A Programmable Microkernel for Real-Time Systems[*]

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Marco A.A. Sanvido
VMWare Inc.

Thomas A. Henzinger
EPFL and UC Berkeley
tah@epfl.ch

ABSTRACT

We present a new software system architecture for the implementation of hard real-time applications. The core of the system is a microkernel whose reactivity (interrupt handling as in synchronous reactive programs) and proactivity (task scheduling as in traditional RTOSs) are fully programmable. The microkernel, which we implemented on a StrongARM processor, consists of two interacting domain-specific virtual machines, a reactive E (Embedded) machine and a proactive S (Scheduling) machine. The microkernel code (or microcode) that runs on the microkernel is partitioned into E and S code. E code manages the interaction of the system with the physical environment: the execution of E code is triggered by environment interrupts, which signal external events such as the arrival of a message or sensor value, and it releases application tasks to the S machine. S code manages the interaction of the system with the processor: the execution of S code is triggered by hardware interrupts, which signal internal events such as the completion of a task or time slice, and it dispatches application tasks to the CPU, possibly preempting a running task. This partition of the system orthogonalizes the two main concerns of real-time implementations: E code refers to environment time and thus defines the reactivity of the system in a hardware- and scheduler-independent fashion; S code refers to CPU time and defines a system scheduler. If both time lines can be reconciled, then the code is called time safe; violations of time safety are handled again in a programmable way, by run-time exceptions. The separation of E from S code permits the independent programming, verification, optimization, composition, dynamic adaptation, and reuse of both reaction and scheduling mechanisms. Our measurements show that the system overhead is very acceptable even for large sets of task, generally in the 0.2–0.3% range.

[*]This research was done while all authors were at UC Berkeley and supported in part by the AFOSR MURI grant F49620-00-1-0327 and the NSF grants CCR-0208875 and CCR-0225610.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Languages

Keywords

Real Time, Operating System, Virtual Machine

1. INTRODUCTION

In [9], we advocated the E (Embedded) machine as a portable target for compiling hard real-time code, and introduced, in [11], the S (Scheduling) machine as a universal target for generating schedules according to arbitrary and possibly non-trivial strategies such as nonpreemptive and multiprocessor scheduling. In this paper, we show that the E machine together with the S machine, form a programmable, low-overhead microkernel for real-time systems. We implemented the microkernel on a StrongARM SA-1110 processor and measured the system overhead to lie in the 0.2–0.3% range. The implementation has a very small footprint, namely, 8kB.

The E machine is woken up by external interrupts caused by environment events, such as the arrival of a message on a channel, or the arrival of a new value at a sensor. Once awake, the E machine follows E code instructions to do three things: first, it may run some drivers for managing sensors, actuators, networking, and other devices; second, it may release some application software tasks for execution; third, it may update the trigger queue, which contains pairs of the form (e, a) indicating that the future environment event e will cause an interrupt that wakes up the E machine with its program counter set to the E code address a. Then, the E machine goes back to sleep and relinquishes control of the CPU to the S machine. The S machine is woken up by the E machine, or by internal interrupts caused by processor events, such as the completion of an application task, or the expiration of a time slice. Once awake, the S machine follows S code instructions to do three things: first, it takes care of processor and memory management, such as context switching; second, it dispatches a single task to the CPU, which may be either an application software task or a special idle task; third, it specifies pairs of the form (i, a) indicating that the future processor event i will cause an interrupt that wakes up the S machine with its program counter set to the S code address a.

The E and S architecture partitions the microkernel code, or simply *microcode*, into two categories: E code supervises the "logical" execution of application tasks relative to environment events; S code supervises the "physical" execution of application tasks on the given resources. At any time, E code may release several tasks, but if there is only a single CPU, then S code can dispatch at most one released task at a time. In other words, E code specifies the reactivity of an embedded system independent of the hardware resources and the task scheduler, and S code implements a particular scheduler. The scheduler implemented in S code is fully programmable; it may be static or dynamic, preemptive or nonpreemptive. Together, the E and S machines form a programmable microkernel for the execution of hard real-time tasks.

There are several benefits this architecture offers over traditional real-time operating systems.

Real-time predictability of the application behavior. Since E code specifies the reactivity and timing of the system independent of the hardware and scheduler, a change in hardware or scheduler does not affect the real-time behavior of the application. This is especially important in control applications, where both slow-downs and speed-ups may lead to instabilities. By contrast, in a traditional RTOS, the real-time behavior of an application depends on the scheduling scheme, the processor performance, and the system load, which makes both code validation and reuse very difficult. In our setting, timing predictability depends, of course, on the fact that the S code meets the timing requirements specified by the E code. Given worst-case execution times of the application code, this can be checked either statically, by scheduling analysis [10], or at run-time. In the latter case, E code run-time exceptions may be used to handle so-called "time-safety" violations (i.e., missed deadlines) in an explicit, programmable way [9].

Composability of real-time applications. Suppose we want to put two software components, each consisting of E, S, and application code, on the same hardware. In a traditional RTOS, the combined real-time behavior of both components may differ significantly from their individual behaviors if run in isolation. Not so in our setting, where the E code of both components is always composable, because its execution is synchronous [6], that is, E instructions are executed in "logical zero time" with interrupts turned off. However, the S code of both components constitutes two threads that may or may not be composable: if there is a single CPU, then both S threads are composable provided whenever one thread dispatches an application task, the other thread dispatches the idle task. Composability can, again, be checked either statically or at run-time. In the latter case, a so-called "time-share" violation occurs whenever two S threads attempt to simultaneously control the CPU. Now, S code (rather than E code) run-time exceptions may be used to handle these situations in an explicit, programmable way. This concept can be generalized to multiprocessor hardware, which executes several threads of S code in parallel.

Dynamic adaptation of real-time code. As both E and S code are interpreted, we can dynamically optimize, patch, and upgrade them. A dynamic change in E code modifies the reactivity of a system and can be used, for example, to switch between different modes or contingencies of an embedded controller [9]. Similarly, a dynamic change in S code can be used to optimize or adapt the scheduling scheme without bringing down the system. This permits, in particular, adjustments in the event of hardware failures and thus provides a basis for achieving fault-tolerance without compromising the platform-independent reactivity specification —the E code— of a real-time application.

The real-time microkernel architecture we introduce here provides programmable timing and scheduling services. In this sense, our work relates to other work on microkernels, which typically provide basic thread and interprocess communication services [16, 1, 19, 2]. System services implemented on top of a microkernel have already been shown to perform well, e.g., in [7]. We demonstrate here that high performance is also possible using an even more flexible, programmable microkernel.

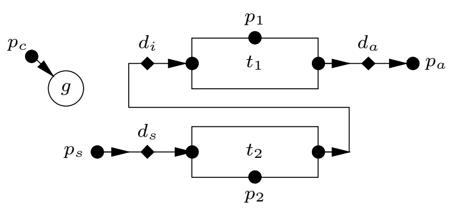
The paper is organized as follows. In Section 2, we briefly review the E machine. In Section 3, we introduce the complementary S machine, in Section 4, we define the combined E and S machines, and in Section 5, we present some of the benefits of this way of architecting a real-time kernel. Section 6 describes our implementation of the programmable microkernel on the StrongARM, and Section 7 evaluates our overhead measurements for many different scenarios. The final Section 8 presents related works.

