EMERALDS: a small-memory real-time microkernel [*]

Khawar M. Zuberi[†] Padmanabhan Pillai, and Kang G. Shin
University of Michigan
zuberi,pillai,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-crafted 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 and 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 today is no longer limited to large, high-powered, expensive applications. Increasingly, real-time embedded controllers are being used in a wide variety of small control applications, from engine control in automobiles, to voice compression in cellular phones and image stabilization in camcorders. As real-time systems, these embedded controllers execute multiple concurrent computation tasks with strict time constraints (deadlines) that must be met in a predictable manner. Furthermore, these tasks are executed on systems with very minimal hardware — slow processors (15-25MHz) with small memories (tens of kilobytes), often having the entire system on a single integrated circuit. Such restricted hardware is needed to keep production costs down in mass-produced items, and to keep weight and power consumption low in portable and hand-held devices. The combined requirements of real-time computing and low-cost, small-memory platforms has created a need for real-time operating systems (RTOSs) optimized specifically for the small-memory computing devices.

Unfortunately, most conventional RTOSs are not applicable to small-memory embedded controllers. Commercial RTOSs like pSOS [31], QNX [9], and VxWorks [35], as well as research RTOSs like HARTOS [27], the Spring Kernel [29], Harmony [7], and RT-Mach [33] collectively cover a wide range of platforms, from stand-alone systems to multiprocessors and distributed systems. However, most of these RTOSs were designed with relatively powerful systems in mind: processors with several megabytes of memory and networks with at least tens of Mbit/s bandwidth. Even Windows CE, designed for small, hand-held machines, requires over 200KB ROM for a minimal kernel [23]. As a result, most conventional RTOSs are not applicable to small-memory embedded controllers.

Some vendors have targeted small-memory embedded applications with products like RTXC [6], pSOS Select, and a dozen other small, real-time kernels. These RTOSs not only provide predictable services, but also are efficient and small in size, with kernel code size under 20 kbytes. Their approach is 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 specifically 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 primarily on real-time task schedul-

[*] The work reported in this paper was supported in part by by the NSF under Grant MIP-9203895, and by the ONR under Grant 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.

[†] Khawar M. Zuberi is now with Microsoft Corporation.

SOSP-17 12/1999 Kiawah Island, SC
©1999 ACM 1-58113-140-2/99/0012...$5.00

ing, task synchronization through semaphores, and intra-node message-passing.[1] We use 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 to lower OS overheads without compromising OS functionality, thus making more computational resources available for the execution of application tasks. 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 general characteristics of small-memory embedded systems and the constraints they place on RTOSs. We then provide a brief overview of EMERALDS and show how it differs from other RTOSs. Sections 5–7 describe and evaluate our real-time scheduling, synchronization, and message-passing schemes, before concluding in Section 8.

2 Application requirements

Our target embedded applications use single-chip microcontrollers with relatively slow processing cores running at 15–25 MHz. Typical examples are the Motorola 68332, Intel i960, and Hitachi SH-2 controllers. All ROM and RAM are on-chip which limits memory size to 32–128 kbytes, thus limiting useful RTOS kernels to around 20 kbytes code-size. 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).

We expect a typical workload on these systems to consist of 10–20 concurrent, periodic[2] real-time tasks, with a mix of short (<10ms), medium (10–100ms), and long (>100ms) period tasks. As with all embedded control applications, interrupt and I/O services must be provided by the RTOS. Small-memory embedded systems do not use disks, so file system support is not needed in the RTOS. Most other OS services, including task synchronization, task communications, and clock services must be provided.

3 Overview of EMERALDS

EMERALDS is a microkernel RTOS written in the C++ language. Following are EMERALDS' salient features as shown in Figure 1.

- Multi-threaded processes:
  - Full memory protection for threads.
  - Threads are scheduled by the kernel.
- IPC based on message-passing, mailboxes, and shared-memory.

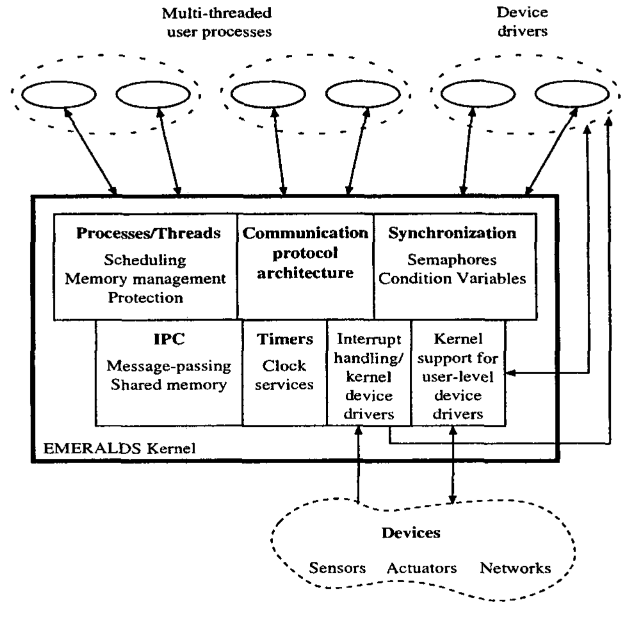

Figure 1. EMERALDS' architecture.

