A Programmable Microkernel for Real-Time Systems

Christoph M. Kirsch  Thomas A. Henzinger  Marco A.A. Sanvido

Report No. UCB/CSD-3-1250
June 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720
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) and proactivity (task scheduling) are fully programmable. The microkernel, which we implemented on a StrongARM processor, consists of two interacting virtual machines a reactive E (Embedded) machine and a proactive S (Scheduling) machine. The system code 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 thus 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 generally in the 0.2(0.3)% range.
<table>
<thead>
<tr>
<th>16. SECURITY CLASSIFICATION OF:</th>
<th>17. LIMITATION OF ABSTRACT</th>
<th>18. NUMBER OF PAGES</th>
<th>19a. NAME OF RESPONSIBLE PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. REPORT unclassified</td>
<td>Same as Report (SAR)</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>b. ABSTRACT unclassified</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c. THIS PAGE unclassified</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18
A Programmable Microkernel for Real-Time Systems

Christoph M. Kirsch  Thomas A. Henzinger  Marco A.A. Sanvido

cm@eecs.berkeley.edu  tah@eecs.berkeley.edu  msanvido@eecs.berkeley.edu

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) and proactivity (task scheduling) are fully programmable. The microkernel, which we implemented on a StrongARM processor, consists of two interacting virtual machines, a reactive E (Embedded) machine and a proactive S (Scheduling) machine. The system code 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, generally in the 0.2–0.3% range.

1. INTRODUCTION

In [9], we advocated the E (Embedded) machine as a portable target for compiling hard real-time code. In this paper, we show that the E machine together with a programmable scheduler, which is called S (Scheduling) 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 system code 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,
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.

Interface. The E machine has two input and two output interfaces as shown in Figure 1.

Environment inputs. The physical processes communicate information to the E machine through environment ports, such as clocks and sensors.

Software inputs. The application software processes, called tasks, communicate information through task ports to the E machine.

Driver outputs. The E machine communicates information to the physical processes and to the tasks by calling system processes, called drivers, which write to driver ports.

Release outputs. 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.

Input Events. 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, \( p_s' \neq p_s \) is a trigger on \( p_s \), where \( p_s' \) refers to the current sensor reading, and \( p_s \) refers to the most recent previous sensor reading. A time trigger on an environment (clock) port \( p_c \) is modeled using a predicate \( p_c' = p_c + \delta \) where \( \delta \) is the number of ticks to wait before the trigger goes off.

Functional Code. Tasks, drivers, and triggers are functional code that is external to the E machine and must be implemented in some programming language like C. To protect IP, tasks, drivers, and triggers may be given as binary executables to which E code refers through symbolic references. The execution of drivers and tasks is supervised by E code, which monitors input events through triggers.

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. In favor of a simpler presentation, we assume that the set of ports on which a task operates is fixed. Logically, a task computes a function from its driver and task ports to its task ports provided its ports are not modified when the task is preempted. Each task has a special task port, called completion port, which indicates that the task completed execution.

A driver is a piece of system-level code, which typically facilitates a communication activity. 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 kind of port and write to driver ports. Similarly to tasks, 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. In other words, interrupts that implement input events are disabled during the execution of a driver.

A trigger is a piece of system-level code that monitors the occurrences of input events. A trigger may observe environment and task ports (including completion 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. Similarly to drivers, triggers are evaluated in logical zero time with event interrupts disabled.

E Code. There are essentially three non-control-flow E code instructions. In an actual implementation of the E machine, E code also has control-flow instructions such as conditional and absolute jumps, which we have omitted here.

A call(d) instruction initiates the execution of a driver d. 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 schedule(t) instruction 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. t does not execute before the E machine relinquishes control of the processor to the scheduler. The schedule 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. The first pair whose trigger is enabled determines the next action of the E machine.

An E Code Example

We use as example a simplified version of the control program for a model helicopter built at ETH Zürich [4]. Figure 2 shows the topology of the program: we denote ports by bullets, tasks by rectangles, drivers by diamonds, and triggers by circles. Consider the helicopter in hover mode m.

There are two tasks, both implemented in native code: the control task t1, and the navigation task t2. The navigation task processes GPS input every 10 ms and provides the processed data to the control task. The control task reads additional sensor data (not modeled here), computes a control law, and writes the result to actuators (reduced here to a single port). The control task is executed every 20 ms. The release port p1 of t1 indicates whether t1 has been released to run. Similarly, p2 is the release port of t2.