2. THE EMBEDDED MACHINE

This section is a summary of the E Machine presented in [9]. The E machine is a mediator between physical processes and application software processes: it executes E code, which is system-level machine code that supervises the execution of software processes in relation to physical events. The E machine has two input and two output interfaces. The physical processes communicate information to the E machine through *environment ports*, such as clocks and sensors. The application software processes, called *tasks*, communicate information through *task ports* to the E machine. The E machine communicates information to the physical processes and to the tasks by calling system processes, called *drivers*, which write to *driver ports*. The E machine releases tasks to an external task scheduler for execution by writing to *release ports*. Logically, the E machine does not need to distinguish between environment and task ports; they are both input ports, while driver and release ports are output ports of the machine.

A change of value at an input port, say, a sensor port p_s, is called an *input event*. Every input event causes an interrupt that is observed by the E machine and may initiate the execution of E code. Such an *event interrupt* can be characterized by a predicate called a *trigger*. For example, $p'_s > p_s$ is a trigger on p_s, where p'_s refers to the current sensor reading, and p_s refers to a threshold. In general, a trigger may observe environment and task ports, which we call its *trigger ports*. Once a trigger is *activated* it is logically evaluated with every input event; an active trigger is *enabled* if it evaluates to true. A *time trigger* is a trigger on an environment (clock) port p_c with a predicate $p'_c = p_c + \delta$ where p_c is the clock port value at the time of the trigger activation, and δ is the number of ticks to wait before the trigger is enabled.

Tasks, drivers, and triggers are functional code that is external to the E machine and must be implemented in some programming language like C. Tasks, drivers, and triggers are given as binary executables to which E code refers through symbolic references. The execution of drivers and



```
mode m() period 20 {
  actfreq 1 do p_a(d_a);
  taskfreq 1 do t_1(d_i);
  taskfreq 2 do t_2(d_s); }
```

Figure 1: A simplified controller

tasks is supervised by E code, which monitors input events through triggers. There are three unique E code instructions. In an actual implementation of the E machine, E code also has standard control-flow instructions such as conditional and absolute jumps. A `call`(d) instruction initiates the execution of a driver d. A driver may provide sensor readings as arguments to a task, or may load task results into actuators, or may provide task results as arguments to other tasks. A driver may read from any port but writes only to driver ports. In favor of a simpler presentation, we assume that a driver has a fixed set of ports on which it operates. A driver executes in logical zero time, i.e., before the next input event can be observed. As the implementation of d is system-level code, the E machine waits until d is finished before interpreting the next instruction of E code. A `release`(t) instruction[1] releases a task t to run concurrently with other released tasks by emitting a signal to an external task scheduler on the release port of t. Then the E machine proceeds to the next instruction. The task t does not execute before the E machine relinquishes control of the processor to the scheduler. A task is a piece of preemptive, user-level code, which typically implements a computation activity. A task has no internal synchronization points. A task reads from driver ports and computes on task ports. The set of ports on which a task operates is fixed. Each task has a special task port, called *completion port*, which indicates that the task completed execution. The `release` instruction itself does not order the execution of tasks. If the E machine runs on top of an operating system, the task scheduler may be implemented by the scheduler of the operating system [9]. An alternative implementation of a task scheduler is the S machine of Section 3. The task scheduler is not under control of the E machine; like the physical environment and the underlying hardware, it is external to the E machine and may or may not be able to satisfy the real-time assumptions of E code. A `future`(g, a) instruction marks the E code at the address a for execution at some future time instant when the trigger g becomes enabled. In order to handle multiple active triggers, a `future` instruction puts the trigger-address pair into a *trigger queue*. With each input event, all triggers in the queue are evaluated in logical zero time. The first pair whose trigger is enabled determines the next actions of the E machine.

[1]The `release` instruction corresponds to the `schedule` instruction in [9] but has been renamed here for clarity.

```
a_0: call(d_a)        a_1: call(d_s)
     call(d_s)             release(t_2)
     call(d_i)             future(g, a_1)
     release(t_1)
     release(t_2)
     future(g, a_1)
```

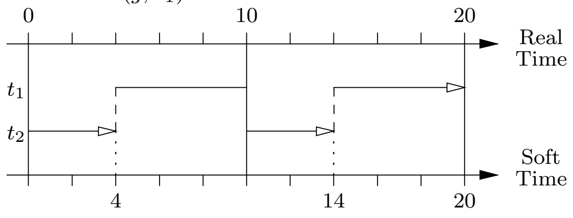


Figure 2: E code for the simplified flight controller and an execution trace of the E code using an EDF scheduler

E Code Example. Fig. 1 shows the topology of the program and a high-level Giotto [8] description of the program timing: we denote ports by bullets, tasks by rectangles, drivers by diamonds, and triggers by circles. Consider the mode m. There are two tasks, both implemented in native code: task t_1, and task t_2. Task t_2 processes input every 10 ms and provides the processed data to task t_1. Task t_1 processes the date, and writes the result to actuators. Task t_1 is executed every 20 ms. The release port p_1 of t_1 indicates whether t_1 has been released to run. Similarly, p_2 is the release port of t_2. The data communication requires three drivers: a sensor driver d_s, which provides the data to task t_1; a connection driver d_i, which provides the result of task t_1 to task t_2; and an actuator driver d_a, which loads the result of task t_2 into the actuator. The drivers may process the data in simple ways (such as type conversion), as long as their WCETs are negligible. There are two environment ports, namely, a clock p_c and the sensor p_s; two task ports, one for the result of each task; and three driver ports —the destinations of the three drivers— including the actuator p_a. The "`actfreq` 1" statement in the Giotto program causes the actuator to be updated once every 20 ms; the "`taskfreq` 2" statement causes the navigation task to be invoked twice every 20 ms; etc. The E code generated by the Giotto compiler [10] is shown in Fig. 2.

The E code consists of two blocks. The block at address a_0 is executed at the beginning of a period, say, at 0 ms: it calls the three drivers, which provide data for the tasks and the actuator, then releases the two tasks to the task scheduler, and finally activates a trigger g with address a_1. When the block finishes, the trigger queue of the E machine contains the trigger g bound to address a_1, and the release ports of the two tasks, t_1 and t_2, are set to ready. Now the E machine relinquishes control, only to wake up with the next input event that causes the trigger g to evaluate to true. In the meantime, the task scheduler takes over and assigns CPU time to the released tasks according to some scheduling scheme. Fig. 2 shows an execution trace of the E code using an earliest deadline first (EDF) scheduler, which gives priority to tasks with earlier deadlines. The deadlines of the tasks are given as *E code annotations* [9] in the `release` instructions (not shown here).

