Real-Time Decentralized Control with CAN*

Khawar M. Zuberi and Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
E-mail: {zuberi,kgshin}@eecs.umich.edu

Abstract – **This paper deals with two important concerns in decentralized control: (1) communication over field bus and (2) the transition from a single powerful, centralized controller to many smaller, distributed controllers. The former requires real-time network scheduling for the field bus to ensure that messages get delivered in a timely manner, while the latter requires a low-overhead real-time operating system to cater to the specific needs of distributed control. We present solutions to both of these problems in the form of a network scheduling scheme for the Controller Area Network (CAN) field bus and the EMERALDS real-time operating system which we have designed with the needs of the decentralized control application designer in mind.**

I. INTRODUCTION

Field buses are becoming increasingly popular in industrial automation and robotics for decentralized control of field devices [1, 2]. Field buses allow field devices like sensors, actuators, and controllers to be interconnected at low cost — using less wiring and requiring less maintenance than point-to-point interconnections [2]. This makes it possible to distribute intelligence: instead of having a central processing unit and using expensive cabling to connect it to physically distributed field devices, designers are now using decentralized control. A separate, small microcontroller is used to control each device (or maybe a small group of closely located devices) and a single field bus interconnects all the controllers. This reduces costs in two ways. First, a single field bus replaces the expensive bundles of shielded cables previously used to connect the centralized controller to all the field devices [3]. Secondly, several small processors are cheaper than one powerful processor. Powerful, state-of-the-art processors like the PowerPC and the Pentium can cost more than a thousand dollars per unit whereas any processor which has been in production for a few years like the Motorola 680x0 series or the Intel 80386, can be as cheap as tens of dollars per unit. Moreover, distributing control between multiple processors makes the system more robust and fault-tolerant whereas centralized control suffers from the drawback of a single point of failure.

As an example, consider the control of a robot with five degrees of freedom. The control system may be designed with one 100 MIPS PowerPC with, say, 2 MBytes of memory. Compare this to a distributed design which places a 20 MIPS Motorola 68040 processor with 400 KBytes of memory at each joint and interconnects them by a field bus. This distributed design has the following advantages over the centralized design:

- Five 68040 processors are cheaper than one PowerPC.
- Cabling needed to connect the single PowerPC to the distributed actuators of the joints is now replaced by a single bus interconnecting the 68040s.
- The distributed design is more fault-tolerant since there is no longer a single point of failure of the centralized controller.
- Writing application software is now modularized: low-level details of controlling each joint can now be programmed into each node (processor) separately, while the high-level coordination between the nodes can occur over the field bus.

There are also some disadvantages in going from a centralized to a distributed design. First of all, we must now worry about communication over the field bus. We want to be able to give *a priori* guarantees that all critical messages will meet their deadlines. This is important because if any of these messages miss their deadlines, the coordination between the different joints of the robot may be lost with possibly catastrophic results.

Another disadvantage of using decentralized control has to do with real-time operating systems (RTOSs) [4]. In the distributed design, a separate copy of the RTOS must run on each of the 68040s whereas previously, only one copy ran on the PowerPC. This means that the processing overhead and memory requirements of the RTOS are now five times that of the centralized system. So, an RTOS which may have been acceptable with the centralized design is no longer suitable for the smaller processors: it may well consume most of the memory and CPU cycles available at the smaller 68040s.

This example shows that decentralized control has many advantages over centralized control, but new technology must first be developed to overcome the disadvantages. The very first requirement of distributed systems is a LAN protocol which fits the requirements of industrial automation. These requirements include support for real-time communications and error-free operation in noisy environments. Several architectures have been proposed for such LANs, including Controller Area Network (CAN) [5], SP-50 FieldBus [6], MAP [7], Profibus [8], FIP [9], etc. Of these networks, CAN has gained wide-spread acceptance in the industry [10] — first in the automotive industry and

*The work reported in this paper was supported in part by the Advanced Research Projects Agency, monitored by the US Airforce Rome Laboratory under Grant F30602-95-1-0044, by the NSF under Grant MIP-9203895, and by the ONR under Grant N00014-94-1-0229. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the funding agencies.

then for industrial automation and computer-integrated manufacturing (CIM) as well. CAN has been especially well received by the textile industry and medical equipment manufacturers. CAN is popular because of its low cost (a CAN interface chip costs about $5) and its useful features like reliability in noisy environments and priority-based bus arbitration.

