An Efficient Semaphore Implementation Scheme for Small-Memory Embedded Systems*

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
{*zuberi,kgshin*}*@eecs.umich.edu*

Abstract

In object-oriented programming, updates to the state variables of objects (by the methods of the object) have to be protected through semaphores to ensure mutual exclusion. Semaphore operations are invoked each time an object is accessed, and this represents significant run-time overhead. This is of special concern in cost-conscious, small-size embedded systems — such as those used in automotive applications — where costs must be kept to an absolute minimum. Object-oriented programming can be feasible in such applications only if the OS provides efficient, low-overhead semaphores. We present a new semaphore implementation scheme which saves one context switch per semaphore lock operation in most circumstances and gives performance improvements of 18–25% over traditional semaphore implementation schemes.

1 Introduction

Real-time computing [1] today is no longer limited to large and expensive systems such as planetary exploration robots or the space shuttle. The sharp drop in microprocessor prices over the recent years and the introduction of the microcontroller incorporating a microprocessor with peripherals like timers, memory, and I/O in a single package has led to digital control now being used in much smaller and simpler embedded systems such as in automotive control, cellular phones, and home electronics (camcorders, TVs, and VCRs).

These embedded systems are mass-produced, making low production costs one of the primary concerns in their design. Automotive applications alone account for millions of embedded systems produced annually. At these volumes, extra costs of even a few dollars per unit translate into a loss of millions of dollars overall, so the microcontrollers used in these cost-conscious applications are those which have been in production for several years and their prices have dropped to a few dollars per unit. These microcontrollers have relatively slow processing cores (typically running at 10–30 MHz), small, on-chip RAMs (about 32–64 kBytes, hence the name "small-memory" embedded systems), and all applications are in-memory (there are no disks/file systems in our target applications). This necessitates that any real-time operating system (RTOS) [2] used in these applications must be both time-efficient and memory-efficient.

In this paper, we focus on OS support for object-oriented (OO) programming in embedded systems. OO design gives benefits such as reduced software design time and software re-use [3]. But with these benefits comes the extra cost of ensuring mutual exclusion when an object's internal state is updated. Semaphores[1] [4, 5] are typically used to provide this mutual exclusion. Because semaphore system calls are invoked every time an execution thread enters or exits an object, it becomes essential that the RTOS provide efficient, low-overhead semaphores; otherwise, OO design will not be feasible for embedded applications because of high costs.

*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.

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

Most research in the area of reducing task synchronization overhead has focused on multiprocessors [6, 7]. But our target architectures are either uniprocessor (as in home appliances) or very loosely-coupled distributed systems (as in automotive applications). Even with the latter, threads typically do not need to access remote objects, so our concern is only with improving task synchronization performance for a single processor. Previous work in this area has focused on either relaxing the semaphore semantics to get better performance [8] or coming up with new semantics and new synchronization policies [9]. The problem with this approach is that these new/modified semantics may be suitable for some particular applications but usually they do not have wide applicability.

We took the approach of providing full semaphore semantics (with priority inheritance [10]), but optimizing the implementation of these semaphores by exploiting certain features of embedded applications. As a result, our semaphore scheme has wide applicability within the domain of embedded applications, while significantly improving performance over standard implementation methods for semaphores. We have implemented this new semaphore scheme in the EMERALDS (Extensible Microkernel for Embedded, ReAL-time, Distributed Systems) RTOS [11] which is being developed in the Real-Time Computing Laboratory at the University of Michigan to satisfy the specific memory and performance requirements of small-size embedded systems.

In the next section, we give a brief overview of OO programming as it pertains to embedded real-time systems, focusing on OS support needed for OO programming. In Section 3, we describe our new implementation scheme. Section 4 discusses some limitations of the scheme and ways to overcome these limitations so that our scheme can be used in almost all embedded applications. Section 5 evaluates the performance of our new scheme, and we conclude with Section 6.

2 Objects and Semaphores in Embedded Real-Time Systems

An object is a collection of private state information (or data) and a set of methods which manipulate the data. Objects are ideal for representing real-world entities: the object's internal data represents the physical state of the entity (such as temperature, pressure, position, RPM, etc.) and the methods allow the state to be read or modified. These notions of encapsulation and modularity greatly help the software design process because various system components such as sensors, actuators, and controllers can be modeled by objects. Then, under the OO paradigm, real-time software is just a collection of threads of execution, each invoking various methods of various objects [12].