There are two kinds of input events, one for each environment port: clock ticks, and changes in the value of the sensor p_s. The trigger g: $p_c' = p_c + 10$ specifies that the

E code at address a_1 will be executed after 10 clock ticks. Logically, the E machine wakes up at every input event to evaluate the trigger, finds it to be false, until at 10 ms, the trigger is true. An efficient implementation, of course, wakes up the E machine only when necessary, in this case at 10 ms. The trigger g is now removed from the trigger queue, and the associated a_1 block is executed. It calls the sensor driver, which updates a port read by task t_2. There are two possible scenarios: the earlier invocation of task t_2 may already have completed with a signal on the completion port of t_2. In this case, the E code proceeds to release t_2 again, and to trigger the a_0 block in another 10 ms, at 20 ms. In this way, the entire process repeats every 20 ms. The other scenario at 10 ms has the earlier invocation of task t_2 still incomplete, i.e., the completion port of t_2 has not yet signaled completion. In this case, the attempt by the sensor driver to overwrite a port read by t_2 causes a run-time exception, called *time-safety violation.* At 20 ms, when ports read by both tasks t_1 and t_2 are updated, and ports written by both t_1 and t_2 are read, a time-safety violation occurs unless both tasks have completed. In other words, an execution of the program is time-safe if the scheduler ensures the following: (1) each invocation of task t_1 at $20n$ ms, for $n \geq 0$, completes by $20n + 20$ ms; (2) each invocation of task t_2 at $20n$ ms completes by $20n + 10$ ms; and (3) each invocation of task t_2 at $20n + 10$ ms completes by $20n + 20$ ms. Therefore, a necessary requirement for schedulability is $\delta_1 + 2\delta_2 < 20$, where δ_1 is the WCET of task t_1, and δ_2 is the WCET of t_2. If this requirement is satisfied, then a scheduler that gives priority to t_2 over t_1 guarantees schedulability.

The E code implements the Giotto program correctly only if it is time-safe: during a time-safe execution, the navigation task is executed every 10 ms, the control task every 20 ms, and the dataflow follows Fig. 1. A schedulable E code program will generate only time-safe executions, and thus the Giotto compiler needs to ensure schedulability when producing E code. In order to ensure this, the compiler needs to know the WCETs of all tasks and drivers (cf., for example, [5]), as well as the scheduling scheme used by the task scheduler. With this information, schedulability for E code produced from Giotto can be checked. However, for arbitrary E code and platforms, such a check is difficult [10], and the programmer may have to rely on run-time exception handling.

3. THE SCHEDULING MACHINE

The S machine [11] is a virtual machine that determines the temporal order of task execution: it interprets S code, which is system-level machine code that dispatches tasks or idles. In the following, we give an overview of the S machine concepts. The S machine reads from three input interfaces to determine a running task. The release of tasks is communicated to the S machine through *release ports.* The tasks communicate information including their completion to the S machine through *task ports.* An external clock writes to a *clock port* that is read by the S machine to time-slice tasks.

The S machine uses *timeouts* to monitor input events. A timeout is similar to a trigger: it is a predicate over the input ports of the S machine. In particular, we are interested in timeouts of the form $p'_c \geq p_c + \delta$ where p_c is the clock port of the S machine. For readability of the code examples, we abbreviate timeouts to the δ value, e.g., 10ms denotes a timeout $p'_c \geq p_c + 10$ms. A timeout *expires* if it evaluates to

```
a0: dispatch(t2)
    dispatch(t1)
    idle(release(t2))
    dispatch(t2)
    idle(release(t1))
    fork(a0)
```

```
a0: dispatch(t2)
    dispatch(t1, release(t2), a1)
    idle(release(t2))
a1: dispatch(t2)
    dispatch(t1)
    idle(release(t1))
    fork(a0)
```

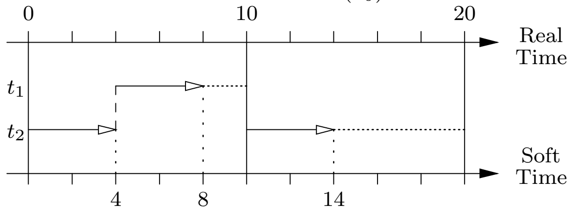


Figure 3: Synchronous S code and an execution trace of the S code, and preemptive S code

true. We also consider timeouts of the form *release*(t) where t is a task: *release*(t) expires if t is released and has not yet completed.

There are three unique S code instructions. Again, in an actual implementation of the S machine, S code also has standard control-flow instructions such as conditional and absolute jumps. A `dispatch`(t, m, a) instruction resumes the execution of a released task t until the timeout m expires. There are three outcomes: (1) the S machine proceeds to the next instruction if t has already completed but has not yet been released again, or else, (2) the S machine proceeds to the next instruction when t completes provided t completes before the timeout expires, or else (3) the S machine proceeds to the instruction at the address a when the timeout expires before t completes. An `idle`(m) instruction makes the S machine idle until the timeout m expires even though there may be released tasks. The S machine proceeds to the next instruction when the timeout expires. A `fork`(a) instruction marks the S code at the address a for execution in parallel to the S code that follows the instruction. The S code at a is a new *S thread* of execution. The running *thread instances* are kept in a *thread set* from which instances are chosen non-deterministically to execute. If multiple threads dispatch more than a single task at any instant the S machine throws a run-time exception, called *time-share violation.* We use `dispatch`(t) to abbreviate `dispatch`(t, false, a) as well as `dispatch`(t, m) to abbreviate `dispatch`(t, m, a), where a is the address of the next instruction.

S Code Examples. We present several S code examples that schedule the tasks t_1 and t_2 of Section 2 in different ways. Recall that t_1 is released once every 20ms while t_2 is released once every 10ms. Fig. 3 shows S code that dispatches t_1 and t_2 as follows: the S machine starts executing the S code at the address a_0 after both tasks have been released for the first time. t_2 is dispatched first. When t_2 completes, t_1 is dispatched. When t_1 completes, the S machine idles until t_2 is released again. Then t_2 is dispatched again until t_2 completes at which point the S machine idles until t_1 is released again. Then the S machine forks back to the S code at a_0. Since there is no S code following the `fork`(a_0) instruction, the current thread is terminated. Even in this case a fork is different than a jump to a_0 because, upon forking, the new thread instance is assigned the cur-

```
a0: dispatch(t2)                       a1: dispatch(t1)
    dispatch(t1, release(t2), a1)          dispatch(t2)
    idle(release(t2))                      idle(release(t1))
    dispatch(t2)                           fork(a0)
    idle(release(t1))
    fork(a0)
```

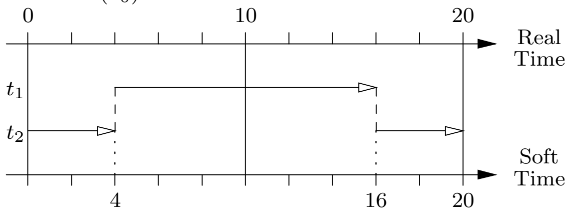


Figure 4: Non-preemptive S code and an execution trace of the S code

rent clock value as its *reference time* for timeouts. This will be explained in more detail below.

