International Journal of Automotive Technology, Vol. ?, No. ?, pp. ?–?(year)

Reducing Memory Footprint of OSEK-based Systems via Stack Sharing and Light-Weight Ready Queues

Daedong Park[1)], Jonghun Yoo[1)], Jiyong Park[1)] and Seongsoo Hong[1),2)*]

[1)] School of Electrical Engineering and Computer Science, Seoul National University, Republic of Korea
[2)] Department of Intelligent Convergence Systems, The Graduate School of Convergence Science and Technology, Seoul National University, Republic of Korea

(*Received 8 March 2010; revised 7 October 2010*)

ABSTRACT– OSEK OS is an open real-time operating system standard for ECU software in vehicles. Since it was originally designed to be used in an extremely resource-constrained environment, an OSEK compliant operating system must incur low processing overhead and memory usage. Unfortunately, as OSEK OS evolves over time, it now specifies non-trivial kernel features along with multiple conformance classes and application modes. This may lead to unwanted dynamic resource usage in a resultant system unless the standard is carefully interpreted and designed into an OSEK OS implementation. In this paper, we analyze the various kernel features of OSEK OS and their interactions in order to identify places in the standard that can warrant further resource usage optimization. We particularly attempt to reduce run-time memory footprint. Based on our analyses, we present two kernel mechanisms: (1) stack sharing among tasks and (2) light-weight ready queue handling specialized for OSEK OS conformance classes. We also offer implementation methods for the proposed mechanisms by extending OIL and associated tools. Finally, we show the effectiveness of the proposed mechanisms via extensive experiments. Our mechanisms allow OSEK-based systems to take up only 36% of the memory requirement of conventional OSEK-based systems on average.

KEY WORDS: OSEK OS, RTOS, Memory optimization, Automotive software

1. INTRODUCTION

OSEK OS (OSEK Group, 2004) is an open real-time operating system standard for ECU software in vehicles. It has attracted wide industry adoptions since its first release in 1993 due to its capabilities for real-time multitasking, effective I/O control and abstraction, resource management and flexible timer/event handling. In addition to such functionalities, it yields a run-time system with extremely low memory footprint and timing overhead. As a result, an OSEK compliant operating system is ideal for ECU software that has to run in an extremely resource-constrained environment where only low-end microcontrollers and a small amount of physical memory are available (Yoon *et al*., 2005).

Unfortunately, OSEK OS has evolved over time, and thus it now specifies non-trivial kernel features along with multiple conformance classes and application modes. Once being implemented, these features interact with each other in complex ways depending on a given conformance class and application mode. This may lead to unwanted resource usage in a resultant system unless the standard is carefully interpreted and designed into an OSEK OS implementation. This problem can get aggravated since OSEK OS specifies only kernel interfaces and functionalities and does not mention internal kernel structures and mechanisms in detail. Consequently, the resource requirements of an OSEK compliant operating system and its applications become dependent on a specific implementation. Some OSEK compliant operating systems may fall behind with others in terms of dynamic memory requirements and timing overhead if resource-saving kernel mechanisms are not exploited.

In this paper, we analyze the kernel features specified in the OSEK OS standard and their interactions in order to identify places for optimizing the dynamic memory footprint of OSEK OS implementations and applications. We particularly examine task and resource subsystems since they are collectively responsible for task

* *Corresponding author*: sshong@redwood.snu.ac.kr

scheduling and affect the ready queue design of OSEK compliant operating systems. We further examine OIL (OSEK Implementation Language) configurations, conformance classes and application modes. An OIL configuration describes task attributes of an application to the OIL interpreter and C code generator. A conformance class determines the range of the possible characteristics that an application task can have. An application mode defines a task set that can run together since OSEK OS allows only tasks with the same application mode to run concurrently at a given time.

Based on our analyses, we propose two kernel mechanisms for reducing the dynamic memory requirement of OSEK-based systems. The first one is stack sharing among multiple tasks. We first reveal that stack spaces for two mutually exclusive tasks can be safely shared by them. We then present a list of task conditions for stack sharing by enumerating combinations over task attributes, conformance classes and application modes. The second is light-weight ready queue handling. Based on the analysis of conformance classes and task synchronization, we propose a ready queue design specialized for conformance classes. Then, we propose a priority reassignment scheme which is needed by the proposed ready queue design.

We have implemented the proposed mechanisms into an OSEK compliant operating system that we developed for an evaluation board equipped with an MPC5554 processor. We have extended the OIL to support our mechanisms and augmented the OIL interpreter and C code generator. Then, we have performed extensive experiments and compared the memory requirement of our implementation with that of others to show its effectiveness. The results show that our mechanisms allow OSEK-based systems to take up only 36% of the memory requirement of conventional OSEK-based systems on average.

This paper is organized as follows. In Section 2, we analyze the kernel features and present a list of task conditions for stack sharing. Section 3 proposes the light-weight ready queue handling mechanism specialized for conformance classes. Section 4 describes extensions to the OIL for the proposed mechanisms. Section 5 reports on the results of experimental evaluation. Section 6 describes related work and Section 7 provides our conclusion.