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

2.1 Active and Passive Object Models

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

Under the active object model [13], one or more server threads are permanently bound to an object. When a client thread invokes a method, a server thread executes the method on behalf of the client.

With the passive object model [13], objects do not have threads of their own. To invoke a method, a thread will enter the object, execute the method, and then exit the object.

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

With the passive object model, multiple threads can be inside the same object at one time, so they must synchronize their activities. Semaphores [4, 5] are commonly used for this purpose (e.g., to provide the monitor construct [14]). Even though locking based on semaphores incurs time overhead, it is decidedly much more memory-efficient than the active object model.

2.2 OO Design Under EMERALDS

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

3 An Efficient Semaphore Implementation Scheme

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

3.1 Standard Semaphore Implementation

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

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

If the semaphore happens to be already locked by some other thread, the thread making the semaphore lock system call is put on a wait queue and is blocked. It is unblocked as part of the semaphore release operation and it then proceeds to reserve the semaphore for itself.

If the caller is to block, priority inheritance [9, 10] also takes place under which the current lock holder thread's priority is increased to that of the caller thread (if the former is less than the latter). This is needed to avoid unbounded priority inversion [10]. If a high-priority thread T_h calls `acquire_sem()` on a semaphore already locked by a low-priority thread T_l, the latter's priority is temporarily increased to that of the former. Without priority inheritance, a medium priority thread T_m can get control of the CPU by preempting T_l while T_h remains blocked on the semaphore, thus causing priority inversion. With priority inheritance, T_l will keep on running until it unlocks the semaphore. At that point, its priority will go back to its original value, but now T_h will be unblocked and it can continue execution.

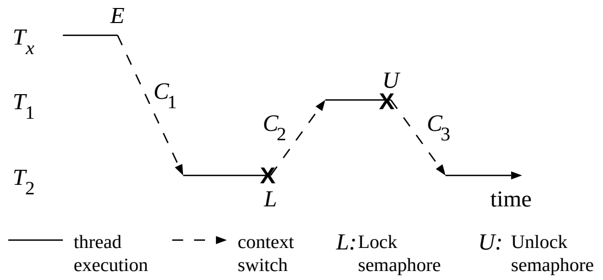

Figure 1: A typical scenario showing thread T_2 attempting to lock a semaphore already held by thread T_1. T_x is an unrelated thread which was executing while T_2 was blocked. Conceptually, T_x can be T_1.

First of all, notice that if the semaphore is free when `acquire_sem()` is called, then the semaphore lock operation has very little overhead[2]. In fact, for this case, only one counter has to be incremented and some other variables updated.

The situation is very different when the semaphore is already locked by thread T_1 when some thread T_2 invokes the `acquire_sem()` call. Figure 1 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.

We observe that it is these context switches which are responsible for much of the overhead (as much as 40–50%) associated with locking and unlocking semaphores (see Section 5 for timing measurements).

Schedulability Analysis: In all critical real-time systems, an off-line guarantee is needed that the task workload is feasible and all execution deadlines will be met at run-time. Schedulability tests [15–17] are used for this purpose. The worst-case execution time of each task is first calculated and then the appropriate schedulability test is used to determine feasibility.

The worst-case execution time for `acquire_sem()` occurs when the semaphore is already locked when

[2]This is especially true in EMERALDS where system call overhead is comparable to subroutine call overhead even with full memory protection between processes [11].

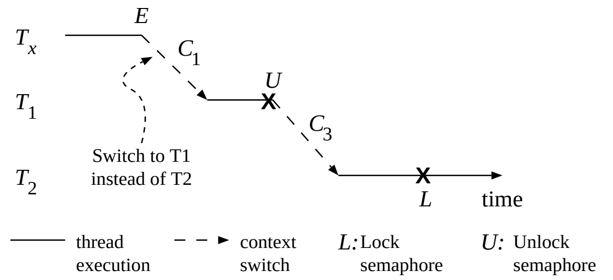

Figure 2: The new semaphore implementation scheme. Context switch C_2 is eliminated.

the system call is made. This means that the context switches C_2 and C_3 shown in Figure 1 must be included when calculating worst-case task execution times.