Fig. 3 shows an execution trace of the S code. The S code guarantees the time-safe execution of the E code in Fig. 2 if both tasks t_1 and t_2 complete within 10ms, i.e., if the tasks are never preempted by the release of a task. We call S code *synchronous* if, in any execution of the S code, all released tasks always complete before another task is released. In contrast, the S code in the right column of Fig. 3 allows the task t_1 to be preempted by the release of t_2, e.g., the E machine, and then the execution of t_2 itself. At startup, t_2 executes until completion. Then the dispatch(t_1, *release*(t_2)) instruction executes t_1 until either t_1 completes or t_2 is released again. If t_2 is released before t_1 completes, t_1 is preempted and the S machine continues executing the S code at the address a_1. Here, t_2 is dispatched until completion before t_1 resumes its execution. If, however, t_1 would have completed before t_2 was released, the dispatch(t_1) instruction following the instruction at a_1 would have no effect on t_1. The execution trace of the S code corresponds to the execution trace of the E code using an EDF scheduler as shown in Fig. 2. Thus the S code describes an EDF schedule for the E code. We call S code *preemptive* if, in any execution of the S code, tasks may be preempted by (1) the release and (2) the execution of other tasks.

The S code in Fig. 4 again allows the task t_1 to be preempted by the release of t_2 but then resumes the execution of t_1 instead of executing t_2. If t_1 does not complete before t_2 is released again, t_1 is dispatched resuming its execution before t_2 is dispatched. Fig. 4 shows an execution trace of the S code. We call S code *non-preemptive* if, in any execution of the S code, tasks are at most preempted by the release but not by the execution of other tasks. Synchronous S code is non-preemptive but not vice versa.

In Section 7, we will show experimental evidence that executing E and S code of different classes occurs at different administrative overhead because of scheduling and context switching. It turns out that synchronous S code causes less overhead than the other classes because there is neither dynamic scheduling nor context switching required. However, the drawback of synchronous S code is that (1) tasks have to compute faster than the basic unit of time, i.e., at least as fast as the most-frequent system activity, and (2) system utilization may be poor. At the other end of the spectrum, there is E code using a task scheduler instead of S code. Depending on the scheduler, tasks may be preempted at any time and system utilization may reach 100%, at the cost of scheduling and context switching overhead. Preemptive S code can reduce the overhead because fewer scheduling decisions are made at run-time. However, context switching is still necessary. Non-preemptive S code can reduce the overhead even further because context switching is only necessary between system and tasks but not between tasks. Generating non-preemptive S code, on the other hand, is an NP-hard problem [12, 3] and can thus only be approximated.

4. INTERACTING E AND S MACHINES

We discuss an implementation of interacting E and S machines. Instead of implementing both machines side-by-side, we propose an implementation that integrates the E and S machine into a single machine, which constitutes the core of the programmable microkernel (Alg. 1). An important aspect of an integrated implementation is to determinize correctly the logical order in which the E and S machine are invoked. For example, logically, the E machine should be invoked before the S machine when an E code trigger is enabled at the same time instant when an S code timeout expires because the E code may release tasks that require immediate scheduling service from the S code. An integrated E and S machine interprets *microcode* that may consist of E code and S code instructions.

We need the following data structures for the integrated implementation. A *system program* consists of ports, drivers, tasks, triggers, and timeouts, as well as microcode. A *configuration* of a system program consists of a system program, a trigger queue, a thread set, a reference time, a running task, and a microcode program counter. Since there are instants at which both the E machine and the S machine must be invoked, event interrupts are evaluated in the following, deterministic order: (1) task completion in the thread instances first, then (2) enabled triggers in the trigger bindings, and finally (3) expired timeouts in the thread instances. There are two motivations for this particular order. (1) before (2): task completion may require special handling, e.g., through driver calls in microcode, prior to executing any other microcode; and (2) before (3): enabled triggers may invoke microcode that releases tasks, which require scheduling service from microcode. Note that the integrated E and S machine may also use a task scheduler to schedule released tasks if there are no thread instances in the thread set in order to support the execution of E code without S code.

The integrated E and S machine uses a microcode interpreter (Alg. 2), which can execute any E and S code. The interpreter implements a straightforward merge of the E and S code interpreters. As a consequence, microcode is in fact more general than interacting E and S code. For example, a thread written in microcode, as opposed to S code, may call drivers, which may actually be useful in practice. However, we have presented E and S code separately because both types of code address equally important but orthogonal aspects of real-time systems. Microcode is an efficient representation of interacting E and S code but generating microcode may still benefit from keeping the logical difference of E and S code in mind.

5. CAPABILITIES

Semantic structure and predictability are the key properties of microcode that form the foundation of the microkernel's capabilities. E and S code address semantically orthogonal issues. E code defines the reactivity of the system with respect to the physical environment while S code defines application task scheduling. Microcode is dynamic in the sense that it can be replaced, modified, extended, and communicated at run-time. Modifying an E code portion of microcode changes the reactive behavior of the system while modifying the S code part changes the scheduling scheme. The semantic structure of microcode enables the analysis and composition of real-time programs on the system level. Microcode does not necessarily replace traditional real-time scheduling technology. Partial microcode can be complemented at run-time, e.g., by a real-time scheduler that either executes application tasks not handled by microcode, or generates the missing microcode on-the-fly. Predictability and composability of microcode enables portability and mobility of real-time programs.

Analyzing Microcode. Microcode is amenable to program and schedulability analysis. There are at least two interesting problems that involve checking schedulability: (1) is some given E code schedulable, i.e., do all tasks released by the E code are schedulable and complete on time, with respect to a given scheduling strategy and WCETs; and (2) does some given S code guarantee the time-safe execution of some given E code and does the S code follow a given scheduling strategy? For general E code with conditional branching, the first problem is difficult but becomes easier if the E code has a particular structure, e.g., is generated from Giotto or simply describes a set of periodic tasks [10]. In this case, the second problem can be solved fast even for non-trivial scheduling schemes such as non-preemptive scheduling [11]. Thus schedulability of E code combined with S code can be verified by the microkernel at run-time, e.g., as part of the integrity check in the idle task. Another interesting problem is to improve schedulability checking using control-sensitive program analysis techniques. Note that a control-insensitive check is a conservative approximation, which may fail on a schedulable program because the check considers program paths that are actually never taken.

Composing Microcode. An important feature of microcode is its composability. E code may be composed with other E code at compile time, or even at run-time through the trigger queue of the E machine. Logically, the reactive behavior of E code does not change when composed with other E code since E code execution is instantaneous. However, operationally, the instantaneousness of E code execution degrades with the number of E code instructions executed at the same instant. S code may also be composed with other S code at compile time, or at run-time through the thread set. In general, composing E code that uses S code for task scheduling requires regenerating the S code from scratch unless the S code was generated according to a compositional scheduling strategy. For example, if S code components are assigned exclusive time slots in which tasks are dispatched, then the composed S code is time-sharing, i.e., dispatches at most a single task at the same time, provided each S code component is already time-sharing. Thus S code can be used to study and utilize compositional scheduling strategies. The time-triggered architecture [14] offers a similar but hardware-based mechanism to time-share a communication bus that connects a distributed system of computers.