The second step in making wide-spread use of decentralized control a reality is to develop scheduling schemes for these field buses so that real-time messages (both periodic and sporadic) will be guaranteed to meet their deadlines while coexisting with non-real-time messages. These schemes must provide off-line guarantees so that the application designer can be sure that all critical messages will be delivered to their destinations in time.

The third step in making decentralized control feasible is easing the transition from a powerful, centralized processor to many smaller, less-powerful, processors by designing an RTOS optimized for small processors. Currently-available operating systems such as pSOS [11], QNX [12], and VxWorks [13] were designed for general real-time applications and have not been optimized for small- to medium-sized distributed embedded controllers. As a result, their overhead is acceptable only when using powerful processors with several megabytes of volatile and non-volatile memory. But when the switch is made to decentralized control using small, cheap processors with limited memory, these RTOSs are just too large and inefficient to be used on each of these small processors. Since application designers cannot use these RTOSs, they are forced to hand-craft the system-level services needed by their particular application. This not only increases design costs but also leads to inflexible solutions which cannot be used in any other project. This indicates a need to identify the common set of services needed by application designers for these small, distributed processors, and then design an RTOS to provide these services. Since this RTOS will be optimized for small- to medium-sized distributed embedded systems such as the one described in the robot example, it will be able to provide OS services at much lower overhead than RTOSs designed for more general applications. This RTOS must not only be efficient, but it must also be small in size (tens of kilobytes or less) so it can fit in the memory available on the smaller processors.

This paper deals with solutions for steps 2 and 3. First, we present a scheduling scheme we have designed for CAN, called the *mixed traffic scheduler* (MTS) [14]. It is capable of handling both real-time (periodic and sporadic) and non-real-time messages on the same bus and differs from previous scheduling schemes for CAN [15, 16] in that it uses dynamic scheduling to increase schedulable utilization and performance. Next, we present the EMERALDS (Extensible Microkernel for Embedded, ReAL-time, Distributed Systems) real-time operating system [17] which has been designed specifically with small, embedded, distributed controllers in mind.

In the next section we give a brief overview of the CAN protocol. Section III describes the various types of messages that exist in a manufacturing environment. The CAN scheduling scheme to handle these messages is described and evaluated in Section IV. Section V describes the EMERALDS microkernel, and we conclude with Section VI.

II. CONTROLLER AREA NETWORK (CAN)

The CAN specification [5] defines the physical and data link layers of the ISO/OSI reference model. Each CAN frame has seven fields, but we are concerned only with the *identifier* (ID) field. It can be of two lengths: the *standard* format is 11-bits, whereas the *extended* format is 29-bits. It controls both bus arbitration and message addressing, but we are interested only in the former which is described next.

CAN makes use of a wired-OR (or wired-AND) bus to connect all the nodes (in the rest of the paper we assume a wired-OR bus). When a processor has to send a message it first calculates the message ID which may be based on the priority of the message. The ID for each message must be unique to prevent a tie. Processors pass their messages and associated IDs to their bus interface chips. The chips wait till the bus is idle, then write the ID on the bus, one bit at a time, starting with the most significant bit. After writing each bit, each chip waits long enough for signals to propagate along the bus, then it reads the bus. If a chip had written a 0 but reads a 1, it means that another node has a message with a higher priority. If so, this node drops out of contention. In the end, there is only one winner which can use the bus.

III. WORKLOAD CHARACTERISTICS

In computer-integrated manufacturing (CIM), the communicating devices are controllers (CPUs or PLCs), actuators, and sensors. Some of these devices exchange periodic messages (such as drives) while others are more event-driven (such as smart sensors). Moreover, operators may inquire status information from various devices, thus generating messages which do not have timing constraints. So, we classify messages into three broad categories: (1) hard-deadline periodic messages, (2) hard-deadline sporadic messages, and (3) non-real-time (best effort) aperiodic messages.