- Semaphores and condition variables for synchronization, with priority inheritance.
- Support communication protocol stacks [41].
- Highly optimized context switching and interrupt handling.
- Support for user-level device drivers.

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, and there is little that the OS designer can do to reduce overhead. The remaining services, however, 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, and communication.

To provide all these services in a small-size kernel, we make use of certain characteristics of embedded applications. First of all, our target applications are in-memory. 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 [38]. With these techniques, EMERALDS provides a rich set of OS services in just 13 kbytes of code (on Motorola 68040).

[1] Inter-node networking issues are discussed in [37, 40] and are not covered in this paper.

[2] Periodic tasks are the major workload of most real-time systems.

4 How is 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 [1], L3 [20], and SPIN [3], researchers focused on optimizing kernel services such as thread management [5, 2], IPC [19], and virtual memory management [25]. Virtual memory is not a concern in our target applications. Thread management and IPC *are* important, but sources of overhead are different for embedded real-time systems, necessitating different optimization techniques.

Thread management is a concern in typical microkernels because either the kernel itself has a large number of threads and switching overhead, and stack use by these threads must be minimized [5], or, in case of user-level threads, the kernel must export the correct interface to these threads [2]. Neither of these apply here, since although EMERALDS has kernel-managed threads, the kernel itself uses no threads, and user threads enter protected kernel mode to simply call kernel procedures, simplifying interfaces. 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 as user-level 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 has not received much attention in the design of most microkernels, but it is of crucial importance in embedded systems. The little research done in this area has focused primarily on multiprocessors [22, 34], 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, and EMERALDS is a first step in this direction.

5 CSD scheduler

Scheduling real-time tasks to ensure that all tasks meet their deadlines is an important part of any RTOS. In small embedded systems, the efficiency of this scheduling takes on great importance, since processing resources are so limited. Until recently, embedded application programmers have primarily used cyclic time-slice scheduling techniques in which the entire execution schedule is calculated off-line, and at run-time, tasks are switched in and out according to the fixed schedule. This eliminates run-time scheduling decisions and minimizes run-time overhead, but introduces several problems as follows:

- Entire schedules must be calculated offline, often by hand, and are difficult and costly to modify as task characteristics change through the design process. Heuristics can be used [12], but result in non-optimal solutions (i.e., feasible workloads may get rejected).
- High-priority aperiodic tasks receive poor response-time because their arrival times cannot be anticipated off-line.
- Workloads containing short and long period tasks (as is common in control systems) or relatively prime periods, result in very large time-slice schedules, wasting scarce memory resources.

As embedded systems use increasingly-complex task sets, cyclic schedulers are no longer suitable for task scheduling. The alternative is to turn to priority-driven schedulers like *rate-monotonic* (RM) [17, 21] and *earliest-deadline-first* (EDF) [28, 21] which use task priorities to make run-time scheduling decisions. They 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, they do incur some overheads, which we seek to minimize in EMERALDS.

The task scheduler's overhead can be broken down into two components: *run-time overhead* and *schedulability overhead*. The run-time overhead is the time consumed by the execution of the scheduler code, and is primarily due to managing the queues of tasks and selecting the highest-priority task to execute. Schedulability overhead refers to the theoretical limits on task sets that are schedulable under a given scheduling algorithm, in the ideal case where run-time overheads are not considered. Together, these overheads limit the amount of useful real-time computation that can be run on a processor.

We analyze the sources of these overheads in RM (which statically assigns higher priority to shorter period tasks [17, 21]) and EDF (which dynamically changes priority, giving highest priority to the earliest-deadline task [28, 21]), and devise a hybrid scheduler that gives better performance than both.

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 each period (unblocks at the beginning of the period, and blocks after executing c_i time), incurring $\Delta t_b + \Delta t_u + 2\Delta t_s$ overhead per period. Overhead is greater if blocking system calls are used; although it is task-dependent, for simplicity, we assume half the tasks use one blocking call per period, thus incurring an average per-period scheduler run-time overhead of $\Delta t = 1.5(\Delta t_b + \Delta t_u + 2\Delta t_s)$. The workload utilization is now calculated as $U = \sum_{i=1}^{n}(c_i + \Delta t)/P_i$.