Partial Microcode. If the microkernel has a default task scheduler, then it is not necessary that microcode describes all behaviors of a real-time program. In fact, the microkernel can generate missing microcode at run-time. For example, S code may only dispatch a subset of all tasks. The task scheduler of the microkernel can then either dispatch the rest of the tasks whenever the S code execution completed, or else generate additional S code that dispatches the remaining tasks. Once the additional S code has been generated, it can execute repeatedly without the need for the task scheduler. Besides improved run-time performance, a benefit at design time is that prototypes of microcode can be developed gradually and executed before the code is complete. We have already taken advantage of this feature in the development and testing of the microkernel and the Giotto compiler.

Microcode may also be optimized at run-time based on information only available at run-time. For example, in the spirit of dynamic code optimization at run-time [13], the microkernel can reduce the number of task preemptions by rearranging S code instructions. We have used this technique for our benchmarks to obtain non-preemptive S code from preemptive S code.

Portable and Mobile Microcode. Portability and mobility of real-time programs are truly as challenging as they are desirable. Here are two examples: embedded systems such as control computers for satellites or power plants, which cannot easily be rebooted, would benefit from portable and mobile real-time code; or the performance of communication devices such as cell phones or network routers could be software-calibrated remotely while speaking or downloading. Predictability and composability of microcode enable portability and mobility. For example, *environment-triggered* microcode [9] whose triggers only refer to events such as the system clock tick or external signals is portable code as long as schedulability can be guaranteed. It is also mobile code because microcode is represented hardware-independently as byte code with symbolic references to functional code.

6. MICROKERNEL IMPLEMENTATION

In this section, we discuss the implementation of the programmable microkernel on a StrongARM SA-1110 processor running at 206MHz. We use a motherboard that was designed originally at ETH Zürich as part of a model helicopter project and is now available from *weControl*, an ETH spin-off company. The implementation is a patch of the custom-designed real-time operating system HelyOS [18] written in Oberon [20]. We have implemented a number of optimizations that exploit features of the processor and the compiler. We discuss the architecture-dependent aspects of the implementation at the end of this section.

Architecture-Independent Implementation

We use the following data structures in the architecture-independent part of the microkernel, which integrates the E and S machine and the microcode interpreter (Alg. 1 and 2). The *system state* consists of two parts: (1) a system program and its configuration; and (2) a *kernel state*, which consists of a *preempted task*, a *processor context*, and a set of *task instances* called *task set*. The running task is preempted when an event interrupt occurs. The processor context is a set of variables that contain the values of all registers of the processor. Typically, the stack and frame pointers as well as the processor status are stored in reserved registers. A task instance consists of a task and a processor context. For efficiency, all sets and queues are implemented by fixed-size arrays. At system startup, a bootstrap program (Alg. 3) initializes the microkernel. To communicate with the microkernel, we use a HelyOS I/O handler bound to an input and an output interrupt that operates on two cyclic buffers: (1) an input buffer that is read by the microkernel and written by the I/O handler when data is received from the host computer (on a serial link); and (2) an output buffer that is written by the microkernel and read by the I/O handler, which sends the content to the host computer. The event interrupt handler of the microkernel (Alg. 5) is bound to the system clock interrupt. When all interrupts are enabled, the special idle task (Alg. 4) is invoked. The idle task only returns when a shutdown command from the host computer was received. Note that during system operation the I/O interrupts remain enabled even when the microkernel is running.

The idle task (Alg. 4) checks in a while loop whether commands from the host computer were received and sends logging information generated by the microkernel and the tasks to the host computer. The idle task may receive microcode from the host, and execute it if it passes an integrity check of opcodes and arguments. The idle switches the system program executed by the microkernel to the received microcode as soon as a *safe* instant is reached. Here, a safe instant is any instant when all tasks have completed. Other, less trivial choices are possible.

Now, suppose that the idle task is running and an event interrupt (system clock tick) occurs. The event interrupt handler (Alg. 5) is invoked, which immediately disables the event interrupts and then saves the registers in the processor context variables. Then the running task is saved as the preempted task before the event loop of the integrated machine checks for microcode to be executed and determines the next running task. If the preempted task is again chosen to be the next running task, the registers are restored from the processor context variables and the handler returns from the interrupt. For this (often frequent) case, we demonstrate below in an architecture-dependent way that saving and restoring the processor context can entirely be avoided.

Architecture-Dependent Optimization

We describe an architecture-dependent optimization that reduces the number of context switches when the microkernel is invoked by an event interrupt. The StrongARM SA-1110 has 16 registers R0–R15 of which R0-R11 are general purpose registers and R12–R15 are reserved for system-specific use such as the stack and frame pointers. A context switch on the StrongARM requires to save all 16 registers to memory and then restore the registers from memory. The processor can operate in six different modes. In our implementation we use three of the six modes: (1) the HelyOS I/O handler runs in the IRQ mode; (2) the microkernel runs in the fast interrupt mode (FIQ); and (3) the tasks (including the idle task) run in the supervisor mode (SVC). The important difference among these modes is that the FIQ mode, unlike the IRQ and SVC modes, has a private set of registers R8–R15 that cover the registers R8–R15 of the other modes when the processor is in the FIQ mode. Thus a context

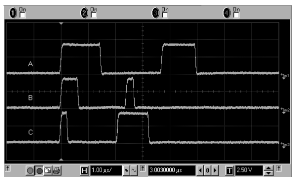


Figure 5: Context Switching on an Oscilloscope

switch in the FIQ mode can be avoided if the FIQ handler, i.e., the microkernel, only uses the registers R8–R15 and if the FIQ handler decides that, upon leaving the FIQ mode, the processor should resume the execution from where it was preempted.

In order to implement the optimization four modifications of our code were required: (1) we modified the Oberon compiler to support a procedure annotation that restricts the choice of registers the compiler can use to compile an annotated procedure; then, (2) we annotated the microkernel and compiled it to machine code that only uses the R8–R15 registers. Most importantly, drivers called by the microkernel are excluded from this restriction because (3) we modified the microcode interpreter to save all 16 registers before a driver is called. This can easily be generalized such that the interpreter distinguishes restricted from unrestricted drivers. Finally, (4) we changed the event interrupt handler (Alg. 5) as follows: we removed the code at the beginning and the end of the handler that saves and restores the registers in the processor context. Then, we inserted the code that saves the registers right before the processor context is needed to update the preempted task instance in the task set. Finally, we inserted the code that restores the registers right after the next running task instance is retrieved from the task set. Now, when an event interrupts occurs, the processor enters the FIQ mode and invokes the event interrupt handler, which first disables the event interrupts. Then, the handler immediately saves the running task as the preempted task and invokes the event loop of the integrated machine without saving any of the registers in the processor context. When the event loop is finished and the next running task and the preempted task are equivalent, the handler immediately returns from the interrupt without the need of restoring any registers. Otherwise, the registers are saved and then restored for the next running task.