2. STACK SHARING AMONG MULTIPLE TASKS

OSEK OS specifies a non-trivial kernel structure with multiple conformance classes and application modes. Once being implemented, these features interact with each other in complex ways depending on a given conformance class and application mode. In this section, we analyze the kernel features of OSEK OS and their interactions in order to identify places for optimizing the dynamic memory footprint of OSEK-based systems. We particularly examine mutually related task and resource subsystems since they are collectively responsible for task scheduling and affect ready queue design. We further examine OIL configurations, conformance classes and application modes in turn. An OIL configuration describes task attributes of an application to the OIL interpreter and C code generator. A conformance class determines the range of the possible characteristics that an application task can have. An application mode defines a task set that can run together since OSEK OS allows only tasks with the same application mode to run concurrently at a given time.

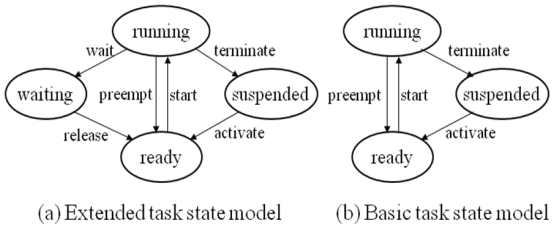


Figure 1. OSEK task state transition diagrams.

2.1. Examining Task Attributes and Application Modes

OSEK OS specifies four attributes for each application task: task type, multiple activations, preemption and scheduling policy. As for task types, OSEK OS defines a basic task type and an extended task type where an extended task can invoke the `WaitEvent()` system call while a basic task cannot. Upon an invocation to `WaitEvent()`, an extended task stops its current execution and puts itself into a waiting state where it waits until other tasks or alarms wake it up. Obviously, a basic task cannot be in a waiting state. Figure 1 gives the state transition diagrams of an extended task and a basic task.

OSEK OS allows multiple activations for a basic task. An activation is a command that triggers a task state transition from a suspended state to a ready state.

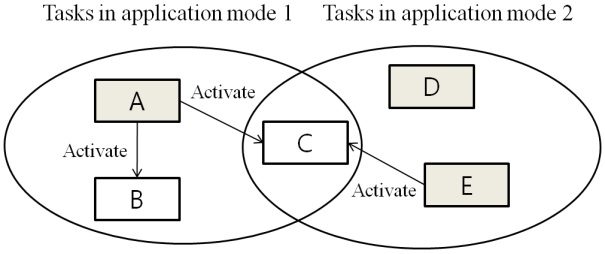
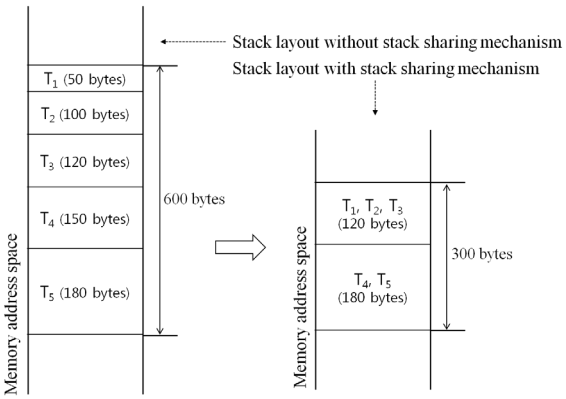


Figure 2. Task activations in two different application modes.

Task name	T_1	T_2	T_3	T_4	T_5
Type (B: Basic, E:Extended)	B	B	B	B	E
preemptability (P: Preemptable, N: Non-preemptable)	N	P	N	P	P
Stack size (bytes)	50	100	120	150	180
Internal resource	-	ResA	ResA	-	
Priority	20	20	10	30	40
Application modes	A, B	A, B	A, B	A	B

(a) An example task set



(b) Stack size reduction for the example task set

Figure 3. Example task set and stack size reduction.

Multiple activations mean that a basic task which was already activated can get more activation commands during its execution. OSEK OS dictates that such multiple commands be stored to reactivate a receiving task immediately after the termination of its current execution. By default, an extended task has multiple activations

OSEK OS uses a priority-based scheduling policy. The priority of each task is determined at design time by a programmer. In addition to priority, a task may has a preemption attribute. The kernel scheduler can preempt a task with a preemption attribute whenever a task with higher priority arrives at the system. On the other hand, if a running task does not have a preemption attribute, the scheduler can preempt it only after it releases the processor or terminates itself.

Application modes are used to group appropriate applications for execution based on conditions related to an ECU. Application programmers can define multiple application modes one of which is selected for execution during the boot-up process of the system. Once an application mode is selected, it cannot be changed until the system restarts. An application mode specifies a set of tasks and alarms that should be activated at the boot-up process. Such tasks and alarms can activate other tasks and alarms which are not automatically activated (Nelson *et al*., 2004). Figure 2 shows two application modes and their tasks. When the OSEK kernel starts with application mode 1, task A is automatically activated. After the boot-up process, tasks B and C are activated by task A. In this example, tasks A, B and C execute in application mode 1. In this paper, we use application modes to determine mutually exclusive task sets for stack sharing

2.2. Deriving Task Conditions for Stack Sharing

The OSEK OS standard, which does not allow dynamic task creation, mandates that a stack with a pre-determined size be statically allocated for each task. This policy leads to the wastage of run-time memory because a task occupies a stack space in memory regardless of its status. For instance, a task in a terminated state does not need any stack spaces. Zuberi *et al*. propose to recycle unused stack spaces in EMERALDS-OSEK (Zuberi *et al*., 2000). They further propose to share a stack among multiple tasks by observing two conditions that guarantee stack sharing among tasks. In principle, any two tasks can share the same stack if they never execute simultaneously.