Now, we calculate Δt for both EDF and RM policies. In EMERALDS, we have implemented EDF as follows. All blocked and unblocked tasks are placed in a single, unsorted queue.[3] A task is blocked and unblocked by changing one entry in the task control block (TCB), so Δt_b and Δt_u are $O(1)$. To select the next task to execute, the list is parsed and the earliest-deadline ready task is picked, so Δt_s is $O(n)$.

RM schedulers usually have a queue of ready tasks, sorted by task priority, and blocking/unblocking involves deleting/inserting tasks into this sorted queue. In EMERALDS, we use a different implementation that permits some semaphore optimizations (Section 6), while maintaining similar run-time costs. All (blocked and unblocked) tasks are kept in a queue sorted by task priority. A pointer highestP points to the first (highest-priority) task on the queue that is ready to execute, so Δt_s is $O(1)$. Blocking a task requires modifying the TCB (as in EDF) and setting highestP to the next ready task. The latter involves scanning the list, so in the worst case Δt_b is $O(n)$. Unblocking, on the other hand, only requires updating the TCB and comparing the task's priority with that of the one pointed to by highestP, changing the pointer if needed. Thus, Δt_u is $O(1)$.

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 (15 or more).

The EDF and RM run-time overheads for EMERALDS measured on a 25MHz Motorola 68040 processor are shown in Table 1. Also shown for comparison is an implementation of RM using a sorted heap. Unless n is very large (58 in this case), the total run-time overhead Δt for a heap is more than for a queue. As most real-time workloads do not have enough tasks, heap implementations are avoided in scheduler structures.

5.2 Schedulability overhead

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.

Consider a workload of n tasks, $\{\tau_i : i = 1, 2, \ldots, n\}$, where each task τ_i has a period P_i and an execution time c_i (assume that a task's relative deadline equals its period). Then, this workload has a utilization $U = \sum_{i=1}^{n} c_i/P_i$. EDF can schedule all workloads with $U \leq 1$ (ignoring run-time overheads)[21], so $U^* = 1$ for EDF. Thus, EDF has zero schedulability overhead.

	EDF - queue	RM - queue	RM - sorted heap
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 1. Run-time overheads for EDF and RM (values are in μs; 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.

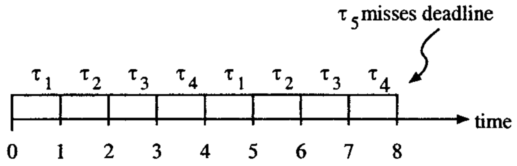
τ_5 misses deadline
τ_1 τ_2 τ_3 τ_4 τ_1 τ_2 τ_3 τ_4
time
0 1 2 3 4 5 6 7 8

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

RM, on the other hand, can have $U^* < 1$. Previous work has shown that for RM, $U^* = 0.88$ on average [17]. As an illustration of nonzero schedulability overhead, consider the workload shown in Table 2. Each task τ_i has deadline $d_i = P_i$.

i	1	2	3	4	5	6	7	8	9	10
P_i (ms)	4	5	6	7	8	20	30	50	100	130
c_i (ms)	1.0	1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.5

Table 2. 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 2, 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.

[3] Simple sorted queues have $O(n)$ insert/delete times, and perform poorly as priorities change often due to semaphore use (Section 6). Heaps have long run times due to code complexity despite $O(\log n)$ insert/remove times since n is rarely very large.

We used this observation as the basis of the combined static/dynamic (CSD) scheduler. Under CSD, τ_1–τ_5 will be scheduled by EDF so that τ_5 will not miss its deadline. The remaining tasks τ_6–τ_{10} will use the low-overhead RM policy. The run-time overhead is less than that of EDF (since the EDF queue's length has been halved), and since in the worst case, CSD simply reduces to EDF, schedulability overhead is same as for EDF (i.e., zero, hence much better than 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 fixed-priority scheduler such as *deadline-monotonic* [18], but for simplicity, we assume RM).

Given a workload $\{\tau_i : i = 1, 2, \ldots, n\}$ sorted by RM-priority (shortest-period-first), let τ_r be the "troublesome" task, the longest period task that cannot be scheduled by RM. Then, tasks τ_1–τ_r are placed in the DP queue while τ_{r+1}–τ_n are in the FP queue. Priority is given to the DP queue, since these tasks have higher RM-priority (shorter periods) than those in the FP queue. A counter keeps track of the number of ready tasks in the DP queue. When the scheduler is invoked, if the counter is non-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

The run-time overhead of CSD depends on whether the task being blocked or unblocked is a DP or FP task. There are four possible cases to consider:

1. DP task blocks: Δt_b is $O(1)$ (same as for EDF). The worst case Δt_s occurs when there are other ready tasks in the DP queue, requiring a scan through the DP queue to select the next task. So, $\Delta t_s = O(r)$.