Any scheme to make semaphores more efficient must target this worst-case scenario. The other scenario (semaphore happens to be free when `acquire_sem()` is called) is quite efficient as is and is of no concern when calculating worst-case execution times, so, from now on, we focus on optimizing the worst-case scenario when the semaphore is already locked by some thread when `acquire_sem()` is called.

3.2 Semaphore Implementation in EMERALDS

Going back to Figure 1, we want to eliminate context switch C_2. Recall that the progression of events was as follows: T_2 blocks (say, waiting for an event such as a message arrival; call this event E); some other threads execute; then event E occurs and T_2 is unblocked. Now, the next blocking call T_2 is to make is to acquire semaphore S. Under our scheme — as part of the blocking call just preceding `acquire_sem()` — we instrument the code (using a code parser described later) to indicate which semaphore T_2 intends to lock (semaphore S in this case). When event E occurs and T_2 is to be unblocked (Figure 2), 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, waiting for S. 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 2 to Figure 1, we see that context switch C_2 is eliminated. The semaphore lock/unlock pair of operations now incur only one context switch instead of two, resulting in considerable savings in execution time overhead (see Section 5 for performance results).

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 possible to write a parser which examines the application code and automatically inserts the correct semaphore identifier into the argument list of blocking calls just preceding `acquire_sem()` calls. Parser design issues are discussed further in Section 4.

Schedulability Analysis for the New Scheme: From the viewpoint of schedulability analysis, there can be two concerns regarding the new semaphore scheme (refer back to Figure 2):

1. What if thread T_2 does not block on the call preceding `acquire_sem()`? This can happen if event E has already occurred when the call is made.
2. Is it safe to delay execution of T_2 even though it may have higher priority than T_1 (by doing priority inheritance earlier than would occur otherwise)?

Regarding the first concern, if T_2 does not block on the call preceding `acquire_sem()`, then a context switch has already been saved. For such a situation, T_2 will continue to execute till it reaches `acquire_sem()` and a context switch will occur here. 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 runtime do not really matter for calculation of worst-case execution times for schedulability analysis.

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

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

Figure 3: A typical sensor-controller-actuator loop commonly found in embedded control applications

on T_2 will miss its deadline, so schedulability of the task workload is not adversely affected.

4 Applicability of the New Scheme

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

1. The code parser is unable to identify which semaphore is to be locked next due to conditional constructs such as loops with a variable number of iterations or `if-then-else` statements.

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

3. The lock holder T_1 (Figure 2) blocks after event E but before releasing the semaphore. Then with standard semaphores, T_2 will be able to execute, but under our scheme it cannot which may lead to T_2 missing its deadline.

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

4.1 Code Parser Issues

Most threads in embedded systems execute sensor-controller-actuator loops as shown in Figure 3. Each device (sensor or actuator) is represented by an object protected by its own semaphore. Each device may be a real sensor/actuator or a logical one representing several devices being controlled as one group.

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

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

but this does not change the order in which semaphores are locked because all methods of an object are protected by the same semaphore. In other words, most embedded applications are structured as in Figure 3, and for such a structure, the parser can easily determine which semaphore is to be locked after a given blocking call.

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

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

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

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

4.2 Consecutive `acquire_sem()` Calls

Going back to Figure 3, the bodies of the methods invoked by the thread may contain blocking calls, especially condition variable and message-passing calls. In these calls, the parser will insert the identifier of the upcoming `acquire_sem()`. But if such calls are not present, then two or more `acquire_sem()` calls can occur with no other blocking call in between them. Then, only one context switch will be saved per pair of `acquire_sem()` calls. This leads to an interesting avenue for future research. Our scheme can be generalized so that the blocking call at the end of the control loop will not unblock until *all* the semaphores needed by the thread for execution become available. In other words:

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

This is somewhat similar to the Spring kernel's notion of reserving all resources a task needs before letting the task execute [18], but with an important difference: the Spring kernel executes tasks non-preemptively while under our proposal, threads execute preemptively. This allows higher priority threads to preempt a given thread (giving good schedulable utilization) while reducing the number of context switches seen by the thread to wait for resources (giving shorter execution times). However, advance reservation of all semaphores will increase scheduler complexity and may also adversely affect task schedulability. Impact of these issues on performance must be studied to determine the viability of this extension.