The data communication requires three drivers: a sensor driver d1, which provides the GPS data to the navigation task; a connection driver d2, which provides the result of the navigation task to the control task; and an actuator driver d3, which loads the result of the control task 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 GPS 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. Here is a high-level Giotto [8] description of the program timing:

```plaintext
mode m() period 20 {
  actfreq 1 do p_a(d1);
  taskfreq 1 do t1(d1);
  taskfreq 2 do t2(d1);
}
```

The “actfreq 1” statement 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 Figure 3.

The E code consists of two blocks. The block at address a1 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 a2. When the block finishes, the trigger queue of the E machine contains the trigger g bound to address a2, and the release ports of the two tasks, t1 and t2, 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. The E machine may observe when a task completes by checking the completion port of the task.

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 a2 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 a2 block is executed. It calls the sensor driver, which updates a port read by task t2. There are two possible scenarios: the earlier invocation of task t2 may already have completed with a signal on the completion port of t2. In this case, the E code proceeds to release t2 again, and to trigger the a1 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 t2 still incomplete, i.e., the completion port of t2 has not yet signaled completion. In this case, the attempt by the sensor driver to overwrite a port read by t2 causes a run-time exception, called time-safety violation. At 20 ms, when ports read by both tasks t1 and t2 are updated, and ports written by both t1 and t2 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 20\( n \) ms, for \( n \geq 0 \), completes by 20\( n + 10 \) ms; (2) each invocation of task \( t_2 \) at 20\( n \) ms completes by 20\( n + 10 \) ms; and (3) each invocation of task \( t_2 \) at 20\( n + 10 \) ms completes by 20\( n + 20 \) ms. Therefore, a necessary requirement for time safety is \( \delta_1 + 2\delta_2 < 20 \), where \( \delta_1 \) is the WCET of task \( t_1 \), and \( \delta_2 \) is the WCET of \( t_2 \). If this requirement is satisfied, then a scheduler that gives priority to \( t_2 \) over \( t_1 \) guarantees time safety.

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 Figure 2. Thus the Giotto compiler needs to ensure time safety 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, time safety 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.

Figure 4 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 schedule instructions (not shown here).

## E Code Execution

We define the E code semantics operationally using a pseudo-code description.

### Data Structures.

An E program consists of ports, drivers, tasks, and triggers, as well as E code. The ports are essentially the memory on which the functional code operates. Ports are given as addresses to memory. We use a function \( \text{Instruction}(a) \) to fetch the E code instruction at the address \( a \). The function \( \text{Next}(a) \) returns the address of the next instruction; if there is no next instruction, then the function returns \( \bot \). This convention is consistent with any control-flow instructions, structured or unstructured, whose choice is of practical importance but entirely orthogonal to the issues discussed here. A configuration of an E program consists of the port values (content of memory), a trigger queue of trigger bindings, and an E code program counter. A trigger binding \((g, a, s)\) is created by a future\((g, a)\) instruction that appends it to the trigger queue, where \( s \) is the current state of the trigger ports of \( g \). In the future, \( g \) will be evaluated with respect to \( s \) and the new state of the trigger ports.

![Figure 3](image_url)

### Algorithm 1 The Embedded Machine

```plaintext
while there is an enabled trigger in TriggerQueue do
    choose the first enabled trigger binding \((g, a, s)\)
    and remove it from TriggerQueue
    invoke the E code interpreter (Algorithm 2)
    with ProgramCounter := a
end while
```

### Operational Semantics.

The execution of an E program starts with a single trigger binding \((\text{true}, a_0, \emptyset)\) in the trigger queue where \( a_0 \) is the address of an initial E code instruction. As soon as the first event interrupt occurs, all event interrupts are disabled and the E machine (Algorithm 1) is invoked (thus it is still possible for low-level interrupts to preempt the E machine, as long as they do not interfere with the triggering mechanism of the machine). Then the E machine scans the trigger queue for enabled trigger bindings. Each enabled trigger binding \((g, a, s)\) is removed from the queue and the E code at the address \( a \) is executed by invoking the E code interpreter (Algorithm 2) with the program counter set to \( a \). Initially, the interpreter is invoked with the program counter set to the initial address \( a_0 \). The interpreter fetches and executes E code instructions until the program counter is set to \( \bot \). In an actual implementation, we use a return instruction for which Next returns \( \bot \).