In addition to the conditions revealed by Zuberi *et al*. (2000), we derive two additional conditions for further stack sharing. These four conditions altogether are listed in Table 1. The first three conditions group mutually exclusive tasks in a given application mode. The last indicates that any two tasks in two different application modes are mutually exclusive by definition. Condition C1 says that any two non-preemptive basic tasks are mutually exclusive since no other task can preempt a non-preemptive basic task since a basic task always runs to completion. Condition C2 holds true since accesses to an internal resource serialize the execution of accessing tasks. Condition C3 holds true since a preemptive basic task can be preempted only by higher priority tasks. While Zuberi *et al*. (2000) use only conditions C1 and

Table 1. Four conditions for stack sharing

- If tasks in a given application mode are
  C1. Non-preemptive basic tasks
  C2. Basic tasks sharing the same internal resource
  C3. Basic tasks with the same priority

- If tasks are
  C4. In distinct application modes

C3, our approach utilizes all of the four for extensive stack sharing.

We demonstrate the stack sharing mechanism using an example task set listed in Figure 3. Using the conditions in Table 1, we derive two mutually exclusive task sets $\{T_1, T_2, T_3\}$ and $\{T_4, T_5\}$, each of which shares a single task stack. Specifically, tasks T_1, T_2 and T_3 are mutually exclusive since (1) C1 is true for T_1 and T_3, (2) C2 for T_2 and T_3 and (3) C3 for T_1 and T_2. Tasks T_4 and T_5 are mutually exclusive since C4 is true for them. We can allocate a shared stack to each task set. A stack of 120 bytes is allocated for T_1, T_2 and T_3, and a stack of 180 bytes for T_4 and T_5. As a result, the total stack size is reduced from 600 bytes to 300 bytes, as shown in Figure 3.

3. LIGHT-WEIGHT READY QUEUE HANDLING

A ready queue is a kernel data structure that holds a list of tasks which are ready to execute whenever the CPU is available to them. OSEK OS mandates that the OSEK kernel should maintain ready tasks in priority order so that it can efficiently select a task for dispatching. In many OSEK OS implementations, a ready queue is constructed with a simple array for efficiency. Each element in the array corresponds to a priority level. Since OSEK OS allows multiple tasks to have the same priority in some configurations, an element becomes another array. This gives rise to a ready queue with a two-dimensional array. In other OSEK configurations, all tasks are forced to have distinct priorities. In this case, the two-dimensional array design of a ready queue leads to memory wastage.

In this section, we present a light-weight ready queue design specialized for various OSEK kernel configurations. In doing so, we take into consideration dynamic priority changes due to task synchronization.

3.1. Examining Conformance Classes and Task Synchronization

OSEK OS provides different OS profiles called conformance classes (CC) so as to prevent excessive resource usage by offering only minimally needed services for an application. Specifically, it defines two types of conformance classes CC1 and CC2. They differ from each other such that CC2 allows multiple tasks with the same priority and multiple activations for a basic task whereas CC1 does not. For both basic and extended tasks, there are four class combinations such as BCC1, BCC2, ECC1 and ECC2.

OSEK OS uses the well-known Priority Ceiling Protocol (PCP) for task synchronization (Goodenough *et al*., 1988, Locke *et al*., 1988). It deals with a race condition where multiple tasks try to use a shared resource simultaneously. The PCP helps avoid unbounded priority inversions and deadlocks. In the PCP, each shared resource is assigned its own priority and the OSEK OS specification provides the following rule for safe priority assignment.

- The resource priority shall be set at least to the highest priority of all tasks that access a resource or any of the resources linked to this resource. The resource priority shall be lower than the lowest priority of all tasks that do not access the resource, and which have priorities higher than the highest priority of all tasks that access the resource (OSEK Group, 2004, pp.31).

When a task starts using a shared resource, it temporarily inherits the resource priority if its priority is

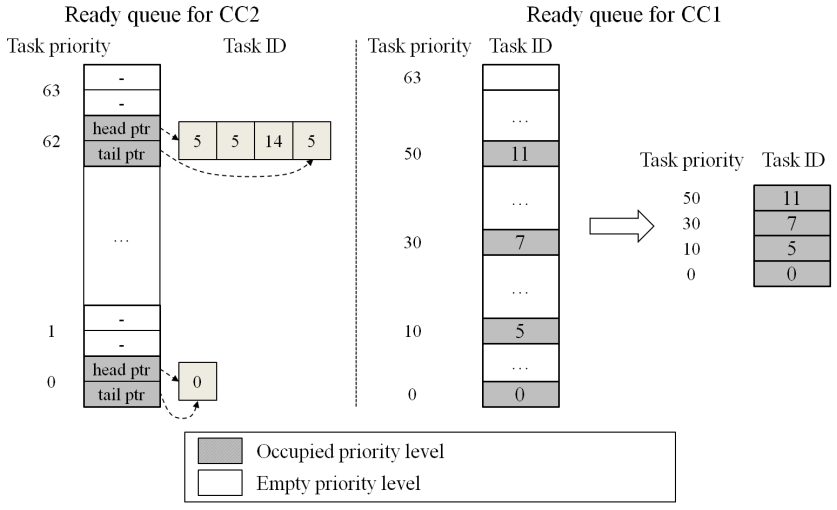


Figure 4. Ready queue design for CC1 and CC2.