4.3 Blocking by the Lock Holder Thread

Going back to Figure 2, suppose the lock holder T_1 blocks after event E but before releasing the semaphore. With standard semaphores, T_2 will then be able to execute (at least, till it reaches `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 2, T_1 had priority less than that of T_2 (call this case A). A different problem arises if T_1 has higher priority than T_2 (call it case B). Suppose semaphore S is free when event E occurs. Then T_2 will become unblocked and it will start executing (Figure 4). 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.

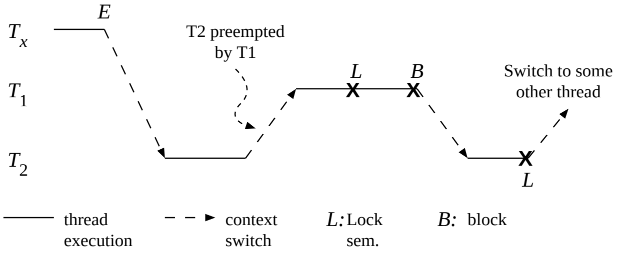

Figure 4: If a higher priority thread T_1 preempts T_2, locks the semaphore, and blocks, then T_2 incurs the full overhead of `acquire_sem()` and a context switch is not saved.

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

Modification to the Semaphore Scheme: For the situation shown in Figure 4, 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 4), the OS first checks if S is available or not before unblocking T_2. Now, let us extend the scheme so that the OS adds T_2 to a special queue associated with S. This queue holds the threads which have completed their blocking call just preceding `acquire_sem()` but have not called `acquire_sem()` yet.

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. This way, T_2 is prevented from executing while S is locked which results in the same behavior as in case A. Also, if done properly, addition and removal of threads from this queue incurs very little overhead

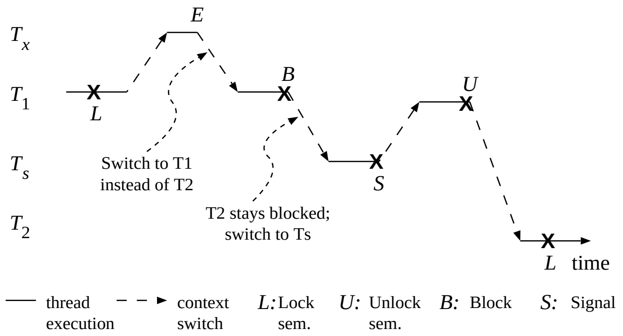

Figure 5: Situation when the lock holder T_1 blocks for a signal from another thread T_s.

(about 5–7 μs on a 25 MHz MC 68040 without caches and just 1–2 μs with caches).

With this modification, the only remaining concern (for both cases A and B) is: if execution of T_2 is delayed like this while other threads (of possibly lower priority) execute, then T_2 may miss its deadline. This concern is addressed next.

Applicability under Various Blocking Situations: There can be two types of blocking:

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

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

Blocking for Internal Events: The typical scenario for this type of blocking is for thread T_1 to enter an object (and lock semaphore S) then block waiting for a signal from another thread T_s. Meanwhile, T_2 stays blocked (Figure 5). The question is: is it safe to delay T_2 like this even if T_s is lower in priority than T_2? The answer is yes, because T_2 cannot lock S till T_1 releases it, and T_1 will not release it till it receives the signal from T_s, so even though T_s may be lower in priority than T_2, it is safe to let T_s execute earlier. This leads to T_1 releasing S earlier than it would otherwise which leaves enough time for T_2 to complete by its deadline.

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

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

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

5 Performance Evaluation

To measure the improvement in performance resulting from our new semaphore scheme, we implemented it under EMERALDS and measured performance on a 25 MHz Motorola 68040 processor [19].