A. Periodic messages

A common example of this type of messages is servo control of digital drives as in cutting tools. The controller must periodically sample the current position/velocity of the drive and then send appropriate corrections to the drive. Such messages have hard deadlines, because if the update message to the drive is delayed beyond its deadline, the cutting tool may deviate significantly from its desired path, thus ruining the workpiece.

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

B. Sporadic messages

Strictly speaking, all events in the real world are aperiodic in nature. If these events are expected to occur frequently enough, periodic monitoring can be used to detect them and take appropriate action (as in servo control).

There are other events which are not as frequent, such as temperature of a process exceeding a critical threshold. In fact, maximum interval between two such events is unbounded (event may never occur again). In such cases, using periodic messages is a waste because there is nothing to say most of the time. *Smart* *sensors* [18] are most suitable for detecting such events. These sensors have DSP capabilities to recognize events on their own, so they signal the controller only when required. If these messages are treated as purely aperiodic, then we are assuming that they may be released at any time — even in rapid succession. If so, we will not be able to guarantee their delivery by their deadlines. Fortunately, in most real-world situations, there is a minimum interval between consecutive aperiodic events, corresponding to a minimum interarrival time (MIT) for these messages. Such aperiodic messages which have a MIT are called *sporadic messages* [19]. Knowing the MIT of a sporadic message makes it possible to guarantee its delivery even under the worst possible situation.

C. Non-real-time messages

In a manufacturing environment, an operator must be able to monitor the status of every device in the system. Also, some devices (especially drives) need to communicate operational data such as torque/speed limits and diagnostic information. Such messages are non-real-time because they do not have timing constraints. Any communication protocol for manufacturing applications must be able to accommodate such messages while guaranteeing the deadlines of real-time traffic.

D. Low-speed vs. high-speed real-time messages

Messages in a manufacturing setting can have a wide range of deadlines ranging from tens of microseconds for drive control to several seconds or more for temperature sensors. Thus, we further classify real-time messages into two classes: *high-speed* and *low-speed*, depending on the tightness of their deadlines. Note that "high-speed" is a relative term — relative to the tightest deadline d_0 in the workload. So, all messages with the same order of magnitude deadlines as d_0 (or within one order of magnitude difference from d_0) can be considered to be high-speed messages. All others will be low-speed.

IV. THE MIXED TRAFFIC SCHEDULER (MTS)

As stated earlier, access to the CAN bus is controlled by the IDs of competing messages. Then, scheduling messages on CAN corresponds to assigning proper IDs to messages so that real-time messages meet their deadlines in the presence of non-real-time messages.

To see the difficulties faced in scheduling messages on CAN, we must first consider a typical CAN bus interface chip. These chips usually have memory space for one or more messages. When a processor has to send a message, it will calculate the ID and transfer the message (with its ID) to the chip's memory. From then on, the chip will function autonomously: it will compete for the bus with the message ID, and upon getting access, it will transmit the message.

Once a message has been transferred to the CAN chip for transmission, its ID will stay fixed unless the processor comes and updates it. If the ID is to be derived from the message's priority, that priority should stay fixed (at least for reasonably long periods of time). So, fixed-priority scheduling — such as *deadline monotonic* (DM) [20] — is a natural fit for these CAN chips. Under DM, messages with tighter relative deadlines are assigned higher priorities. These priorities then form the message IDs which not only uniquely identify the messages for reception purposes but also schedule the messages in a predictable fashion. However, in general, fixed-priority schemes give lower utilization than other schemes such as non-preemptive earliest-deadline[1] (ED). This motivates us to use ED to schedule messages on CAN under which the message IDs must reflect the absolute message deadline. But as time progresses, absolute deadline values get larger and larger, and eventually they will overflow the CAN ID. This problem can be solved by using some type of a wrap-around scheme (which we present in Section A), but even then, putting the deadline in the ID forces one to use the extended CAN format with its 29-bit IDs. Compared to the standard CAN format with 11-bit IDs, this wastes 20–30% bandwidth, negating any benefit obtained by going from fixed-priority to dynamic-priority scheduling. This makes ED impractical for CAN.