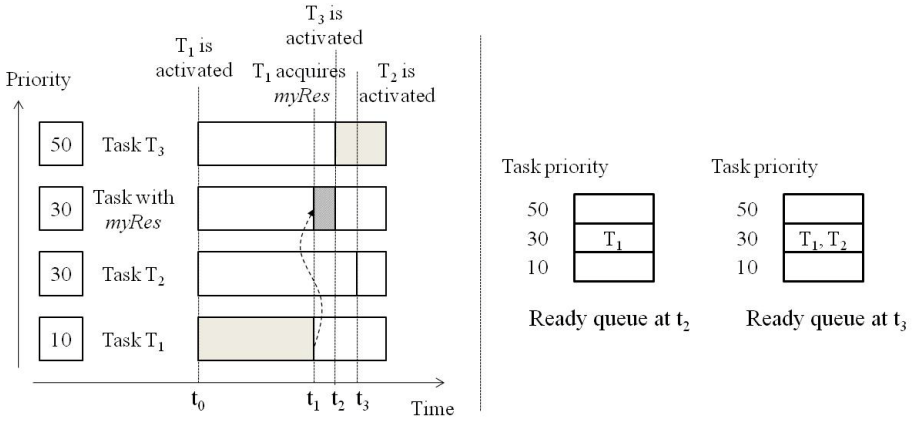


Figure 5. Problem caused by priority overlap.

lower than the resource's. Due to such dynamic priority inheritance of the PCP, any two distinct tasks may have the same priority even though the OSEK kernel is configured as CC1. This surely complicates the ready queue design for CC1.

3.2. Ready Queue Specialization and Non-Overlapping Priority Reassignment

We propose a ready queue design specialized for conformance classes. Conceptually, a prioritized ready queue needs to be implemented with a two-dimensional array to support tasks with the same priority as in CC2. On the other hand, a ready queue for CC1 can be simplified into a one-dimensional array since task and resource priorities do not overlap. Figure 4 depicts a general ready queue structure that works for CC2 and a light-weight ready queue structure for CC1. We can further shorten the one-dimensional array by removing priorities which are not used by any tasks or resources. Figure 4 also shows such optimization with a case where tasks use only four distinct priorities.

Unfortunately, such a straightforward design of the CC1 ready queue can run into a problem when the PCP is used for task synchronization. Figure 5 illustrates the case where there exists priority overlap between a task and a shared resource. When task T_1 acquires resource *myRes* at time t_1, its priority is promoted to 30. At t_2, task T_3 is activated and preempts task T_1. The scheduler stores T_1 into the ready queue at priority 30. At t_3, task T_2 is activated and immediately enters the ready queue at priority 30. However, the scheduler cannot store T_2 into the one-dimensional array ready queue since that priority is already occupied.

In order to rectify this problem, we propose a priority reassignment scheme that avoids priority overlap between tasks and resources. It attempts to reassign tasks and resources distinct priorities while maintaining their relative priority order. Figure 6 illustrates, with the same task set as Figure 5, how task priorities are reassigned. In this example, tasks T_1, T_2 and T_3 are reassigned priorities 0, 1 and 3, respectively. Resource *myRes* is reassigned priority 2. As a result, when T_1 acquires *myRes* at t_1, its priority becomes 3. At t_3, the scheduler can safely store T_2 into the ready queue.

Clearly, such a priority reassignment is not always

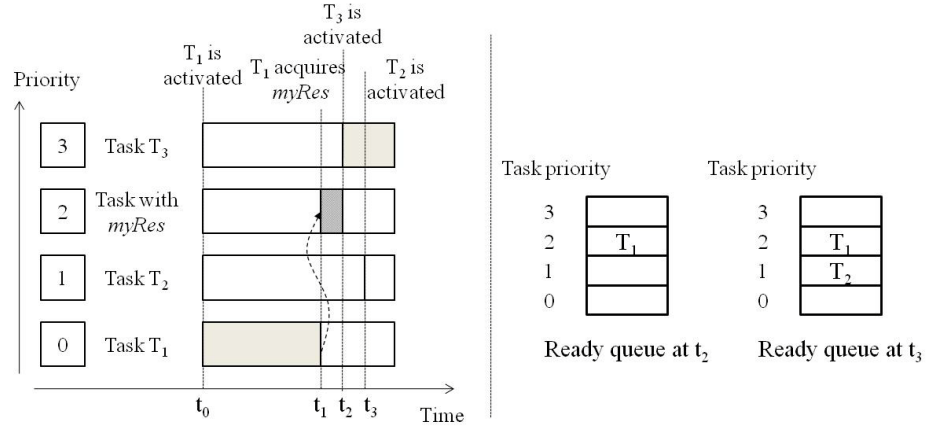


Figure 6. Priority reassignment scheme to avoid priority overlap.

```
TASK Task3 {
      TYPE = BCC1;
      SCHEDULE = NON;
      ACTIVATION = 1;
      AUTOSTART = FALSE;
      RESOURCE = Resource1;
      PRIORITY = 20;
      StackSize = 100;
      UsedMode = A, B;
      UsingSchedule = NO;
};

RESOURCE Resource1 {
      RESOURCEPROPERTY = STANDARD;
};
```

```
ALARM Alarm1 {
     COUNTER = SystemTimer;
     ACTION = ACTIVATETASK
     {
        TASK = Task3;
     };
     AUTOSTART = TRUE
     {
        AlarmUnit = Ticks;
        StaticAlarm = FALSE;
     };
};

EVENT Event1 {
     MASK = AUTO;
};
```