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

5.1 The Test Procedure

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

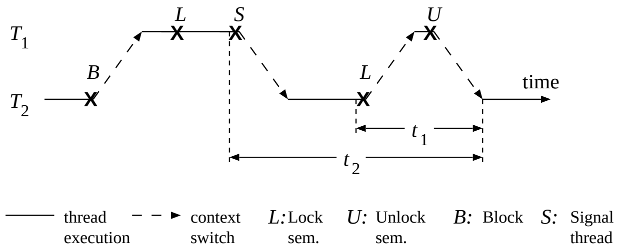

Figure 6: Test procedure for standard semaphores. Interval t_1 is the overhead for acquire/release operations.

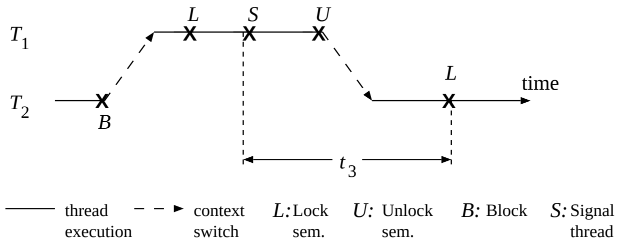

Figure 7: Test procedure for the new semaphore scheme.

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

To get this behavior, we use two threads in our tests, T_1 and T_2, with T_2 having higher priority. For the standard semaphore implementation, the test proceeds as shown in Figure 6. T_2 executes first and blocks waiting for a signal from T_1. T_1 executes, locks semaphore S, and signals T_2 which is unblocked, goes on to execute `acquire_sem()`, and priority inheritance occurs. Thread T_1 then releases S, its priority goes back to its original value, and a context switch occurs back to T_2. We measure interval t_1 which is the time for an acquire plus a release and includes relevant context switches.

We repeated this test with the new semaphore scheme. Figure 7 shows the new sequence of events. In this case, priority inheritance is done by the OS when T_1 signals T_2, so T_1 continues after the signal and unlocks S. T_1's priority goes back to its original value, T_2 is unblocked, and it goes on to lock S without needing any more context switches. Then the difference $t_2 - t_3$ (Figures 6 and 7) represents the improvement due to the new scheme and $t_1 - (t_2 - t_3)$ is the overhead for acquire/release under the new scheme. Note that we cannot directly measure the acquire/release overhead for the new scheme because priority inheritance occurs well before the rest of the acquire operation.

5.2 Experimental Results

EMERALDS uses dynamic thread scheduling,[3] so the context switch overhead depends on the number of threads in the system. Because our semaphore scheme eliminates one context switch, the improvement in performance depends on the number of threads in the scheduler queue. Experience shows that typical embedded applications have about 10–20 threads (anything more will consume too much memory for stack, thread control block, etc.). For evaluation purposes, we chose a slightly wider range of thread counts, from 3 to 30. For each case, two of the threads are T_1 and T_2 mentioned in Section 5.1 while the remaining threads just execute infinite loops and serve only to fill the scheduler queue.

First, we ran our tests on the MC 68040 with caches disabled (to simulate processors which do not have caches). Figure 8 shows the results for both the standard and the new semaphore implementation schemes. Since the context switch overhead is a linear function of the number of threads, the acquire/release times also increase linearly with the thread count. But the standard implementation's overhead involves two context switches while our new scheme incurs only one, which is why the measurements for the standard scheme have a slope twice that of our new scheme. For a typical thread count of 15 threads, our new scheme gives savings of about 35 μs over the standard implementation and these savings grow even larger as the thread count increases.

We repeated our tests on the MC 68040 with both instruction and data caches enabled. The results are shown in Figure 9. Again, the results for our new scheme have a slope roughly half that of the standard scheme. But, notice that the percent improvement in performance is more with caches enabled than with caches disabled as shown in Figure 10. The reason is that the context switch overhead is greater (relatively speaking) when caches are used because of the cache misses incurred when a new thread begins to execute. The old context is flushed out to main memory, the new context is fetched, and this increases the context switch overhead, which is why our scheme gives greater improvement over the standard implementation with caches enabled than with caches disabled.

These results show that our new scheme improves

[3] With priority inheritance, thread priorities change so often that it makes no sense to have fixed priority scheduling.