A. Time epochs

As already mentioned, using deadlines in the ID necessitates having some type of a wrap-around scheme. We use a simple scheme which expresses message deadlines relative to a periodically increasing reference called the *start of epoch* (SOE). The time between two consecutive SOEs is called the *length of epoch*, ℓ. Then, the deadline field in the ID of message i will be the logical inverse of $d_i - \text{SOE} = d_i - \left\lfloor \frac{t}{\ell} \right\rfloor \ell$, where d_i is the absolute deadline of message i and t is the current time (it is assumed that all nodes have synchronized clocks). Value of ℓ depends on what fraction of CPU-time the designer is willing to allow for ID updates. Let this fraction be x, M be the MIPS of the CPU, and n be the number of instructions required to do the update. Then $\ell = \frac{n}{xM \times 10^6}$. So at every node, there is a periodic (timer-driven) process which wakes up every ℓ seconds and updates IDs of all ready messages according to the above equation.

B. MTS

MTS attempts to give high utilization (like ED) while using the standard 11-bit ID format (like DM). High-speed messages consume most of the bus bandwidth, so the idea behind MTS is to try to use ED for high-speed messages and DM for low-speed ones. First, we give high-speed messages priority over low-speed and non-real-time ones by setting the most significant bit to 1 in the ID for high-speed messages (Figure 1a). This makes sense because high-speed messages have tighter deadlines, so they should have higher priority than low-speed messages.

A uniqueness field is needed within the ID to ensure that no two IDs are the same. Its length should be $\lceil \log_2(\text{ num high-speed messages }) \rceil$ which would typically be about 5 bits. This leaves 5 bits for the deadline field which are not enough to encode message deadlines. Our solution to this problem is to quantize time into *regions* and encode deadlines according to which region they fall in. To distinguish messages whose deadlines fall in the same region, we use the DM-priority of a message as its

[1] Non-preemptive scheduling under release time constraints is NP-hard in the strong sense [21], meaning that there is no polynomial time scheduler which will always give the maximum schedulable utilization. However, Zhao and Ramamritham [22] showed that ED performs better than other simple heuristics.

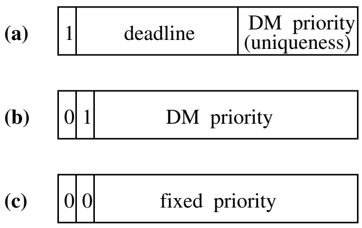

Fig. 1. Parts (a) through (c) show the IDs for high-speed, low-speed, and non-real-time messages, respectively.

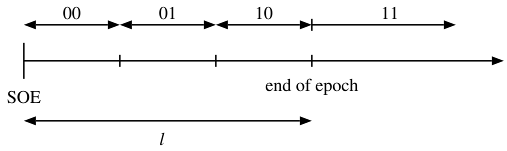

Fig. 2. Quantization of deadlines for $m = 2$.

uniqueness code. This makes MTS a hierarchical scheduler. At the top level is ED: if the deadlines of two messages can be distinguished after quantization, then the one with the earlier deadline has higher priority. At the lower level is DM: if messages have deadlines in the same region, they will be scheduled by their DM priority.

We can calculate length of a region (l_r) as $l_r = \frac{\ell}{2^m - 1}$, where m is the length of the deadline field. This is clear from Figure 2 (shown for $m = 2$). We must reserve one coding for messages whose deadlines fall beyond the end of the current epoch. This leaves $2^m - 1$ codings for deadlines before the end of epoch.

A uniqueness field of 5 bits allows at most 32 real-time messages to be treated as high-speed. To accommodate the remaining (possibly large number of) messages, we use DM scheduling for low-speed messages and fixed-priority scheduling for non-real-time ones, with the latter being assigned priorities arbitrarily. The IDs for these messages are shown in Figures 1 (b) and (c). The second most significant bit gives low-speed messages higher priority than non-real-time ones. In each case, the priority field also acts as the uniqueness code, allowing 512 low-speed and 480 non-real-time messages (32 IDs of non-real-time messages are illegal under the CAN protocol, leaving $512 - 32 = 480$ valid IDs). Schedulability conditions for MTS can be found in [14]. They allow off-line checks to be made to determine whether a message workload is feasible or not.

C. Performance of MTS

We have designed MTS to achieve better performance and schedulability than DM. Since deadlines in MTS are quantized, we would expect the performance of MTS to be close to that of ED if somehow message length did not increase when using ED. So, let ED* be an ideal (imaginary) scheduling policy which works the same as ED but requires only an 11-bit ID. Then ED*'s performance should be an upper bound on MTS's.