The E code interpreter implements the E code instructions as follows: a call\((d)\) instruction executes the code of the driver \( d \). Here, the interpreter may check for a possible time-safety violation by verifying that all currently released tasks neither read from driver ports written by \( d \) nor write to task ports read by \( d \). If there is a violating task \( t \), the interpreter may throw a run-time exception, i.e., not execute the call instruction but jump to some other E code that handles the situation [9], e.g., terminates \( t \) and executes some other driver. We have implemented run-time exceptions using a compiler-generated \( n \times m \) matrix of bits where \( n \) is the number of tasks and \( m \) is the number of drivers in the E program. The \((i, j)\)th bit in the matrix is \( 1 \) if and only if the \( i \)th task shares ports with the \( j \)th driver. An n-vector of bits keeps track of the released tasks. Then a fast AND operation between the vector and a row of the matrix tells us whether a time-safety violation.

A schedule\((t)\) instruction releases the task \( t \) by emitting a signal on the release port of \( t \). Again, the interpreter may check for a possible time-safety violation by verifying that all currently released tasks do not share task ports with \( t \). If there is a violating task, the interpreter may either throw a run-time exception, or else release \( t \) anyway, which then requires, however, a more sophisticated task scheduler. A future\((g, a)\) instruction appends a trigger binding \((g, a, s)\)

### Algorithm 2 The E Code Interpreter

```plaintext
while ProgramCounter ≠ \bot do
    i := Instruction(ProgramCounter)
    if call\((d) = i \) then
        execute the driver \( d \)
    else if schedule\((t) = i \) then
        emit signal on the release port of the task \( t \)
    else if 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 \( g \)
    end if
    ProgramCounter := Next(ProgramCounter)
end while
```
to the trigger queue where \( s \) is the current state of the ports monitored by the trigger \( g \). The E code at the address \( a \) will be executed as soon as \( g \) is enabled.

When there are no more enabled trigger bindings in the trigger queue, the E machine enables the event interrupts and relinquishes control of the processor. At this point, a task scheduler, e.g., Algorithm 3, may take over and schedule the released tasks. The task scheduler may apply any scheduling strategy to choose the next running task including a special idle task. Then the task scheduler relinquishes control of the processor to the running task until the next event interrupt occurs at which the E machine is invoked again.

### 3. The Scheduling Machine

The S machine is a virtual machine that determines the temporal order of task execution: it interpretes 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.

**Interface.** The S machine reads from three input interfaces to determine a running task.

- *Release inputs.* The release of tasks is communicated to the S machine through release ports.
- *Software inputs.* The tasks communicate information including their completion to the S machine through task ports.
- *Clock input.* An external clock writes to a clock port that is read by the S machine to time-slice tasks.

**Timeouts.** 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 \( \delta \) value, e.g., \( 10\text{ms} \). A timeout *expires* if it evaluates to 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.

**S Code.** There are essentially three non-control-flow S code instructions. Again, in an actual implementation of the S machine, S code also has control-flow instructions such as conditional and absolute jumps, which we have omitted here.

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\((t, m, a)\) 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. Figure 5 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.

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.

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

Figure 6 shows an execution trace of the S code. The S code guarantees the time-safe execution of the E code in Figure 3 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 Figure 7 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\((t_1, \text{release}(t_2))\) instruction

---

**Algorithm 3** A Task Scheduler

```
if there are released tasks then
  RunningTask := Schedule(released tasks)
else
  RunningTask := idle
end if
```

---

Figure 5: Synchronous S code

Figure 6: An execution trace of the S code of Figure 5
executes t₁ until either t₁ completes or t₂ is released again. If t₂ is released before t₁ completes, t₁ is preempted and the S machine continues executing the S code at the address a₁. Here, t₂ is dispatched until completion before t₁ resumes its execution. If, however, t₁ would have completed before t₂ was released, the dispatch(t₁) instruction following the instruction at a₁ would have no effect on t₁. The execution trace of the S code corresponds to the execution trace of the E code using an EDF scheduler as shown in Figure 4. Thus the S code describes an EDF scheduler 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 Figure 8 again allows the task t₁ to be preempted by the release of t₂ but then resumes the execution of t₁ instead of executing t₂. If t₁ does not complete before t₂ is released again, t₁ is dispatched resuming its execution before t₂ is dispatched. Figure 9 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 6, 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. As shown in Figure 10, 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, 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.