Figure 7. Example application configuration in an OIL file.

possible. Then, even a CC1 application needs to be configured to use the CC2 ready queue. Fortunately, such applications are rare in practice.

4. EXTENDING OIL FOR PROPOSED MECHANISMS

An OIL interpreter and code generator realize the stack sharing and light-weight ready queue handling mechanisms into an OSEK OS implementation after obtaining the kernel and application configuration information written in OIL. It is thus necessary to extend the original OIL with extra task attributes needed for the proposed mechanisms. In this section, we review the OIL and its accompanying tools. And then we identify task attributes that must be included in the extended OIL and show a revised code generation process via a code example.

4.1. Configuring Applications and Generating Code using OIL

OSEK OS offers the OIL as a configuration language so that developers can describe task attributes and declare kernel objects in their application. Since OSEK OS defines only a library kernel, application code is statically linked with the kernel code and built into a single executable image. Also, OSEK OS mandates that all kernel objects such as tasks, alarms, events and resources be statically created. Such static linking and kernel object creation help avoid dynamic memory allocation which increases the complexity of kernel design and decreases the predictability of task execution. Figure 7 depicts a code fragment of an application configuration written in OIL. Configurations of `Task3`, `Resource1`, `Alarm1` and `Event1` are described in the example. From the descriptions, we know that `Task3` is a non-preemptive basic task and its priority is 20. The task requires 100 bytes for its stack, executes in application modes A and B, and does not invoke the `Schedule()` API. In the subsection that follows, we explain the code generation process for the extended OIL.

The OIL comes with two tools, an OIL interpreter and a C code generator. They parse an OIL file and

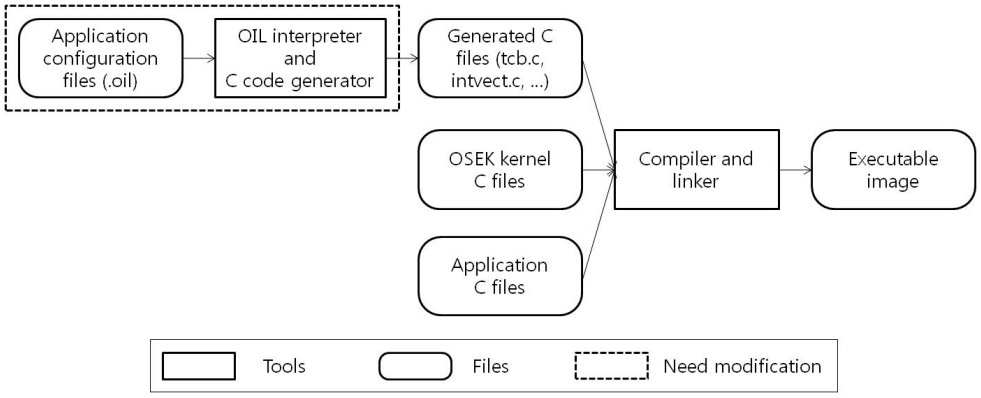


Figure 8. OSEK application development process with OIL.

```
uint8 stack0[120];
uint8 stack1[180];
uint8 taskStack[5] = {stack0, stack0,
       stack0, stack1, stack1};
```

Figure 9. Task stack declarations for example tasks of Figure 3.

generate C source code files as described in the OIL file. Figure 8 demonstrates the OSEK application development process in our implementation. File "intvect.c" contains an interrupt vector table and "tcb.h" and "tcb.c" together describe task attributes, alarms, events and resources. These files are dependent on our mechanisms since they contain kernel data structures such as task control blocks. They are also dependent on underlying hardware architecture since it describes an interrupt vector table and priorities of interrupts. Thus, it is necessary to redesign the OIL interpreter and C code generator. As shown in Figure 8, generated files are compiled with the OSEK kernel code and application code. They altogether become an executable image for the target hardware.

4.2. Extending OIL for Augmented Task Attributes

An OIL file describes the properties of tasks such as priority, preemptability, multiple activations, auto-start, resource usage, events and messages. We identify all task properties required for the proposed mechanisms and introduce new properties to the OIL if they are not originally supported.