To evaluate the performance of MTS, we generated random workloads and tested their feasibility under DM, ED*, and MTS. The base workload consists of high-speed periodic and sporadic messages as shown in Table I (our simulations showed no real

TABLE I
BASIC SIMULATION WORKLOAD.

Class	*Period/MIT*	*Deadline*	*Num. mssgs.*	*Mssg. len*
Periodic	833μs (1.2 kHz)	600.0μs	5	79μs
Periodic	625μs (1.6 kHz)	400.0μs	2	79μs
Sporadic	2s	1200μs	2	47μs

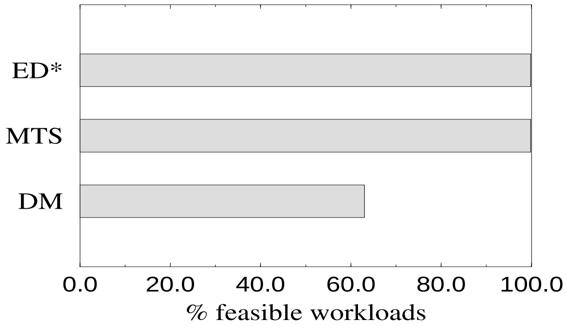

Fig. 3. Percentage of workloads feasible under DM, MTS, and ED* (with five 1.2 kHz periodic messages and utilization = 72.7%).

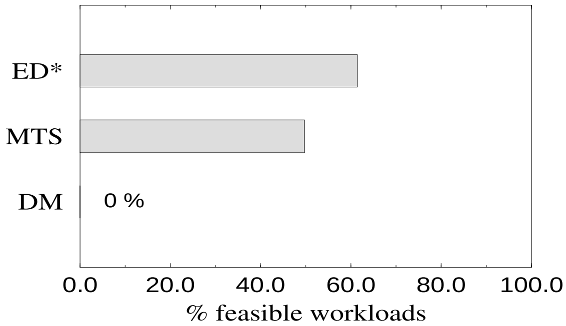

Fig. 4. Percentage of workloads feasible under DM, MTS, and ED* (with six 1.2 kHz periodic messages and utilization = 82.2%).

difference between DM, MTS, and ED* in handling low-speed messages, so we focus on high-speed messages only). This workload gives a utilization of 72.7%. An actual test workload is generated by adding a random value between 0 and 130 μs to each message's deadline. One thousand such workloads were generated and their feasibility tested under the three scheduling schemes for a 1 Mbit/s CAN bus. Figure 3 shows the percentage of workloads feasible under each of the three schemes. It shows that for these workload characteristics, MTS performs as well as ED*, and much better than DM.

To evaluate MTS further, we increased the number of 1.2 kHz periodic messages to six (giving a utilization of 82.2%), and repeated the simulation. Results are shown in Figure 4. We see that even under this high utilization, MTS performs close to ED* while DM is not able to feasibly schedule even a single workload.

V. EMERALDS: DESIGNED FOR EMBEDDED USE

As already mentioned, the transition from a powerful, centralized controller to several smaller, distributed controllers