2. DP task unblocks: Δt_u is $O(1)$. At least one ready task is in the DP queue (the one that was just unblocked), always requiring a parse of the r-long DP queue, so $\Delta t_s = O(r)$.

3. FP task blocks: Δt_b is the same as for RM, but with a shorter queue, so $\Delta t_b = O(n - r)$. Since an FP task was executing and all DP tasks have higher priority, the DP queue cannot have any ready tasks at this point. The scheduler just selects `highestP` from the FP queue, so $\Delta t_s = O(1)$ (same as for RM).

4. FP task unblocks: Δt_u is $O(1)$ (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 $O(1) + O(r) + O(1) + O(r) = 2O(r)$, equivalent to an r-long list parsed twice, whereas the overhead for FP tasks equals $O(n - r) + O(1) + O(1) + O(r) = O(n)$ (n-long list parse once). Therefore, overhead of CSD is significantly less than that of EDF (n-long list parsed twice) and only slightly greater than that of RM (n-long list parsed once), as is corroborated by performance measurements in Section 5.7.

With lower total overheads, CSD can schedule some task sets that are not schedulable under EDF or RM when run-time overheads are included. A detailed analysis of workload schedulability tests for CSD, EDF, and RM that take into account run-time overheads is presented in [36].

5.5 Reducing CSD run-time overhead

The CSD's main advantage is that even though it uses EDF to deliver good schedulable utilization, it reduces run-time overhead by keeping the DP queue short. As the number of tasks in the workload increases, the DP queue length also increases thus degrading CSD performance. We need to modify CSD to keep run-time overhead under control as the number of tasks n increases.

5.5.1 *Controlling DP queue run-time overhead*

Under CSD, the 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 is to split the DP queue into two queues DP1 and DP2. DP1 has tasks with higher RM-priority, so the scheduler gives priority to DP1. Both DP1 and DP2 are expected to be significantly shorter than the original DP queue so that the run-time overhead of the modified scheme (called CSD-3 for its three queues) should be well below that of the original CSD scheme (henceforth called CSD-2).

5.5.2 *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 3, where q is the length of the DP1 queue and r is the total number of DP tasks (so $r - q$ is the length of DP2 queue). The run-time overhead associated with DP1 tasks is $O(r)$, a significant improvement over $O(2r)$ for CSD-2. Since DP1 tasks have the shortest periods in the workload, they execute most frequently, and the reduction in their overheads greatly improves CSD-3 performance over CSD-2.

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

5.5.3 *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

		DP1	DP2	FP
Task Blocks	Δt_b	$O(1)$	$O(1)$	$O(n-r)$
	Δt_s	$O(r-q)$	$O(r)$	$O(1)$
Task Unblocks	Δt_u	$O(1)$	$O(1)$	$O(1)$
	Δt_s	$O(q)$	$O(r-q)$	$O(r-q)$
Total Overhead		$O(r)$	$O(2r-q)$	$O(n-q)$

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

half that of the original DP queue, cutting the run-time overhead of scheduling DP tasks in half.[4] 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%. We should keep only a few tasks in DP1 to keep $\Delta t(DP1)$ small. DP2 will have more tasks making $\Delta t(DP2) > \Delta t(DP1)$, but since DP2 tasks execute less frequently, $\sum_i \Delta t / P_i$ for the two queues will be approximately balanced.
- Once the DP tasks are split into two queues, they no longer incur zero schedulability overhead. Although 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 CSD-3 has non-zero schedulability overhead. Task allocation should minimize the *sum* of the run-time and schedulability overheads. For example, consider the workload in Table 2. Suppose the least run-time overhead results by putting tasks τ_1–τ_4 in DP1 and the rest of the DP tasks in DP2, but this will cause τ_5 to miss its deadline (see Figure 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 off-line exhaustive search (using the schedulability test described in [36]) to find the best possible allocation of tasks to DP1, DP2, and FP queues. The search runs in $O(n^2)$ time for three queues, taking 2–3 minutes on a 167MHz Ultra-1 Sun workstation for a workload with 100 tasks.

5.6 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 5.7), 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.

5.7 CSD performance

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. Table 1 shows run-time overhead for EDF and RM on a 25MHz Motorola 68040 processor; the same overheads apply to CSD DP and FP queues respectively, though fewer tasks are in these queues (only $n-r$ in FP queue, etc.). CSD-x also requires an additional $x * 0.55\mu s$ to parse the list of queues to find a queue with ready tasks.

Our test procedure involves generating random task workloads, then for each workload, scaling the execution times of tasks until the workload is no longer feasible for a given scheduler. The utilization at which the workload becomes infeasible is called the *breakdown utilization* [13]. 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 until the workloads becomes infeasible to find the average breakdown utilizations.

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. For each base workload (with a fixed n), we produce two additional workloads from it by dividing the periods of tasks by a factor of 2 and 3. This allows us to evaluate the impact of varying task periods on various scheduling policies.