First, we identify extra information required for the stack sharing mechanism. It includes application mode, existence of a call to `Schedule()` and task stack size. We respectively denote them by *UsedMode*, *UsingSchedule* and *StackSize* fields. Note that a non-preemptive task that shares an internal resource with others must not call `Schedule()`. The *UsingSchedule* field is false if `Schedule()` is not called in the task.

```
// in Schedule() API

uint32_t hID =
   getHighestPriorityTaskID();
if (TCB[hID].stackAlloc == FALSE) {
   TCB[hID].stackPointer =
      createContext(taskStack[hID], ...);
   TCB[hID].stackAlloc = TRUE;
}
// make this task run

restoreContext(TCB[hID].stackPointer[hID]);
```

Figure 10. Context initialization code included in the `schedule()` API.

```
#if defined (BCC1) || defined (ECC1)
typedef readyQ_t uint8;

#elif defined (BCC2) || defined (ECC2)
typedef struct readyQueueType {
   uint8 head;
   uint8 tail;
} readyQ_t;
#endif

readyQ_t readyQueue[numOfDifferentPriority
            +
            numOfDifferentCeilingPriority];
uint8 priorityQueue0[];
uint8 priorityQueue1[];
…
```

Figure 11. Specialized ready queue implementation for CC1 and CC2.

The *StackSize* field explicitly describes the stack size of a task.

Second, we derive additional information required for the light-weight ready queue design. In order to select an appropriate ready queue design to use for an application, developers need to know task types, multiple activations and the number of tasks with the same priority. They can obtain such information from an ordinary OIL file.

4.3. Code Generation using Extended OIL

We demonstrate how the OIL interpreter and C code generator work with task descriptions in an extended OIL file. In Figure 9, we show a code fragment generated for the stack sharing tasks example of Figure 3. Recall that tasks T_1, T_2 and T_3 can share a stack of 120 bytes and that tasks T_4 and T_5 can share a stack of 180 bytes. The code fragment contains the stack declaration of the tasks saying that T_1, T_2 and T_3 share `stack0` and T_4 and T_5 share `stack1`.

Figure 10 shows a code fragment initializing task context. Conventional OSEK OS implementations build the context of a task onto the task stack during the boot-up process of the system. Our OSEK OS implementation, however, cannot initialize the context of stack sharing tasks during the boot-up process since they use the same stack. Instead, the context of such tasks is initialized during context switching time. To do so, the scheduler must know whether the context of a task to run next is already initialized. The `stackAlloc` flag keeps such information. It is set to false when the task's state is changed from suspended to ready and it becomes true after context initialization.

Figure 11 gives an code example of our light-weight ready queue design. The ready queue for CC1 is a simple array of characters and that for CC2 is an array of structures. As explained earlier on, a ready queue

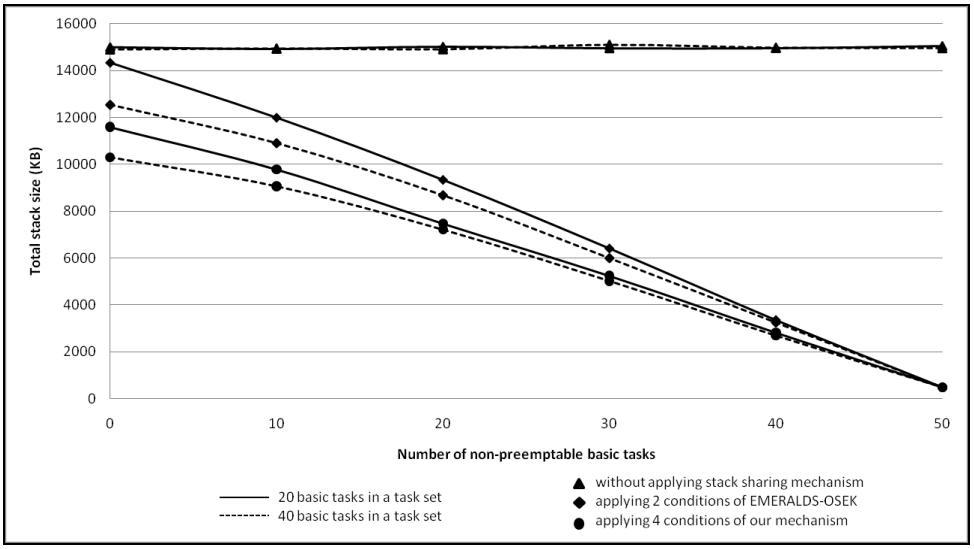


Figure 12. Total stack size reduction for different number of non-preemptable basic tasks.

holds as many elements as distinct task priorities and resources priorities altogether. In CC2, each element points to a circular queue that holds multiple tasks with the same priority. `priorityQueue` arrays in the example are used for this purpose.

5. EXPERIMENTAL EVALUATION

We have implemented the proposed mechanisms into an OSEK compliant operating system that we developed for an evaluation board equipped with an MPC5554 processor. We have extended OIL as explained in Section 5 and augmented the OIL interpreter and C code generator. For experiments, we made OIL files which described synthetically produced task sets and generated the source code of tasks using the OIL files. Each task set consisted of 50 randomly generated tasks. We then measured the total stack size demand of each task set by summing individual task stack sizes declared in the generated source code. The proposed mechanisms achieve memory footprint reduction by extending OIL and modifying the C code generator, while dynamic behavior of tasks remains unchanged. Thus there is no performance degradation for reducing memory footprint. For comparing our approach with others, we measured three different total stack size demands as follows.

- S_1: Total stack size when stack sharing is not applied.
- S_2: Total stack size when the two conditions proposed in (Zuberi *et al*. 2000) are applied.
- S_3: Total stack size when all the conditions proposed in this paper are applied.