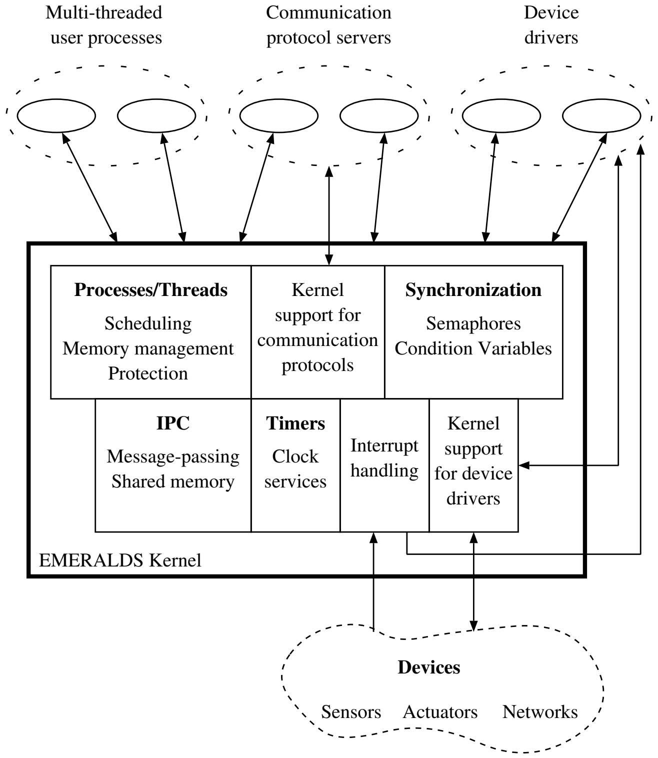

Fig. 5. EMERALDS' architecture.

presents two problems: managing communications over the fieldbus and the need for a small, efficient RTOS for the smaller embedded processors. We presented MTS as a solution to the former problem. Now, we describe a solution for the latter problem in the form of EMERALDS which is a microkernel real-time operating system written in the C++ language. Figure 5 shows the architecture of EMERALDS.

Two things can prevent an embedded systems designer from using an off-the-shelf RTOS: the RTOS is too large and inefficient, or it is not easy to use. To reduce the size and overhead of the kernel, we must omit some features, but this may make the OS difficult to use. This was the biggest challenge in the design of EMERALDS — deciding what to include and what to leave out. Fortunately, embedded systems such as those used in manufacturing have several characteristics which allow OS services to be simplified. In the following, we first outline these characteristics, then summarize those features and design choices which make EMERALDS efficient and easy to use in small- to medium-sized distributed microcontrollers.

A. Simplifying Characteristics

Following are some characteristics of embedded systems that allowed us to simplify the implementation of EMERALDS to achieve our goal of a small, fast kernel.

1) Location of Resources is Known: Most operating systems provide naming services to translate easy-to-remember names into numerical identifiers which may also contain location information. These services are necessary in large, rapidly-changing systems where resources may move about. However, in embedded systems, the designer is very much cognizant of the location of various resources. For example, he knows which threads run on which node (because he is the one who placed them there). This is why EMERALDS does not provide any naming service. For example, to send a message to a remote thread, the local thread must know the node on which the remote thread runs and the identifier of that thread's mailbox. When creating a shared-memory segment, a process must supply a CPU-wide unique identifier. If some other process wants to use that segment it must also know that identifier. Same applies to semaphores and condition variables.

2) Memory-Resident Applications: Our target applications, in general, do not use disks. ROM is used as non-volatile storage and on-board RAM satisfies all run-time memory requirements of the application. So EMERALDS provides neither a file system nor a backing store for virtual memory. This means no device drivers for disks, no disk buffer management, no page fault handlers, or any other such services.

3) Simple Messages: In embedded systems, the most common messages are sensor readings and actuator commands. Threads can exchange such simple messages by talking directly to the network device driver without using any protocol stack, so EMERALDS does not have a built-in communication protocol stack.

B. Extensibility, Efficiency, and Ease of Use

It is not enough for an RTOS to be small in size — it must also be efficient and easy to use. Here we describe those features of EMERALDS which make it fast and user-friendly.

1) Microkernel Architecture: The microkernel architecture of EMERALDS was necessary to allow users the flexibility to install their own communication protocols and device drivers.

As already mentioned, nodes in most embedded systems exchange simple messages for which no protocol stack is needed. Such systems prefer to access the network directly to get maximum bandwidth. However, some applications require more complicated communication protocols to handle duplicate detection, ordering, message fragmentation, etc. In EMERALDS, communication protocols run as user-level servers, so users are free to use protocols which suit their particular applications. The server can even be bypassed completely to directly access the communication network if needed. Moreover, EMERALDS provides the flexibility to have multiple protocol stacks on the same node. For example, one set of processes may require that messages be causally ordered, then they can use one protocol stack. Another set of processes may not have this requirement, so they can use a simpler (and faster) protocol stack.

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

2) Need for Memory Protection: Providing memory protection requires maintaining page tables and programming the memory management unit (MMU). 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. Here we justify providing memory protection in EMERALDS.