To mimic the mix of short and long period tasks expected in real-time embedded systems, we generate the base task workloads by randomly selecting task periods such that each period has an equal probability of being single-digit (5–9ms),

[4] 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 on Motorola 68040) when going from two to three queues.

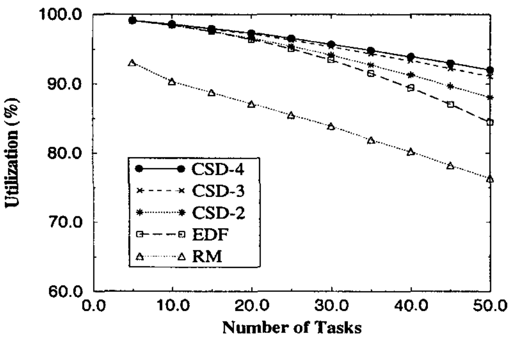

Figure 3. Average breakdown utilizations for CSD, EDF, and RM for base workloads.

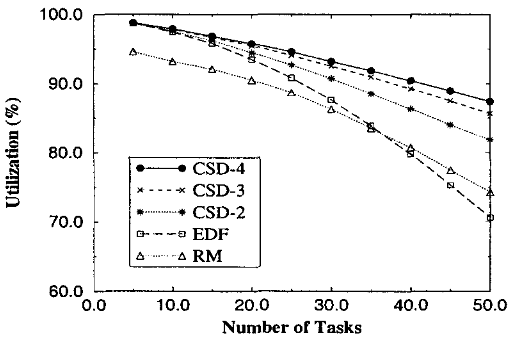

Figure 4. Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 2.

double-digit (10–99ms), or triple-digit (100–999ms). Figures 3–5 show breakdown utilizations for base workloads and when task periods are divided by 2 and 3, respectively. Each point represents the average breakdown utilization for 500 workloads with a fixed n. In Figure 3, task periods are relatively long (5ms–1s). The run-time overheads are low which allows EDF to perform close to its theoretical limits. Even then, CSD performs better than EDF. CSD-4 has 17% lower total scheduling overhead for $n = 15$ and this increases to more than 40% for $n = 40$ as EDF's strong dependency on n begins to degrade its performance.

Figure 4 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 5 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.

Figures 3–5 also show a comparison between three varieties of CSD. They show that even though a significant performance improvement is seen from CSD-2 to CSD-3 (especially for large n), only a minimal improvement is observed

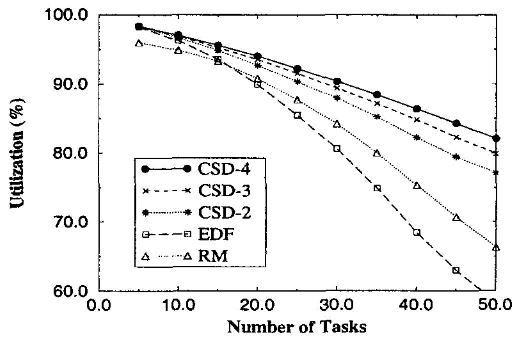

Figure 5. Average breakdown utilizations for CSD, EDF, and RM when task periods are scaled down by a factor of 3.

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.

6 Efficient semaphore implementation

Object-oriented (OO) programming is ideal for designing real-time software, as it models nicely the real-world entities, such as sensors, actuators, and controllers, that real-time systems deal with: the object's internal data represents the physical state (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 simplify the software design process reducing real-time software to a collection of threads of execution, that invoke the methods of various objects [11].

Conceptually, the OO paradigm is very appealing, but does incur some costs. Object methods must synchronize access to object data and ensure mutual exclusion, typically done through semaphores [4, 8, 10]). As semaphore calls are made in every method invocation, semaphore operations are among the most heavily-used OS primitives when OO design is used. This calls for new and efficient schemes for implementing semaphore locking in EMERALDS.

Previous work in lowering the overhead of semaphore operations has focused on either relaxing the semaphore semantics to get better performance [30] or coming up with new semantics and new synchronization policies [32]. 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 take an approach of providing full semaphore semantics (with priority inheritance [26]), but optimizing the implementation of these semaphores by exploiting certain features of embedded applications. We note that the following discussion primarily deals with semaphores used as binary mutual-exclusion locks (mutexes), but is more generally applicable to counting semaphores as well.

6.1 Standard implementation

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

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

Priority inheritance [26] is needed in real-time systems to avoid unbounded priority inversion [32]. Without it, a medium-priority task may indefinitely block a higher-priority task waiting for some low-priority task holding a needed semaphore.

We are most interested in worst-case overheads, which occur when some thread T_2 invokes the `acquire_sem()` call on a semaphore already locked by some lower priority thread T_1. Figure 6 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 7.