The S code in Figure 11 enables a time-sliced execution of the tasks t₁ and t₂. Here, a time slice is 5ms long. First,

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

Figure 8: Non-preemptive S code

```
t₂ is dispatched for 5ms, then t₁ for 5ms. After 10ms, t₂ is dispatched again for 5ms, then t₁ for 5ms. The S code guarantees time safety provided t₁ has a WCET of 10ms and t₂ has a WCET of 5ms. An execution trace of the S code is shown in Figure 12. Executing the S code in Figure 13 results in the same execution trace. Here, the S code forks two threads: the S code at the address a₁ dispatches t₁ whereas the S code at the address a₂ dispatches t₂. Both tasks are dispatched in disjoint time slots. We say that the S code is time-sharing. Thus a scheduler may generate S code in a modular fashion with one thread of S code for each task that needs to be dispatched.

The S code in Figure 14 demonstrates multi-processor S code as it is generated by a prototypical Giotto compiler that targets distributed architectures. Suppose there are two hosts h₁ and h₂ where each host runs an S machine. The hosts are connected by an Ethernet network. Moreover, there is a clock synchronization service running between both S machines. h₁ executes the S code at the address a₁, h₂ executes the S code at the address a₂ in a synchronized fashion with h₁. Both hosts also execute different subsets of the E code of Figure 3. However, here it is only important that the intertask driver d₁ of Figure 2 will be invoked by h₁. Recall that the task t₁ receives the output of the task t₂ at a rate of 20ms when d₁ is invoked. Thus the output of the second invocation of t₂ must be sent across the network after the invocation completed and before the 20ms period is finished. The S code on h₂ dispatches a task t₂ that sends the output of t₂ across the network within the 15-20ms time slot. The S code on h₁ dispatches at the same time a task s₂₁ that receives the output. Figure 15 shows an execution trace of the distributed system. Note that another interesting target platform for this kind of S code is the time-triggered architecture [15].

**S Code Execution**

We define the S code semantics operationally using a pseudo-code description.

**Data Structures.** An S program consists of ports, tasks, and timeouts, as well as S code. Similar to E code, we use a function Instruction(a) to fetch the S code instruction at the address a and a function Next(a) that returns the address

Figure 10: The relative overhead for executing different classes of S code and an EDF scheduler.
of the next instruction. A configuration of an S program consists of the port values, a set of thread instances called the thread set, a reference time, a running task, and an S code program counter. The reference time is the clock value at the instant when the current thread was started. The running task is the currently dispatched task. A thread instance is of the form \((t, b, m, a, s)\), which is created by the three S code instructions. \(t\) is either a task or the special idle task for which \(b = \perp\). Otherwise, \(b\) is the address from which the S machine continues executing when \(t\) completes before the timeout \(m\) expires. When \(m\) expires before \(t\) completes, we say that the thread instance is enabled and the S machine continues executing S code at the address \(a\). The clock value \(s\) is the reference time at which the thread instance was started by a fork instruction.

![Figure 12: An execution trace of the S code of Figure 11](image)

\[
\begin{align*}
a_0: & \quad \text{fork}(a_0) \\
a_1: & \quad \text{idle}(5\text{ms}) \\
      & \quad \text{dispatch}(t_1, 10\text{ms}) \\
      & \quad \text{idle}(10\text{ms}) \\
      & \quad \text{fork}(a_1) \\
\end{align*}
\]

![Figure 13: Multi-threaded and time-sliced S code](image)

![Figure 14: Multi-processor S code](image)