The need for memory protection in time-shared systems is indisputable. One user's processes must be protected from all other — possibly malicious — users. But in embedded systems, all processes are cooperative and will never try to intentionally harm another process, so providing memory protection seems 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. 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.

Another benefit of memory protection is easier debugging of application code. During application development, if there is no memory protection, each time one process crashes, the entire CPU may crash. This makes tracking down bugs extremely frustrating and time-consuming. With memory protection, software failures are contained within an address space and can be easily tracked down.

3) Efficient System Calls: The main disadvantage of memory protection is the context switch overhead incurred when making system calls (because the user and kernel usually exist in separate address spaces). This is why some RTOSs omit memory protection so they only have to make subroutine calls to access kernel services; not so with memory protection. We resolved this problem by mapping the kernel into each user-level address space (unlike other OSs in which the kernel runs in its own address space). This way, a system call reduces to a TRAP, then a jump to the appropriate kernel address, without the need to switch address spaces. □

The flexibility EMERALDS offers in writing protocol stacks and device drivers, its small size, and its speed set it apart from other modern RTOSs. EMERALDS also provides many standard features found in today's RTOSs such as multi-threaded processes, message-based IPC, real-time priority-based scheduling, synchronization variables, timers, etc. Important system calls for these functions are described in Tables II–IV. See [17] for more detail.

C. Performance

We have completed a uniprocessor version of EMERALDS for the Motorola 68040 processor. Its size is only 13 KBytes. Table V shows the timing for some EMERALDS operations for a 25 MHz MC 68040. The operations labeled with * involve a context switch to another thread. All other operations return to the calling thread.

TABLE II
PROCESS AND THREAD SYSTEM CALLS.

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

TABLE III
SYSTEM CALLS FOR SEMAPHORES AND CONDITION VARIABLES.

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

Comparing the `null()` system call to the `null()` subroutine call, we see that EMERALDS' technique of mapping the kernel into each address space results in efficient system calls, incurring only a 1.8 μs more overhead than subroutine calls. Even when a context switch to a different address space is required, it incurs less than 12 μs overhead.

VI. CONCLUSION

Manufacturing systems are moving away from centralized control toward decentralized design. This not only reduces costs but also increases modularity and fault-tolerance. However, decentralized control presents its own problems. First and foremost is the issue of managing real-time traffic over the field bus interconnecting the distributed nodes. This requires ensuring that critical real-time messages (both periodic and sporadic) meet their deadlines while sharing bus bandwidth with non-real-time messages. Another problem has to do with the overhead of real-time operating systems. An RTOS feasible for a powerful, centralized controller may not be suitable for the smaller, less powerful processors typically used in decentralized industrial control.

In this paper we presented solutions to both of these problems in the form of the mixed traffic scheduler (MTS) for the CAN field bus and the EMERALDS microkernel operating system for small, embedded processors. Through simulations, MTS was shown to give good performance even under high bus utilizations. It performed close to the idealized earliest-deadline scheduler and far out-performed deadline-monotonic scheduler.

EMERALDS is still in the implementation phase. A uniprocessor version is ready which is only 13 KBytes in size. We are in the process of adding CAN networking to EMERALDS. This will allow us to implement MTS within EMERALDS and make performance measurements.

TABLE IV

MESSAGE-PASSING SYSTEM CALLS. THE LAST TWO CALLS ARE FOR USE BY PROTOCOL SERVERS.

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

TABLE V

TIMING OF VARIOUS OPERATIONS IN EMERALDS.

Operation	*Latency (µs)*
Context switch (kernel to thread)	11.8
`null()` subroutine call	0.2
`null()` system call	2.0
`shm_attach()` (one page mem. allocated)	10.6
`shm_attach()` (attach existing segment)	8.6
`shm_detach()` (one page mem. deallocated)	10.0
`shm_detach()` (just unmap segment)	8.0
`cv_create()`	6.2
`cv_delete()`	5.4
`cv_wait()`*	25.4
`cv_signal()` (no thread waiting)	3.4
`cv_signal()`* (thread waiting)	30.4
`mbox_create()`	6.4
`mbox_delete()`	3.6
`msg_send()` (10 bytes, no server)	12.6
`msg_receive()` (10 bytes, no server)	11.0

VII. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] 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.

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

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

[17] 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.

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

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

[20] 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.

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

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