Two context switches (C_2 and C_3) are directly due to the `acquire_sem()` call. As context switches incur a significant overhead, eliminating some of these context switches will greatly reduce run-time overhead. Another area of improvement is in the priority inheritance (PI) steps. For DP tasks, the PI steps take $O(1)$ time, since the DP tasks are not kept sorted. However, for tasks in the FP queue, each of the two PI steps will take $O(n - r)$ time, since the tasks must be removed and reinserted according to their new priority. We have addressed both context switch elimination and optimization of the PI step in EMERALDS.

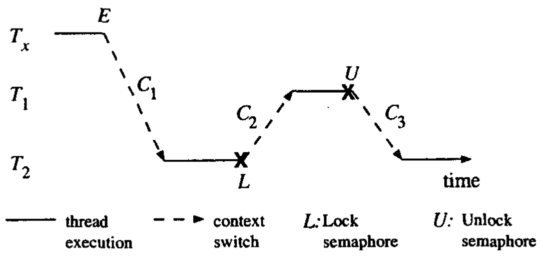

Figure 6. A typical scenario showing thread T_2 attempting to lock a semaphore already held by thread T_1. T_x is an unrelated thread which was executing while T_2 was blocked.

   unblock T_2
   context switch C_1 (T_x to T_2)
(T_2 executes and calls acquire_sem())
   do priority inheritance (T_2 to T_1)
   block T_2
   context switch C_2 (T_2 to T_1)
(T_1 executes and calls release_sem())
   undo priority inheritance of T_1
   unblock T_2
   context switch C_3 (T_1 to T_2)

Figure 7. Operations involved in locking a semaphore for the scenario shown in Figure 6

6.2 Implementation in EMERALDS

Going back to Figure 7, we want to eliminate context switch C_2 [39]. We can do this by letting T_1 execute, rather than switching to T_2 immediately following the unblocking event E. T_1 will go on to release the semaphore and T_2 can be activated at this point, saving C_2 (Figure 8). 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 indicating which semaphore T_2 intends to lock. 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 8 to Figure 6, 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 (Section 6.4).

We also want to optimize the two PI steps for FP tasks, each of which takes $O(n - r)$ time with normal queue manipulation. The first PI step (T_1 inherits T_2's priority) is easily optimized by using the observation that, according to T_1's

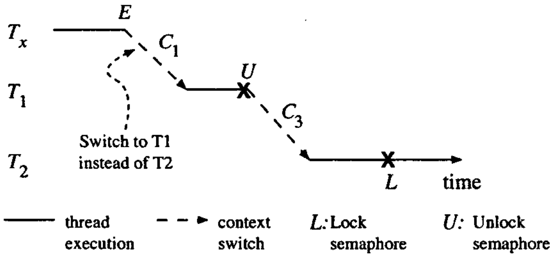

Figure 8. The new semaphore implementation scheme. Context switch C_2 is eliminated.

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, reducing overhead to $O(1)$.

We want to reduce the overhead of the second PI step (T_1 returns to its original priority) to $O(1)$ as well. In EMERALDS, we accomplish this by switching 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" keeping track of 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 is simply put back to its original position in the queue. 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 task 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 fairly straightforward 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 Analysis of new scheme

From the viewpoint of schedulability analysis, there can be two concerns regarding the new semaphore scheme (refer back to Figure 8):

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 until 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 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 8, 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, until it reaches `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 8, 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 execution (Figure 9). 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.

Both of 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.

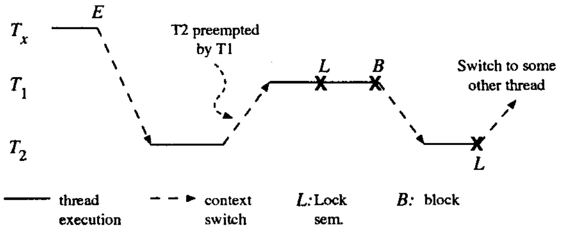

Figure 9. If a higher priority thread T_1 preempts T_2, locks the semaphore, and blocks, then T_2 incurs the full overhead of acquire_sem() and a context switch is not saved.

6.3.1 *Modification to the semaphore scheme*

The problem illustrated in Figure 9 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 9), 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 remaining 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: This case includes waiting on all events generated directly by some executing threads, including releasing semaphores and messaging. 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 10). 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.

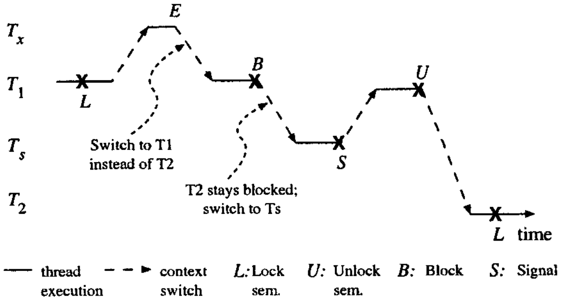

Figure 10. Situation when the lock holder T_1 blocks for a signal from another thread T_s.