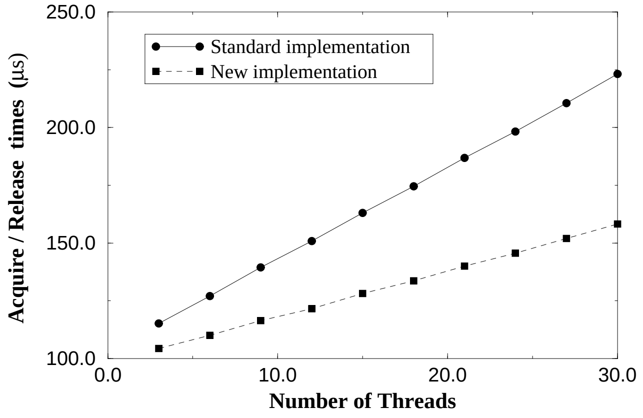

Figure 8: Performance measurements with caches disabled. The overhead for the standard implementation increases twice as rapidly as for the new scheme.

performance by 10–40%, depending on the number of threads in the application and whether caches are used or not. Since most embedded applications have about 10–20 threads, they can expect improvements of about 18–25% (without caches) or 25–30% (with caches).

6 Conclusion

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

In this paper, we presented a new semaphore implementation scheme which saves one context switch per semaphore acquire/release pair of operations (for most scenarios found in embedded applications) and improves performance by 18–25%. We used the fact that in small-size embedded applications, the identifiers of semaphores are fixed at compile time. Then, during run-time, we use these known identifiers to do ahead-of-time checks on the status of semaphores (whether they are available or not). If a semaphore is unavailable, we delay the execution of threads until the semaphore is released. This way, the semaphores are always available when threads actually make the `acquire_sem()` system call and the call does not block, saving one context switch.

Future work includes studying the advantages and disadvantages of extending our scheme so that instead of looking ahead only to the next `acquire_sem()` call,

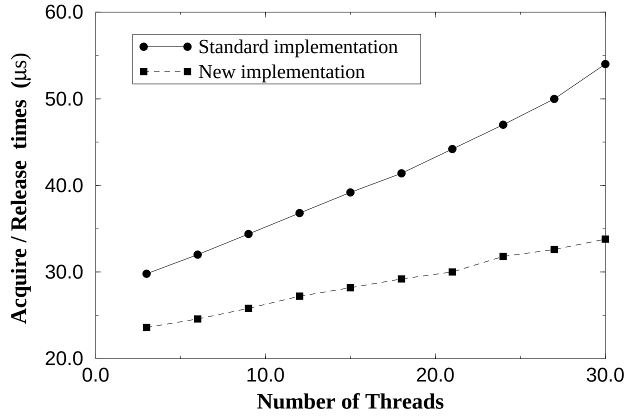

Figure 9: Performance measurements with caches disabled.

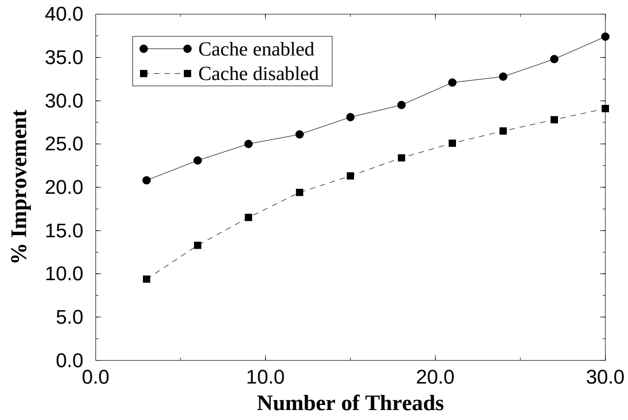

Figure 10: Percent improvement in performance due to our new semaphore scheme.

the scheduler will consider *all* the semaphores a thread may need to execute so that all resource conflict-related context switches are eliminated. Also, in this paper we focused only on improving the semaphore lock operation. In the future, we plan to investigate optimizations related to the release operation to get further improvements in synchronization overheads.

References

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

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

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

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

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

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

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

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

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

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

[11] 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.

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

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

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

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

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

[17] Q. Zheng and K. G. Shin, "On the ability of establishing real-time channels in point-to-point packet-switched networks," *IEEE Trans. Communications*, pp. 1096–1105, February/March/April 1994.

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

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