Task attributes such as task type, preemptability, resource usage, priority, stack size and application mode affect total stack sizes. In our experiments, we used three of them as test variables: (1) the number of basic tasks in a task set, being either 20 or 40, (2) the number of non-preemptable tasks in a task set, ranging from 0 to 50, and (3) the number of basic tasks sharing an internal resource, ranging from 0 to 50. We did not choose the application mode as a test variable since it is straightforward to estimate the effect of condition C4 that involves the application mode. We used only one application mode in our experiments. Other task attributes were randomly selected. Each task's priority was a uniformly distributed random value in the range of [0, 63]. Each task stack size was also a uniformly distributed random value in the range of [100, 500] KB.

We have performed two kinds of experiments. First, we varied the number of non-preemptable tasks while the other two test variables were fixed. Second, we varied the number of tasks that shared an internal resource while the other two test variables were fixed. For each task set, we measured two total stack sizes, one with 20 basic tasks and the other with 40 basic tasks. We have performed measurements 100 times and averaged the total stack sizes.

Figure 12 depicts the total stack size as the number of non-preemptable tasks is varied. We set to 10 the number of basic tasks that shared an internal resource. The result shows that S_3 is only 40% of S_1 on average. S_3 is smaller than or equal to S_2 since the conditions in our mechanism subsume those proposed in (Zuberi *et al*. 2000).

Figure 13 depicts the total stack size of task sets as the number of tasks that shared an internal resource is varied. We set the number of non-preemptable tasks to 20. Since only condition C2 is related to internal resources, S_2 is not affected in this case. The result shows that S_3 is only 32% of S_1 on average. S_3 is also smaller than or equal to S_2 which is 59% of S_1.

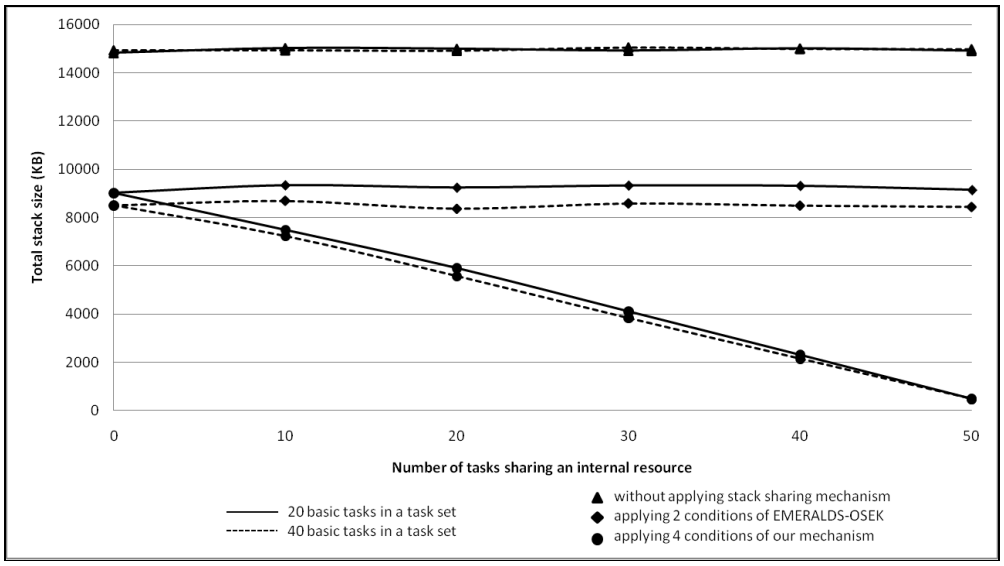


Figure 13. Total stack size reduction for different number of tasks sharing an internal resource.

6. RELATED WORK

The run-time memory of an executing program can be subdivided into three disjoint segments: code, data and stack segments. In the literature, most of attempts to reduce the run-time memory requirement of an OSEK OS implementation focus on code and stack segments since there are little room for optimizing a data segment for a given microcontroller. Zuberi *et al*. (2000), Chen, T. *et al*. (2005) and Chen, W. *et al*. (2005) address stack size reduction via sharing the same stack among multiple tasks. These approaches differ in task conditions for stack sharing. In principle, any two disjoint tasks can occupy the same stack if they never execute simultaneously. In (Chen, T. *et al*., 2005) and (Chen, W. *et al*., 2005), only non-preemptable basic tasks are allowed to share a stack. In (Zuberi *et al*., 2000), basic tasks at the same priority level are additionally included for stack sharing, based on the fact that a basic task runs to completion and can be preempted only by higher priority tasks. In our approach, we further include for stack sharing basic tasks accessing the same internal resource and tasks belonging to different application modes. This effectively increases the number of stack sharing tasks. In order to realize such extensive stack sharing, we add two extra fields in the OIL task descriptions such as an application mode in which a task runs and a flag denoting whether a task invokes a schedule API call. We also develop an OIL interpreter and C code generator that support the extended OIL.

