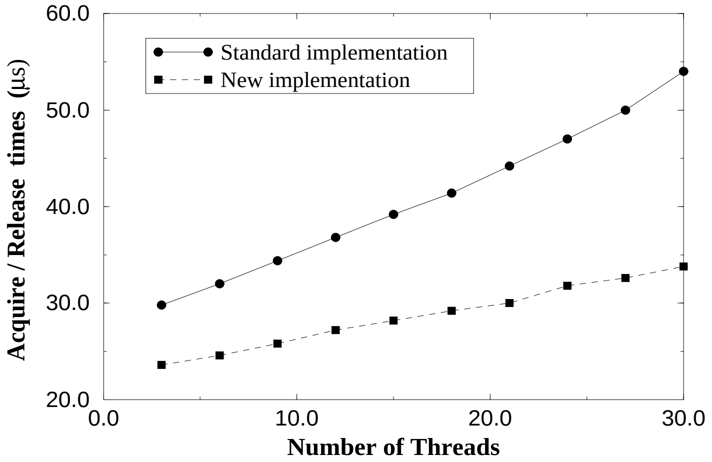

Figure 14: Worst-case performance measurements for DP tasks. The overhead for the standard implementation increases twice as rapidly as for the new scheme.

9.2 Semaphore Scheme

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.

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 14 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 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.

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

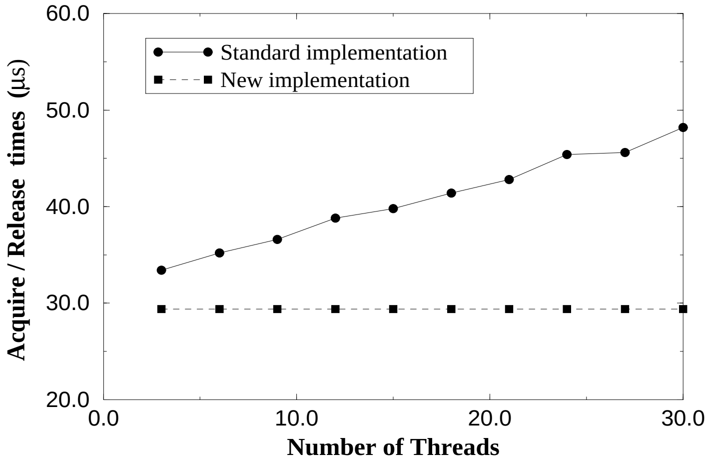

Figure 15: Worst-case performance measurements for FP tasks. The overhead for the standard implementation increases linearly while new scheme has a constant overhead.

	State messages	Mailboxes
`send` (8 bytes)	2.4μs	16.0μs
`receive` (8 bytes)	2.0μs	7.6μs
`receive_slow` (8 bytes)	4.4μs	—

Table 4: Overheads for sending and receiving 8-byte messages.

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 the length of the queue.

9.3 State Messages

Table 4 shows a comparison between the overheads for state messages and for mailbox-based message-passing. The measurements are for message sizes of 8 bytes which are enough to exchange sensor readings and actuator commands in embedded control applications.

Most of the overhead for the state message operations is due to copying the message to and from the SMmailbox, whereas mailbox-based IPC has many other overheads as well (allocation/deallocation of kernel data structures, manipulation of message queues, etc.), which is why state messages clearly outperform mailboxes for small message lengths typical in embedded applications. For example, if an application exchanges 5000 8-byte messages per second (assume 1000 of these are received by tasks with long periods, i.e., they must use `receive_slow`), then mailboxes give an overhead of 118ms/s or 11.8% whereas using state messages results in an overhead of only

	Overhead
`null()` subroutine call	0.2μs
`null()` system call	2.0μs

Table 5: Comparison of overheads for the `null()` subroutine and system calls.

24ms/s or 2.4%. This overhead decreases even further if one message has multiple recipients: for mailboxes, a separate `send` is needed for each recipient while only one `send` is enough for state messages.

9.4 Efficient System Calls

Table 5 shows the comparison between the overheads for a `null()` subroutine call and a `null()` system call. The latter incurs an extra overhead of only 1.8μs because no context switch is needed since the kernel is mapped into each address space.

9.5 Comparison with Commercial RTOSs

The Scientific Research Laboratory (SRL) of Ford Motor Company has 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, 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 16 (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 by SRL standards.

These results show that as far as basic OS overheads are concerned, EMERALDS' implementation is quite efficient. This also gives reassurance that various performance measurements presented earlier in this section are not distorted due to poor implementation of basic OS services.