![Figure 15: An execution trace of the S code of Figure 14](image)

Algorithm 4 The S Code Interpreter

YieldToThreads := 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 S code interpreter (Algorithm 5)
    with ReferenceTime := \(s\)
    and ProgramCounter := \(b\)
  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 S code interpreter (Algorithm 5)
    with ReferenceTime := \(s\)
    and ProgramCounter := \(a\)
  else
    YieldToTask := true
  end if
end while
if there is a task \(t \neq \text{idle}\) in ThreadSet then
  RunningTask := \(t\)
else
  RunningTask := \(\text{idle}\)
end if

Algorithm 5 The S Code Interpreter

YieldToThreads := false
while ProgramCounter \(\neq \perp\) and \(\neg\) YieldToThreads do
  if dispatch \((t, m, a)\) then
    insert the thread instance \((t, s(t, m, a))\)
    into ThreadSet
    YieldToThreads := true
  else if idle \((m)\) then
    insert the thread instance \((t, \perp, m, s(t, m))\)
    into ThreadSet
    YieldToThreads := true
  else if fork \((a)\) then
    insert the thread instance \((t, \perp, true, a)\)
    into ThreadSet where \(s\) is the current clock value
  end if
end while

Operational Semantics. The execution of an S program starts with a single thread instance \((\text{idle}, \perp, \text{true}, a_0, 0)\) in the thread set where \(a_0\) is the address of an initial S code instruction. As soon as the first event interrupt occurs, all event interrupts are disabled and the S machine (Algorithm 4) is invoked. Then the S machine scans the thread set for completed tasks and enabled thread instances. Each thread instance \((t, b, m, a, s)\) containing a completed task \(t\) (or an expired timeout \(m\)) is removed from the thread set and the S code at the address \(b\) is executed by invoking the S code interpreter (Algorithm 5) with the program counter set to \(b\). Thus, similar to the E code interpreter, the S code interpreter is initially invoked with the program counter set to the initial address \(a_0\).

The S code interpreter implements the S code instructions as follows: a dispatch \((t, m, a)\) instruction creates a thread instance \((t, b, m, a, s)\) where \(b\) is the address of the next instruction and \(s\) is the reference time at which the current thread was started. Here, the interpreter may check for a possible time-share violation by verifying that all thread instances in the thread set contain the special idle task. If there is a thread instance with a task in the thread set, the
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. An important aspect of an integrated implementation is to determine 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 system code that may consist of E code and S code instructions. Figure 16 shows an example of system code that implements the E code of Figure 3 interacting with the S code of Figure 5. Logically, after the E code at the address \( a_0 \) in Figure 3 has been executed, the S machine is invoked to execute the S code at the address \( a_0 \) in Figure 5. Operationally, instead of executing E and S code in different programs, we append the S code to the E code as shown in the example and use the integrated machine. For example, after executing the future(\( g, a_1 \)) instruction, the machine immediately proceeds to the dispatch(\( t_2 \)) instruction, creates a thread instance with the task \( t_2 \), and then executes \( t_2 \). Logically, the E machine terminates and is then followed by an invocation of the S machine, which evaluates the release ports to determine the release status of the tasks. System code is a more succinct representation of this behavior. Note that the idle and fork instructions in the S code of Figure 5 are not necessary in the system code of Figure 16.

![Figure 16: System code implementing the E code of Figure 3 interacting with the S code of Figure 5](image)

**Algorithm 6 The Event Loop of the Integrated Machine**