Blocking for external events: This includes all triggers not generated by executing code, such as interrupts and external hardware status. 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.

6.4 Semaphore scheme performance

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 11 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 $11\mu s$ 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 both priority inheritance and scheduler task selection overhead are $O(1)$ time). Also, one context switch is eliminated. As a result, the acquire/release overhead stays constant at $29.4\mu s$. For an FP queue length of 15, this is an improvement of $10.4\mu s$ or 26%

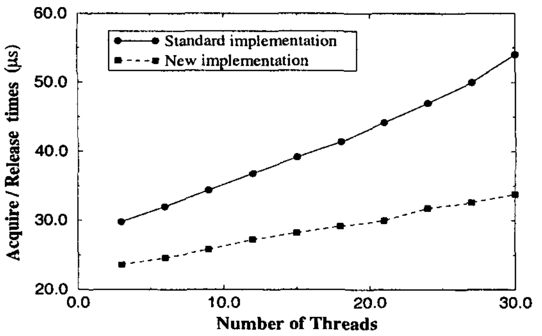

Figure 11. Worst-case performance measurements for DP tasks. The overhead for the standard implementation increases twice as rapidly as for the new scheme.

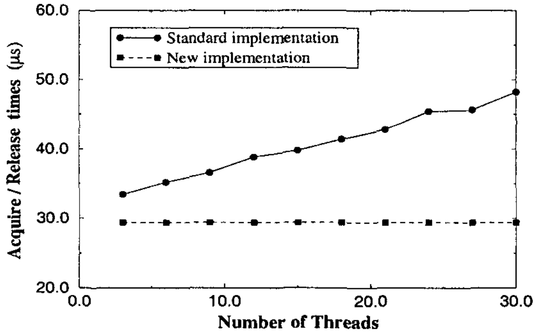

Figure 12. Worst-case performance measurements for FP tasks. The overhead for the standard implementation increases linearly while new scheme has a constant overhead.

over the standard implementation.

In general, our improved semaphore 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.

7 State messages for inter-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 and can introduce subtle, hard-to-trace bugs in the software.

The *state message* paradigm [14] provides the performance of global variables while avoiding the pitfalls. State messages 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.

We have implemented state messages in EMERALDS, optimizing the basic scheme to reduce execution overhead and memory consumption. EMERALDS includes mailbox based message passing as well, since state messages are not meant to replace traditional message-passing, but are meant as an efficient alternative in a wide range of situations.

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 we will henceforth 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 SM-mailbox $SM1$ 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 on state messages

Theoretical work on data sharing without synchronization was first presented by Lamport [16]. State messages were first used in the MARS OS [14] and have also been implemented in ERCOS [24]. The state message implementation used in these systems as described in [15] 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 [15] 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 State message implementation

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:

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

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 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 13 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}.

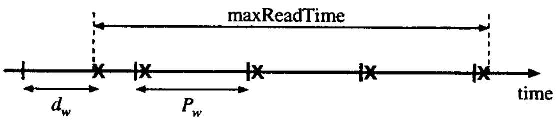

Figure 13. Calculation of x_{max}. Write operations are denoted by **X**. Excluding the first write, there are $\lfloor (maxReadTime - (P_w - d_w))/P_w \rfloor = 4$ writes, so $x_{max} = 5$.

	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.

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.

7.5 State message performance

Table 4 shows a comparison between the overheads for state messages and for mailbox-based message-passing on a 25MHz Motorola 68040. 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 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.

8 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. 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. All of this has been implemented within just 13 Kbytes of code.

EMERALDS has been developed and evaluated primarily on the Motorola 68040 processor. We have also ported it to the PowerPC 505, the Super Hitachi 2 (SH-2), and the Motorola 68332 microcontroller, the last two of which are popular in automotive control applications. EMERALDS is also being evaluated by the Scientific Research Laboratory of Ford Motor Company for use in automotive engine control. They are comparing EMERALDS and various commercial RTOSs, focusing on basic OS overheads related to interrupt handling, context switching, event signaling, and timer services.

In the future, we plan to focus on networking issues. We have already investigated fieldbus networking among a small number (5–10) of nodes [37, 40]. 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.

References

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

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

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

[4] Dijkstra, E. W. Cooperating sequential processes. Tech. Rep. EWD-123, Technical University, Eindhoven, the Netherlands, 1965.

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

[6] Embedded System Products, Inc. *RTXC User's Manual*. Houston, TX, 1995.

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

[8] Habermann, A. N. Synchronization of communicating processes. *Commun. ACM 15*, 3 (March 1972), 171–176.