Fig. 5 shows the performance gain of the optimization. The measurement (A) shows an invocation of the event interrupt handler without the optimization and (B) with the optimization. An invocation begins with the first rising edge and ends with the second falling edge. The time between the two pulses is the time it takes the microkernel to determine in the event loop whether microcode needs to be executed. Here, microcode is not executed. The mean time it takes to go through the event loop with a single trigger binding in the trigger queue and a single thread instance in the thread set is around 2μs in our experiments. The first pulse of (A) shows the time (1.26μs) it takes to set the next timer interrupt and to save the registers. Each pulse includes a 200ns I/O overhead to toggle from zero to one and back, which we exclude in the numbers. Thus the actual execution time of the handler is 400ns shorter than shown in the figure. The second pulse of (A) shows the time (1.05μs) it takes to restore the registers. Thus the minimum execution time to

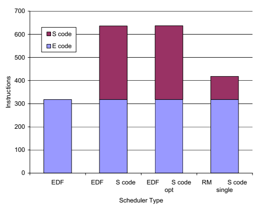


Figure 6: E and S code size for 100 tasks

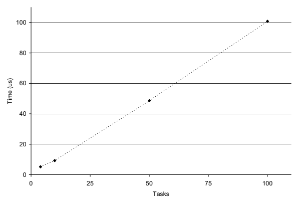


Figure 7: E code execution overhead

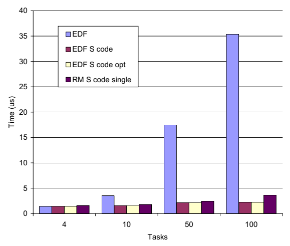


Figure 8: Scheduling overhead (EDF scheduling and S code execution)

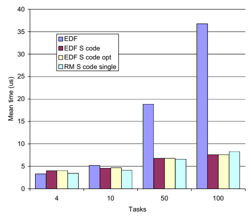


Figure 9: E and S code execution overhead

handle a timer interrupt without the optimization is 4.31μs. The first pulse of (B) shows the time (0.43μs) it takes to set only the next timer interrupt while the second pulse of (B) shows only the time (200ns) to toggle from zero to one and back. Thus the minimum execution time to handle a timer interrupt with the optimization is 2.43μs. The measurement (C) shows a task completion followed by an invocation of the microkernel that determines a next running task that requires initializing the processor context. The first pulse of (C) shows only the time (200ns) it takes to toggle from zero to one and back while the second pulse of (C) shows the time (0.94μs) it takes to initialize the processor context and load the registers. Thus the minimum time it takes to complete the execution of a task and to determine the next running task (using S code) is 2.94μs. Here, the optimization has no effect.

7. MICROKERNEL BENCHMARKS

The binary code size of the programmable microkernel is 8kB, which includes the I/O handling code of HelyOS. Thus, due to its small size, the microkernel can even be used on small embedded devices with limited CPU and memory resources. For the benchmarks, the microkernel is invoked at 1kHz. Thus tasks are preempted every 1ms, called the *microkernel period*. We evaluated the microkernel on four periodic, non-harmonic task sets with 4, 10, 50, and 100 tasks. Each set consists of four equally large task groups with 16.66Hz, 33.33Hz, 50Hz, and 100Hz tasks. The task sets are described by E code. There is an E code block for each instant in the hyperperiod of the task sets at which tasks are released.

Micro-Code Size. Each task set is scheduled using four different methods: (1) an EDF scheduler; (2) preemptive (EDF) S code generated by the EDF scheduler; (3) non-preemptive (EDF) S code; and (4) preemptive (RM) S code generated by a rate-monotonic (RM) scheduler. We have implemented the EDF scheduler as default task scheduler of the microkernel. The non-preemptive (EDF) S code was generated from the preemptive (EDF) S code by reordering task execution, which was not always possible. The preemptive (RM) S code was generated such that at each instant all tasks are dispatched in the order of decreasing frequencies.

In order to avoid WCET analysis of task code, we generated all S code at run time and implemented the tasks without branching such that the actual execution times are close to the WCETs. The task code consists of integer operations (the StrongARM has no FPU) and I/O operations in order to visualize the task behavior on an oscilloscope. For each task set we used three different task implementations with short, medium, and long execution times. We ran a total of 48 different test cases. For each test, we measured (per invocation of the microkernel): (1) the overall microcode execution time as well as its parts: (2) the E code execution time; (3) the S code execution time; (4) the EDF scheduler execution time. We also measured the total CPU utilization (U) and counted the number of task preemptions (P) per hyperperiod (60ms) as well as the number of S code and E code instructions. For the time measurements, we used the internal 3.6864MHz OS timer of the StrongARM. The test results are summarized in Table 1, 2, and 3.

Fig. 6 shows the microcode size (E code and S code) for 100 tasks that are scheduled according to the four different scheduling methods (the E code is always the same). The preemptive (RM) S code is shorter than the (EDF) S code because the same code is re-used on all phases of the 16.66Hz hyperperiod. The S code statically dispatches each task according to its frequency but independently of its execution time. In other words, tasks may have already completed before they are (unnecessarily) dispatched again.

E code Execution Overhead. Fig. 7 shows E code execution times with respect to the number of tasks released by the E code. Independently of the task scheduling method, the increase of E code execution times is linear in the number of tasks because the number of E code instructions grows in a linear fashion with the number of tasks. The non-linearity shown in Fig. 7 is due to the (fixed) microkernel overhead.

Scheduling Overhead. Fig. 8 shows the scheduling overhead of the microkernel. The EDF scheduler maintains a sorted list to determine the next running task. The S code we consider here, on the other hand, determines the next running task directly through current control locations in S code, which explains the near constant growth. Note that more efficient implementations of our EDF scheduler using multiple lists [21] are possible but do not achieve S code performance. The S code shown here trades space for time although the S code size even for 100 tasks is still small. The preemptive (RM) S code is slower than other S code because the (RM) S code dispatches more often already completed tasks. The execution times of preemptive and non-preemptive (EDF) S code for 50 and 100 tasks are equivalent since preemptions do not occur in both cases.

System Overhead. Fig. 9 shows the total system overhead in terms of microcode execution times (E code and S code execution times plus the execution time of the EDF scheduler when used). With S code, it is possible to keep the system overhead under 10μs even for 100 tasks.

CPU Utilization. CPU utilization improves with the number of tasks when switching from the EDF scheduler to preemptive (EDF) S code. With 100 tasks, S code performs 35% better than the traditional EDF scheduler (51% utilization versus 78% utilization).