```
YieldToTask := false
while ¬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 system code interpreter
        with \( \text{ReferenceTime} := s \) and \( \text{ProgramCounter} := b \)
    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 system code interpreter
        where \( \text{ReferenceTime} \) is the current clock value
        and \( \text{ProgramCounter} := a \)
    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 system code interpreter
        with \( \text{ReferenceTime} := s \) and \( \text{ProgramCounter} := a \)
    else
        YieldToTask := true
    end if
end while
if ThreadSet ≠ ∅ then
    if there is a task \( t \neq \text{idle} \) in ThreadSet then
        RunningTask := t
    else
        RunningTask := idle
    end if
else
    invoke the task scheduler (Algorithm 3) if present
end if
```

**Data Structures.** We need the following data structures for the integrated implementation. A system program consists of ports, drivers, tasks, triggers, and timeouts, as well as system code. A configuration of a system program consists of a system program, a trigger queue, a reference time, a running task, and a system code program counter.

**Implementation.** Algorithm 6 implements the integrated E and S machine. 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 system code, prior to executing any other system code; and (2) before (3): enabled triggers may invoke system code that releases tasks, which require scheduling service from system code. Note that the integrated E and S machine may also use a task scheduler (Algorithm 3) 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 system code interpreter, which can execute any E and S code. The interpreter implements a straightforward merge of the E and S code interpreters. As a consequence, system code is in fact more general than interacting E and S code. For example, a thread written in system code, 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. System code is an efficient
representation of interacting E and S code but generating system code may still benefit from keeping the logical difference of E and S code in mind.

5. 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 board was already used to implement a Giotto-based flight controller [13] for the helicopter. The implementation is a patch of the custom-designed real-time operating system HelyOS [17] written in Oberon [19]. 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 present pseudo-code descriptions of the Oberon code that embeds the integrated E and S machine (Algorithm 6) and the system code interpreter into the microkernel.

Data Structures. We use the following data structures. 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 becomes the preempted task when an event interrupt occurs that pre-empts the running task. 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.

Implementation. At system startup, the bootstrap program (Algorithm 7) initializes the microkernel as follows: first, the initial task instance of the special idle task (Algorithm 8) is inserted into the task set. Then the trigger queue, thread set, and system program are initialized to be empty. The auxiliary system code and run variables are used by the idle task to receive system code from the host computer. Next, a HelyOS I/O handler is bound to an input and an output interrupt. We maintain two cyclic buffers that decouple the microkernel from the hardware: (1) an input buffer that is filled by the I/O handler when data is received from the host computer (on a serial link); and (2) an output buffer that is emptied by the I/O handler, which sends the content to the host computer. The next step of the bootstrap program is to bind the event interrupt handler (Algorithm 9) that invokes the integrated E and S machine to the system clock interrupt. We have not yet bound the event interrupt handler to other interrupts although this is possible. Finally, all interrupts are enabled and the idle task is invoked.

The idle task only returns when a shutdown command from the host computer was received. Then all interrupts are disabled and the system is shutdown. Note that during system operation the I/O interrupts remain enabled even when the microkernel is running achieving nanosecond-latency I/O.

The idle task 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. Algorithm 8 shows only two commands: (1) the switchcode command announces system code sent from the host computer; and (2) the shutdown command terminates the idle task and shuts down the system. The host computer sends system code along with the switchcode command. The idle task receives the code, and stores it if it passes an integrity check of opcodes and arguments. In Section 7 we discuss various other integrity checks as well as more potential commands for the microkernel. The idle task switches the system program executed by the microkernel to the received system code 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 (Algorithm 9) 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 system code 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.

If the next running task is different from the preempted task, the preempted task instance in the task set is updated.

Algorithm 7 The Bootstrap Program

| disable all interrupts |
| add (idle, InitialContext(idle)) to TaskSet |
| TriggerQueue := ⊥; ThreadSet := ø |
| SystemProgram := ⊥; SystemCode := ⊥ |
| Run := true |
| bind HelyOS I/O handler to I/O interrupts |
| bind the event interrupt handler (Algorithm 9) to system clock interrupt |
| enable all interrupts |
| disable all interrupts |
| shutdown system |

Algorithm 8 The Idle Task

| while Run do |
| receive Command from host |
| if Command = switchcode then |
| receive SystemCode from host |
| if checking SystemCode integrity fails then |
| SystemCode := ⊥ |
| end if |
| else if Command = shutdown then |
| Run := false |
| end if |
| if SystemCode ≠ ⊥ then |
| disable event interrupts |
| if TaskSet = { (idle, e) } then |
| SystemProgram := SystemCode; SystemCode := ⊥ |
| TriggerQueue := ((true, 0, ø)); ThreadSet := ø |
| end if |
| enable events interrupts |
| end if |
| send logging information to host |
| end while |
with the current processor context. Then, if the next running task has previously been preempted, its processor context is retrieved from the task set, the registers are restored, and the handler returns from the interrupt. Otherwise, if the next running task has not been dispatched before, the processor context variables are initialized for the task and an initial task instance is created in the task set. Then, the stack pointers are set according to the processor context variables before the handler leaves the interrupt by invoking the task. When the task completes, the processor returns to the event interrupt handler with an empty stack where the event interrupts are disabled before its task instance is removed from the task set. This part of the event interrupt handler is in fact not handling a processor interrupt anymore. Next, the event loop of the integrated machine is invoked to execute system code and to determine the next running task. Once the machine is finished, the completion handler (Algorithm 10) is invoked to dispatch the task. The completion handler is similar to the event interrupt handler except that context switching is explicit rather than upon returning from a processor interrupt.

**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 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 system code 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 (Algorithm 9) 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 of the registers. Otherwise, the registers are saved and then restored for the next running task.

**Algorithm 9** The Event Interrupt Handler