[9] Hildebrand, D. An architectural overview of QNX. In *Proc. Usenix Workshop on Micro-Kernels and Other Kernel Architectures* (April 1992), pp. 113–126.

[10] Hoare, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (October 1974), 549–557.

[11] Ishikawa, Y., Tokuda, H., and Mercer, C. W. An object-oriented real-time programming language. *IEEE Computer 25*, 10 (October 1992), 66–73.

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

[13] Katcher, D., Arakawa, H., and Strosnider, J. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Software Engineering 19*, 9 (September 1993), 920–934.

[14] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. Distributed fault-tolerant real-time systems: the MARS approach. *IEEE Micro 9*, 1 (February 1989), 25–40.

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

[16] Lamport, L. Concurrent reading and writing. *Communications of the ACM 20*, 11 (November 1977), 806–811.

[17] Lehoczky, J., Sha, L., and Ding, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proc. Real-Time Systems Symposium* (1989), pp. 166–171.

[18] Leung, J. Y.-T., and Whitehead, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation 2*, 4 (December 1982), 237–250.

[19] Liedtke, J. Improving IPC by kernel design. In *Proc. Symp. Operating Systems Principles* (1993), pp. 175–188.

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

[21] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (January 1973), 46–61.

[22] Mellor-Crummey, J., and Scott, M. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (February 1991), 21–65.

[23] Murray, J. Microsoft Windows CE memory use. *http://msdn.microsoft.com/library/backgrnd/html/msdn_memdrft2.htm* (1997).

[24] Poledna, S., Mocken, T., and Schiemann, J. ERCOS: an operating system for automotive applications. In *Society of Automotive Engineers International Congress and Exposition* (February 1996), pp. 55–65. SAE Technical Paper Series 960623.

[25] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers 37*, 8 (August 1988), 896–908.

[26] Sha, L., Rajkumar, R., and Lehoczky, J. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers 39*, 3 (1990), 1175–1198.

[27] SHIN, K. G., KANDLUR, D. D., KISKIS, D., DODD, P., ROSENBERG, H., AND INDIRESAN, A. A distributed real-time operating system. *IEEE Software* (September 1992), 58–68.

[28] STANKOVIC, J., ET AL. *Deadline Scheduling for Real-Time Systems.* Kluwer Academic Publishers, 1998.

[29] STANKOVIC, J., AND RAMAMRITHAM, K. The Spring Kernel: a new paradigm for real-time operating systems. *ACM Operating Systems Review 23*, 3 (July 1989), 54–71.

[30] TAKADA, H., AND SAKAMURA, K. Experimental implementations of priority inheritance semaphore on ITRON-specification kernel. In *11th TRON Project International Symposium* (1994), pp. 106–113.

[31] THOMPSON, L. M. Using pSOS+ for embedded real-time computing. In *COMPCON* (1990), pp. 282–288.

[32] TOKUDA, H., AND NAKAJIMA, T. Evaluation of real-time synchronization in Real-Time Mach. In *Second Mach Symposium* (1991), Usenix, pp. 213–221.

[33] TOKUDA, H., NAKAJIMA, T., AND RAO, P. Real-Time Mach: Towards a predictable real-time system. In *Proc. USENIX Mach Workshop* (October 1990), pp. 73–82.

[34] WANG, C.-D., TAKADA, H., AND SAKAMURA, K. Priority inheritance spin locks for multiprocessor real-time systems. In *2nd International Symposium on Parallel Architectures, Algorithms, and Networks* (1996), pp. 70–76.

[35] WIND RIVER SYSTEMS. *VxWorks Programmer's Guide, 5.1.* Alameda, CA, 1993.

[36] ZUBERI, K. M. *Real-Time Operating System Services for Networked Embedded Systems.* PhD thesis, University of Michigan, 1998.

[37] ZUBERI, K. M., AND SHIN, K. G. Non-preemptive scheduling of messages on Controller Area Network for real-time control applications. In *Proc. Real-Time Technology and Applications Symposium* (May 1995), pp. 240–249.

[38] ZUBERI, K. M., AND SHIN, K. G. EMERALDS: A microkernel for embedded real-time systems. In *Proc. Real-Time Technology and Applications Symposium* (June 1996), pp. 241–249.

[39] ZUBERI, K. M., AND SHIN, K. G. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proc. Real-Time Technology and Applications Symposium* (1997), pp. 25–34.

[40] ZUBERI, K. M., AND SHIN, K. G. Scheduling messages on Controller Area Network for real-time CIM applications. *IEEE Trans. Robotics and Automation* (April 1997), 310–314.

[41] ZUBERI, K. M., AND SHIN, K. G. An efficient end-host protocol processing architecture for real-time audio and video traffic. In *Proc. Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (July 1998), pp. 111–114.