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

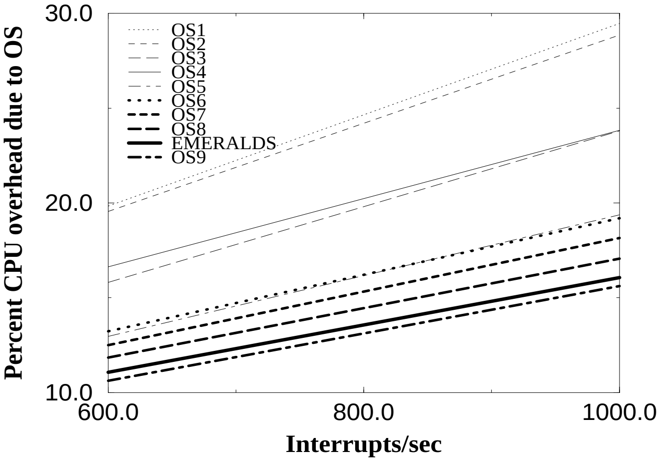

Figure 16: OS overhead due to interrupts, 250 periodic task switches/s and a 4ms clock tick timer.

SRL is planning a more detailed evaluation of these ten RTOSs to measure the effect on OS overheads of varying the number of resources (threads, semaphores, mailboxes, etc.) being used by the application.

10 Conclusions

Small-memory embedded applications are not only becoming more commonplace (automotive, home electronics, avionics, etc.), but the complexity of these applications is increasing as well. As a result, embedded applications which previously managed the hardware resources directly now need embedded RTOSs to handle the increased complexity of the application. These RTOSs must be efficient and small in size to be feasible on the slow/cheap processors used in small-memory applications. Commercial embedded RTOSs rely on optimized code for achieving efficiency, but in the design of EMERALDS, we took a different approach. We identified key OS services which are responsible for a large portion of the OS overhead seen by applications and re-designed these services using new schemes which exploit certain characteristics common to all embedded applications. In the area of task scheduling, 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. For task synchronization, we presented a new implementation for semaphores which eliminates one context switch and reduces priority inheritance overhead to achieve 20–30% improvement in semaphore lock/unlock times. Our semaphore scheme has wide applicability compared to some other optimization schemes which work well only for certain applications. For message-passing, EMERALDS uses the state-message paradigm which incurs 1/4 to 1/5 the overhead of mailbox-based message passing for message sizes typical in embedded applications. Unlike previous schemes for state messages, our scheme bounds the RAM overhead by providing OS support for state messages. Finally, by mapping the kernel into each address space, EMERALDS reduces the overhead of system calls so that the `null()` system call takes only 1.8μs more than a `null()` subroutine call on a 25MHz MC68040.

In the future, we plan to focus on networking issues. We have already investigated fieldbus networking among a small number (5–10) of nodes [12, 13]. Next, we will investigate ways to efficiently and cheaply interconnect a large number (10–100) of clusters of embedded processors. Each cluster can be a small number of nodes connected by a fieldbus. The clusters must be interconnected using cheap, off-the-shelf networks and new protocols must be designed to allow efficient, real-time communication among the clusters. This type of networking is needed in aircraft, ships, and factories to allow various semi-independent embedded controllers (some of which may be small-memory while others may not be) to coordinate their activities.

Acknowledgment

The authors would like to thank Steven Toeppe and Scott Ranville of Ford SRL for the comparative evaluation data for EMERALDS and nine commercial embedded RTOSs.

References

[1] 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.

[2] L. M. Thompson, "Using pSOS+ for embedded real-time computing," in *COMPCON*, pp. 282–288, 1990.

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

[4] *VxWorks Programmer's Guide, 5.1*, Wind River Systems, 1993.

[5] K. G. Shin, D. D. Kandlur, D. Kiskis, P. Dodd, H. Rosenberg, and A. Indiresan, "A distributed real-time operating system," *IEEE Software*, pp. 58–68, September 1992.

[6] 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.

[7] W. M. Gentleman, "Realtime applications: Multiprocessors in Harmony," in *Proc. BUSCON/88 East*, pp. 269–278, October 1988.

[8] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a predictable real-time system," in *Proc. USENIX Mach Workshop*, October 1990.

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

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

[11] *RTXC User's Manual*, Embedded System Products, Inc, 1995.

[12] 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.

[13] 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.

[14] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: a new kernel foundation for UNIX development," in *Proc. Summer Usenix*, pp. 93–113, July 1986.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

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

[24] 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.

[25] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay, "Two years of experience with a μ-kernel based OS," *Operating Systems Review*, pp. 51–62, April 1991.

[26] R. Draves, B. Bershad, R. Rashid, and R. Dean, "Using continuations to implement thread management and communication in operating systems," in *Proc. Symp. Operating Systems Principles*, pp. 122–136, 1991.

[27] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," in *Proc. Symp. Operating Systems Principles*, pp. 95–109, 1991.

[28] J. Liedtke, "Improving IPC by kernel design," in *Proc. Symp. Operating Systems Principles*, pp. 175–188, 1993.

[29] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 896–908, August 1988.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] K. M. Zuberi and K. G. Shin, "An efficient semaphore implementation scheme for small-memory embedded systems," in *to appear in IEEE Real-Time Technology and Applications Symposium*, 1997.

[40] 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.

[41] 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.

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

[43] D. Soloman, *Inside Windows NT*, Microsoft Publishing, 1998.

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

[45] *MC68332 User's Manual*, Motorola Inc., 1993.

[46] 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.