In (Zuberi *et al*., 2000), (Chen, T. *et al*., 2005), (Chen, W. *et al*., 2005) and (Barthelmann, 2004), multiple stack emulation is proposed to reduce the per-task stack space. It is quite often the case in automotive applications that an OSEK OS implementation is hosted on a low-end microcontroller that has only a single stack pointer register. For the sake of simplicity and efficiency, legacy OSEK kernels support only per-task stacks and use them for interrupt handling and alarm handling as well as task execution. This leads to the per-task stack size increase since an extra space is reserved in each task stack for interrupt and alarm handling. To avoid this problem, Zuberi *et al*. (2000), Chen, T. *et al*. (2005) and Chen, W. *et al*. (2005) offer an additional interrupt stack separate from a task stack by emulating multiple stack pointers and stack switching. ProOSEK discussed in (Barthelmann, 2004) uses a separate kernel stack as well as an interrupt stack.

In (Barthelmann, 2004), inter-task register allocation is adopted for ProOSEK in that a compiler allocates a separate register group to each task. This helps reduce the stack size of a task because the number of registers that need to be saved in a stack during context switching is decreased. Also, it is important to precisely estimate the maximum stack size requirement of a task since such information is specified in an OIL file. The C code generator uses this information later to statically allocate a task stack. In (Gu *et al*., 2005) and (Barthelmann, 2004), a compiler is used to calculate the tight stack size bound of a task while other legacy OSEK kernels reply on manually calculated stack size information.

There are several attempts to reduce the memory space used by kernel data structures such as a ready queue. In (Zuberi *et al*., 2000), (Chen, T. *et al*., 2005) and (Chen, W. *et al*., 2005), a one-dimensional array, instead of a two-dimensional one, is used as a ready queue for conformance classes BCC1 and ECC1. In this

case, the length of the array is equal to the number of priorities supported by the kernel. Our approach uses the same ready queue for BCC1 and ECC1. However, it is different from (Zuberi *et al*., 2000), (Chen, T. *et al*., 2005) and (Chen, W. *et al*., 2005) in a sense that it further reduces the size of the ready queue by reassigning distinct priorities to resources and tasks so as to eliminate unused priorities and hence unused array elements.

7. CONCLUSION

In this paper, we proposed two kernel mechanisms for reducing the dynamic memory requirement of OSEK-based systems. They are stack sharing among tasks and light-weight ready queue handling specialized for OSEK OS conformance classes. Our stack sharing mechanism saves memory by exploiting conditions derived from the run-to-completion property of a basic task. The light-weight ready queue safely works due to the priority reassignment scheme that avoids the priority overlap problem. We also presented implementation methods for the proposed mechanisms by extending OIL and associated tools.

We have performed extensive experiments to measure the amount of memory footprint reduction. The results show that our approach cut the memory requirement by 36% on average, in comparison with conventional OSEK OS implementations. This result is achieved without incurring run-time performance degradation.

We are currently looking to extend the proposed mechanisms so that they can be applied to OSEKTime which is a kernel standard for a time-triggered real-time operating system (OSEK Group, 2001). Since OSEKtime has a different scheduling policy from OSEK OS, it may require additional stack sharing conditions.

ACKNOWLEDGEMENT–The work reported in this paper was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MEST) (No. 2010-0027809 and No. 2010-0001201).

REFERENCES

Barthelmann, V. (2004). Advanced Compiling Techniques to Reduce RAM Usage of Static Operating Systems, PhD thesis, Friedrich-Alexander University of Erlangen-Nuremberg, Erlangen.

Chen, T., Chen, W., Wang, X. and Hu, W. (2005). Implementing and Evaluation of an OSEK/VDX-Compliant Configurable Real-time Kernel, *IEEE Networking, Sensing and Control*, 555-559.

Chen, W., Wu, Z. and Wang, X. (2005). Minimizing Memory Utilization of Task Sets in SmartOSEK, *Advanced Information Networking and Applications* **2**, 552-558.

Goodenough, J. B. and Sha, L. (1988). The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks, *Proceedings of the 2nd International Workshop on Real-time Ada Issues*.

Gu, Y., Wu, Z. and Yue, L. (2005). AlphaOS, An Automotive RTOS Based on OSEK/VDX: Design and Test, *IEEE Networking, Sensing and Control*, 174-179.

John, D. (1998). OSEK/VDX History and Structure, *IEE Seminar 523 OSEK/VDX Open Systems in Automotive Networks*, 2/1-2/13.

Locke, C. D. *et al*. (1988). Priority Inversion and Its Control: An Experimental Investigation, *ACM SIGADA Ada Letters* **8**, **7**, 39-42.

Nelson, E. C., Prasad, K. V., Rasin, V. and Simonds, C. J. (2004). An Embedded Architectural Framework for Interaction Between Automobiles and Consumer Devices, *Proceedings of the 10th IEEE RTAS*.

OSEK/VDX. (2005). Operating System Specification 2.2.3, OSEK Group.

OSEK/VDX. (2004). System Generation, OIL: OSEK Implementation Language 2.5, OSEK Group.

OSEK/VDX. (2001). Time-Triggered Operating System Specification 1.0, OSEK Group.

Yoon, M., Lee, W. and Sunwoo, M. (2005). Development and Implementation of Distributed Hardware-in-the-loop Simulator for Automotive Engine Control Systems, *Int. J. Automotive Technology* **6**, **2**, 107-117.

Zuberi, K. M. *et al*. (2000). EMERALDS-OSEK: A Small Real-time Operating System for Automotive Control and Monitoring, *SAE transactions* **108**, **6**.

LIST OF FIGURES