8. RELATED WORK

After ambiguous first experiences in the early 90's, the microkernel approach now seems to become a successful and mature technology. Several efficient microkernel implementations [16, 1, 2] have demonstrated the advantages of the architecture. These microkernels were developed to be minimal and highly flexible, so that both conventional and nonclassical operating systems can be built or adapted to run on top of them. Moreover, these microkernels support extensibility, customizability, robustness, reliability, fault tolerance, protection, and security. Supported by these demonstrations that the performance and flexibility of microkernel-based systems are usable in practice [7], we have extended the microkernel idea to the real-time domain. In particular, we have focused on achieving the schedulability, portability, and composability of real-time systems. The core paradigm we introduced for achieving these properties is the separation of reacting (E code) from scheduling (S code). Our microkernel architecture can be thought of as a simple, highly optimized, programmable interrupt handler (E machine), and a programmable scheduler and dispatcher (S machine). There have been other systems that introduce a single, system-wide, programmable event-handler [15, 4]. However, our approach differs by explicitly supporting the composability and predictability of real-time behavior. Composability is the ability to compose different real-time programs (microcode) off-line, and more interestingly, during run-time, while guaranteeing not only the schedulability but also the unchanged behavior of the composite program.

There have been other microkernel investigations for the real-time domain. For example, Real-Time Mach [19] targets distributed real-time operating systems. The RT-Mach approach is built around resource reservation and allocation, in that each application is allowed to use resources only if the global real-time requirements can be guaranteed. Recently RT-Mach ideas have been extended and ported to the Linux/RK system [17]. However, these architectures do address scheduling in system-centric manner. We propose a different solution in which the scheduling and reactivity of the system are decomposed. Consequently we achieve a higher degree of portability and composability, by guaranteeing predictability and schedulability.

9. REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. USENIX Summer Conference*, pages 93–113, 1986.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. SOSP*. ACM Press, 1995.

[3] Y. Cai and M.C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[4] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: A programming model for event-driven embedded systems. In *Proc. SAC*. ACM Press, 2003.

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. EMSOFT*, LNCS 2211, pages 469–485. Springer, 2001.

[6] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[7] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proc. SOSP*. ACM Press, 1997.

[8] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. GIOTTO: A time-triggered language for embedded programming. *Proc. of the IEEE*, 91(1):84–99, 2003.

[9] T.A. Henzinger and C.M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proc. PLDI*, pages 315–326. ACM Press, 2002.

[10] T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time-safety checking for embedded programs. In *Proc. EMSOFT*, LNCS 2491, pages 76–92. Springer, 2002.

[11] T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In *Proc. EMSOFT*, LNCS 2855, pages 241–256. Springer, 2003.

[12] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. RTSS*, pages 129–139. IEEE Computer Society Press, 1991.

[13] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, 2001.

[14] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[15] P. Levis and D. Culler. Maté: a virtual machine for tiny networked sensors. In *Proc. ASPLOS*, 2002.

[16] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[17] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proc. MMCN*, 1998.

[18] M.A.A. Sanvido. A computer system for model helicopter flight control; technical memo nr. 3: The software core. Technical Report 317, ETH Zürich, Institute for Computer Systems, 1999.

[19] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *Proc. USENIX Mach Workshop*, pages 73–82, 1990.

[20] N. Wirth and J. Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. ACM Press, 1992.

[21] K.M. Zuberi, P. Pillai, and K.G. Shin. EMERALDS: a small-memory real-time microkernel. In *Proc. SOSP*, pages 277–299. ACM Press, 1999.

Appendix

For more details on the E and S machine, we also refer to [9] and [11], respectively. The figures 6–9 have been compiled from the data shown in the tables 1–3.

Algorithm 1 The Event Loop of the Integrated Machine

```
$YieldToTask := \mathsf{false}$
while $\neg$ $YieldToTask$ do
  if there is a completed task $t$ in $ThreadSet$ then
    choose a thread instance $(t, b, m, a, s)$
      and remove it from $ThreadSet$
    invoke the microcode interpreter
      with $ReferenceTime := s$
      and $ProgramCounter := b$
  else if there is an enabled trigger in $TriggerQueue$ then
    choose the first enabled trigger binding $(g, a, s)$
      and remove it from $TriggerQueue$
    invoke the microcode interpreter
      where $ReferenceTime$ is the current clock value
      and $ProgramCounter := a$
  else if there is an enabled thread in $ThreadSet$ then
    choose an enabled thread instance $(t, b, m, a, s)$
      and remove it from $ThreadSet$
    invoke the microcode interpreter
      with $ReferenceTime := s$
      and $ProgramCounter := a$
  else
    $YieldToTask := \mathsf{true}$
  end if
end while
if $ThreadSet \neq \emptyset$ then
  if there is a task $t \neq \mathsf{idle}$ in $ThreadSet$ then
    $RunningTask := t$
  else
    $RunningTask := \mathsf{idle}$
  end if
else
  invoke a task scheduler if present
end if
```

Algorithm 2 The Microcode Interpreter

```
$YieldToThreads := \mathsf{false}$
while $ProgramCounter \neq \perp$ and $\neg$ $YieldToThreads$ do
  $i := Instruction(ProgramCounter)$
  if $\mathtt{call}(d) = i$ then
    execute the driver $d$
  else if $\mathtt{release}(t) = i$ then
    emit signal on the release port of the task $t$
  else if $\mathtt{future}(g, a) = i$ then
    append the trigger binding $(g, a, s)$ to $TriggerQueue$
    where $s$ is the current state of the trigger ports of $g$
  else if $\mathtt{dispatch}(t, m, a) = i$ then
    insert the thread instance
      $(t, Next(ProgramCounter), m, a, ReferenceTime)$
       into $ThreadSet$
    $YieldToThreads := \mathsf{true}$
  else if $\mathtt{idle}(m) = i$ then
    insert the thread instance
      $(\mathsf{idle}, \perp, m, Next(ProgramCounter), ReferenceTime)$
       into $ThreadSet$
    $YieldToThreads := \mathsf{true}$
  else if $\mathtt{fork}(a) = i$ then
    insert the thread instance $(\mathsf{idle}, \perp, \mathsf{true}, a, s)$ into
      $ThreadSet$ where $s$ is the current clock value
  end if
  $ProgramCounter := Next(ProgramCounter)$
end while
```

Algorithm 3 The Bootstrap Program

```
disable all interrupts
add $(\mathsf{idle}, InitialContext(\mathsf{idle}))$ to $TaskSet$
$TriggerQueue := \perp$; $ThreadSet := \emptyset$
$SystemProgram := \perp$; $MicroCode := \perp$
$RunningTask := \mathsf{idle}$; $Run := \mathsf{true}$
bind HelyOS I/O handler to I/O interrupts
bind the event interrupt handler
  to system clock interrupt
enable all interrupts
invoke the idle task
disable all interrupts
shutdown system
```

Algorithm 4 The Idle Task

```
while $Run$ do
  receive $Command$ from host
  if $Command = \mathtt{switchcode}$ then
    receive $NewMicroCode$ from host
    if checking $NewMicroCode$ integrity fails then
      $NewMicroCode := \perp$
    end if
  else if $Command = \mathtt{shutdown}$ then
    $Run := \mathsf{false}$
  end if
  if $NewMicroCode \neq \perp$ then
    disable event interrupts
    if $TaskSet = \{(\mathsf{idle}, c)\}$ then
      $MicroCode := NewMicroCode$; $NewMicroCode := \perp$
      $TriggerQueue := \langle(\mathsf{true}, 0, \emptyset)\rangle$; $ThreadSet := \emptyset$
    end if
    enable event interrupts
  end if
  send logging information to host
end while
```