```
with registers in ProcessorContext
disable event interrupts
PreemptedTask := RunningTask
invoke the event loop (Algorithm 6)
if RunningTask ≠ PreemptedTask then
  update (PreemptedTask, ProcessorContext) in TaskSet
  if there is a RunningTask instance in TaskSet then
    get (RunningTask, ProcessorContext) from TaskSet
    switch context and enable event interrupts
    // never returns here
  else
    ProcessorContext := InitialContext(RunningTask)
    add (RunningTask, ProcessorContext) to TaskSet
    invoke the completion handler (Algorithm 10)
    // never returns here
  end if
end if
restore registers from ProcessorContext
return from interrupt with enabled event interrupts
```

**Algorithm 10** 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 (Algorithm 6)
  invoke the completion handler (Algorithm 10)
  // never returns here
end if
```

**Figure 17:** Context Switching on an Oscilloscope
Figure 17 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 system code needs to be executed. Here, system code 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 handle a timer interrupt without the optimization is 4.31μs. The first pulse of (B) shows the time (0.43μs) it takes to set the 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.

6. MICROKERN E 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.

**System 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 system code (microkernel) 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.

Figure 18 shows the system code 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 larger than the (EDF) S code because each task is dispatched 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.** Figure 19 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.

**Scheduling Overhead.** Figure 20 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 [20] 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.** Figure 21 shows the total system overhead in terms of system code 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.** Figure 22 shows the CPU utilization with different scheduling methods. 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).

7. **CAPABILITIES**

We have presented a new software system architecture for the implementation of hard real-time applications. The core of the system is a programmable microkernel that executes system-level functionality written in system code, which consists of E and S code. Semantic structure and predictability are the key properties of system code 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. System code is dynamic in the sense that it can be replaced, modified, extended, and communicated at run-time. Modifying an E code portion of system code changes the reactive behavior of the system while modifying the S code part changes the scheduling scheme. The semantic structure of system code enables the analysis and composition of real-time programs on the system level. System code does not necessarily replace traditional real-time scheduling technology. Partial system code can be complemented at run-time, e.g., by a real-time scheduler that either executes application tasks not handled by system code, or generates the missing system code on-the-fly. Predictability and composability of system code enables portability and mobility of real-time programs.

**Analyzing System Code.** System code is amenable to program and schedulability analysis. There are at least two interesting problems that involve checking time safety: (1) is some given E code time-safe, i.e., do all tasks released by the E code 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 time safety 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 time safety checking using control-sensitive program analysis techniques. Note that a control-insensitive check is a conservative approximation, which may fail on a time-safe program because the check considers program paths that are actually never taken.

**Composing System Code.** An important feature of system code 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. Note that the time-triggered architecture [15] already offers a similar compositional strategy to provide each S code component is already time-sharing.

**Partial System Code.** If the microkernel has a default task scheduler, then it is not necessary that system code describes all behaviors of a real-time program. In fact, the microkernel can generate missing system code 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 system code 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.

System code 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 [14], 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 System Code.** 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 system code enable portability and mobility. For example, environment-triggered system code [9] whose triggers only refer to events such as the system clock tick or external signals is portable code as long as time safety can be guaranteed. It is also mobile code because system code is represented hardware-independently as byte code with symbolic references to functional code.

8. REFERENCES


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


## Table 1: Tasks with long execution times

<table>
<thead>
<tr>
<th>Tasks (#)</th>
<th>Mode</th>
<th>System code (µs)</th>
<th>U (%)</th>
<th>P (#)</th>
<th>S code (µs)</th>
<th>E code (µs)</th>
<th>EDF (µs)</th>
<th>E+S code (µs)</th>
<th>S code (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>EDF</td>
<td>12.207 1.356</td>
<td>3.087</td>
<td>0.350</td>
<td>1.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>20.888 2.170</td>
<td>4.065</td>
<td>0.350</td>
<td>1.428</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>20.888 2.170</td>
<td>4.065</td>
<td>0.350</td>
<td>1.428</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>12,478 1.356</td>
<td>3.083</td>
<td>0.352</td>
<td>1.619</td>
<td>0.493</td>
<td>0.352</td>
<td>1.619</td>
<td>0.493</td>
</tr>
<tr>
<td>10</td>
<td>EDF</td>
<td>21.159 1.356</td>
<td>4.730</td>
<td>0.350</td>
<td>1.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>27.127 2.170</td>
<td>4.682</td>
<td>0.350</td>
<td>1.595</td>
<td>0.958</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>27.127 2.170</td>
<td>4.682</td>
<td>0.350</td>
<td>1.595</td>
<td>0.958</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>16,819 1.356</td>
<td>3.988</td>
<td>0.351</td>
<td>1.843</td>
<td>0.895</td>
<td>0.351</td>
<td>1.843</td>
<td>0.895</td>
</tr>
<tr>
<td>50</td>
<td>EDF</td>
<td>126.680 1.356</td>
<td>17.299</td>
<td>0.417</td>
<td>1.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>113.390 1.899</td>
<td>6.799</td>
<td>0.334</td>
<td>0.215</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>116.100 1.899</td>
<td>6.816</td>
<td>0.333</td>
<td>0.213</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>87.348 1.356</td>
<td>7.156</td>
<td>0.384</td>
<td>1.329</td>
<td>4.787</td>
<td>0.384</td>
<td>1.329</td>
<td>4.787</td>
</tr>
<tr>
<td>100</td>
<td>EDF</td>
<td>281.850 1.356</td>
<td>36.765</td>
<td>0.601</td>
<td>2.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>247.120 1.899</td>
<td>7.584</td>
<td>0.366</td>
<td>0.222</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>251.740 1.899</td>
<td>7.591</td>
<td>0.367</td>
<td>0.223</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>174.420 1.356</td>
<td>8.590</td>
<td>0.400</td>
<td>4.395</td>
<td>9.730</td>
<td>0.400</td>
<td>4.395</td>
<td>9.730</td>
</tr>
</tbody>
</table>