Algorithm 5 The Event Interrupt Handler

```
disable event interrupts
save registers in $ProcessorContext$
$PreemptedTask := RunningTask$
invoke the event loop (Alg. 1)
if $RunningTask \neq PreemptedTask$ then
  update $(PreemptedTask, ProcessorContext)$ in $TaskSet$
  if there is a $RunningTask$ instance in $TaskSet$ then
    get $(RunningTask, ProcessorContext)$ from $TaskSet$
  else
    $ProcessorContext := InitialContext(RunningTask)$
    add $(RunningTask, ProcessorContext)$ to $TaskSet$
    set stack pointers according to $ProcessorContext$
    leave interrupt handler by invoking $RunningTask$
      with enabled event interrupts
    disable event interrupts
    remove $RunningTask$ instance from $TaskSet$
    invoke the event loop (Alg. 1)
    invoke the completion handler
    // never returns here
  end if
end if
restore registers from $ProcessorContext$
return from interrupt with enabled event interrupts
```

Algorithm 6 The Completion Handler

```
if there is a $RunningTask$ instance in $TaskSet$ then
  get $(RunningTask, ProcessorContext)$ from $TaskSet$
  restore registers from $ProcessorContext$
  switch context and enable event interrupts
  // never returns here
else
  $ProcessorContext := InitialContext(RunningTask)$
  add $(RunningTask, ProcessorContext)$ to $TaskSet$
  invoke $RunningTask$ with enabled event interrupts
  disable event interrupts
  remove $RunningTask$ instance from $TaskSet$
  invoke the event loop (Alg. 1)
  invoke the completion handler
  // never returns here
end if
```

Tasks	Mode	Microcode (μs)			U	P	S code	E code	EDF	E+S code	S code
(#)		peak	min	average	(%)	(#)	(μs)	(μs)	(μs)	(#)	(#)
4	EDF	12.207	1.356	3.778	0.818	4	0	4.653	1.473	30	0
	EDF sc	20.888	2.170	3.884	0.818	4	1.411	5.574	0	60	30
	EDF sco	infeasible								64	34
	RM scs	14.377	1.356	3.545	0.817	5	1.475	5.356	0	28	4
10	EDF	21.159	1.356	6.129	0.917	4	0	8.792	3.658	54	0
	EDF sc	26.855	2.170	4.472	0.850	4	1.565	9.510	0	107	53
	EDF sco	infeasible								111	57
	RM scs	18.175	1.356	4.412	0.9001	5	1.724	9.330	0	58	10
50	EDF	129.120	1.356	21.767	0.950	4	0	44.972	18.508	174	0
	EDF sc	120.710	1.899	6.822	0.750	0	2.207	52.882	0	347	173
	EDF sco	115.020	1.899	6.843	0.750	0	2.212	52.885	0	351	177
	RM scs	93.859	1.356	6.746	0.883	5	2.393	47.784	0	218	50
100	EDF	287.000	1.356	38.573	0.786	4	0	81.749	36.074	318	0
	EDF sc	247.400	1.899	7.628	0.517	0	2.258	116.190	0	636	318
	EDF sco	248.210	2.170	7.643	0.518	0	2.248	116.790	0	640	322
	RM scs	159.780	1.356	8.361	0.550	2	3.604	98.175	0	412	100

Table 1: Tasks with long execution times

Tasks	Mode	Microcode (μs)			U	P	S code	E code	EDF	E+S code	S code
(#)		peak	min	average	(%)	(#)	(μs)	(μs)	(μs)	(#)	(#)
4	EDF	12.207	1.356	3.087	0.350	1	0	4.608	1.384	30	0
	EDF sc	20.888	2.170	4.065	0.350	1	1.428	5.513	0	60	30
	EDF sco	20.888	2.170	4.061	0.351	0	1.439	5.528	0	61	31
	RM scs	13.292	1.356	3.123	0.350	1	1.639	5.356	0	28	4
10	EDF	21.159	1.356	4.730	0.350	1	0	8.647	3.460	54	0
	EDF sc	27.127	2.170	4.676	0.350	1	1.595	9.588	0	108	54
	EDF sco	27.127	27.127	4.682	0.350	0	1.601	9.668	0	109	55
	RM scs	16.819	1.356	4.032	0.350	1	1.821	9.340	0	58	10
50	EDF	126.680	1.356	17.929	0.417	1	0	43.574	16.983	174	0
	EDF sc	113.390	1.899	6.799	0.334	0	2.152	52.417	0	348	174
	EDF sco	116.100	1.899	6.816	0.333	0	2.163	52.701	0	349	175
	RM scs	87.077	1.356	6.466	0.350	1	2.442	46.375	0	218	50
100	EDF	281.850	1.356	36.765	0.601	2	0	81.655	35.190	318	0
	EDF sc	247.120	1.899	7.584	0.368	0	2.229	113.880	0	636	318
	EDF sco	251.740	1.899	7.591	0.367	0	2.230	115.550	0	638	320
	RM scs	159.510	1.356	8.308	0.384	1	3.647	95.773	0	412	100

Table 2: Tasks with medium execution times

Tasks	Mode	Microcode (μs)			U	P	S code	E code	EDF	E+S code	S code
(#)		peak	min	average	(%)	(#)	(μs)	(μs)	(μs)	(#)	(#)
4	EDF	12.207	1.356	3.091	0.352	1	0	4.693	1.382	30	0
	EDF sc	20.888	2.170	4.063	0.350	1	1.435	5.468	0	60	30
	EDF sco	20.888	2.170	4.061	0.350	0	1.441	5.548	0	61	31
	RM scs	13.292	1.356	3.123	0.350	1	1.639	5.356	0	28	4
10	EDF	21.159	1.356	4.730	0.350	1	0	8.661	3.460	54	0
	EDF sc	27.127	2.170	4.677	0.351	1	1.595	9.606	0	108	54
	EDF sco	27.127	2.170	4.682	0.352	0	1.601	9.683	0	109	55
	RM scs	16.819	1.356	4.032	0.350	1	1.821	9.382	0	58	10
50	EDF	131.020	1.356	16.902	0.234	1	0	41.721	16.905	174	0
	EDF sc	116.370	1.899	6.742	0.200	0	2.160	51.498	0	348	174
	EDF sco	118.270	1.899	6.754	0.200	0	2.169	51.701	0	349	175
	RM scs	89.247	1.356	6.366	0.200	0	2.460	45.442	0	218	50
100	EDF	300.560	1.356	34.913	0.349	1	0	81.572	34.844	318	0
	EDF sc	242.510	1.899	7.580	0.200	0	2.216	115.570	0	636	318
	EDF sco	243.600	1.899	7.580	0.200	0	2.211	115.700	0	637	319
	RM scs	160.860	1.356	8.237	0.200	0	3.641	96.869	0	412	100

Table 3: Tasks with short execution times