## Table 2: Tasks with medium execution times

<table>
<thead>
<tr>
<th>Tasks (#)</th>
<th>Mode</th>
<th>System code (µs)</th>
<th>U (%)</th>
<th>P (#)</th>
<th>S code (µs)</th>
<th>E code (µs)</th>
<th>EDF (µs)</th>
<th>E+S code (µs)</th>
<th>S code (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>EDF</td>
<td>12.207 1.356</td>
<td>3.091</td>
<td>0.352</td>
<td>1.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>20.888 2.170</td>
<td>4.063</td>
<td>0.350</td>
<td>1.436</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>20.888 2.170</td>
<td>4.063</td>
<td>0.350</td>
<td>1.436</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>12,478 1.356</td>
<td>3.084</td>
<td>0.351</td>
<td>1.623</td>
<td>0.493</td>
<td>0.351</td>
<td>1.623</td>
<td>0.493</td>
</tr>
<tr>
<td>10</td>
<td>EDF</td>
<td>21.159 1.356</td>
<td>4.730</td>
<td>0.350</td>
<td>1.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>27.127 2.170</td>
<td>4.677</td>
<td>0.351</td>
<td>1.595</td>
<td>0.958</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>27.127 2.170</td>
<td>4.677</td>
<td>0.351</td>
<td>1.595</td>
<td>0.958</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>16,819 1.356</td>
<td>3.988</td>
<td>0.351</td>
<td>1.843</td>
<td>0.895</td>
<td>0.351</td>
<td>1.843</td>
<td>0.895</td>
</tr>
<tr>
<td>50</td>
<td>EDF</td>
<td>131.020 1.356</td>
<td>16.902</td>
<td>0.234</td>
<td>1.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>116.370 1.899</td>
<td>6.742</td>
<td>0.200</td>
<td>0.216</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>118.270 1.899</td>
<td>6.754</td>
<td>0.200</td>
<td>0.216</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>87.348 1.356</td>
<td>7.025</td>
<td>0.200</td>
<td>0.329</td>
<td>4.787</td>
<td>0.200</td>
<td>0.329</td>
<td>4.787</td>
</tr>
<tr>
<td>100</td>
<td>EDF</td>
<td>300.560 1.356</td>
<td>34.913</td>
<td>0.349</td>
<td>1.043</td>
<td>0.844</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF sc</td>
<td>242.510 1.899</td>
<td>7.580</td>
<td>0.200</td>
<td>0.216</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>EDF spo</td>
<td>243.600 1.899</td>
<td>7.580</td>
<td>0.200</td>
<td>0.216</td>
<td>0.547</td>
<td>1.356</td>
<td>0.516</td>
<td>0.350</td>
</tr>
<tr>
<td></td>
<td>RM sc</td>
<td>177.140 1.356</td>
<td>8.535</td>
<td>0.201</td>
<td>3.979</td>
<td>9.803</td>
<td>0.201</td>
<td>3.979</td>
<td>9.803</td>
</tr>
</tbody>
</table>

## Table 3: Tasks with short execution times