The Embedded Machine: Predictable, Portable Real-Time Code*

Thomas A. Henzinger
EECS, University of California, Berkeley
tah@eecs.berkeley.edu

Christoph M. Kirsch
EECS, University of California, Berkeley
cm@eecs.berkeley.edu

ABSTRACT

The Embedded Machine is a virtual machine that mediates in real time the interaction between software processes and physical processes. It separates the compilation of embedded programs into two phases. The first, platform-independent compiler phase generates E code (code executed by the Embedded Machine), which supervises the timing — not the scheduling— of application tasks relative to external events, such as clock ticks and sensor interrupts. E code is portable and exhibits, given an input behavior, predictable (i.e., deterministic) timing and output behavior. The second, platform-dependent compiler phase checks the *time safety* of the E code, that is, whether platform performance (determined by the hardware) and platform utilization (determined by the scheduler of the operating system) enable its timely execution. We have used the Embedded Machine to compile and execute high-performance control applications written in Giotto, such as the flight control system of an autonomous model helicopter.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

Real Time, Virtual Machine

1. INTRODUCTION

We define a real-time execution model, called the Embedded Machine (E machine, for short), which provides a portable target for the compilation of languages with hard real-time constraints, such as Giotto [8]. E code (the code executed by the E machine) has strong theoretical properties, in particular, its timing and behavior are predictable. These benefits do not come at undue cost in performance. We have demonstrated this by using E code to reimplement the flight control system of an autonomous model helicopter [10].

From Platform-Centric to Requirements-Centric Real-Time Programming

In embedded systems, there are two time lines. The interaction of software processes with physical processes (sensors, actuators, clocks) happens in *environment time*. Application requirements are specified in environment time, e.g., "the actuator is set within 2 clock ticks of a sensor interrupt." On the other hand, the execution of software processes on a specific platform happens in *platform time*. By *platform* we mean the combination of hardware and real-time operating system (RTOS). Issues of platform performance, such as worst-case execution times (WCETs), and platform utilization, such as distribution and scheduling, must be addressed in terms of CPU time. The art of embedded programming is to reconcile the two time lines.

The E machine proposes a paradigm shift in real-time programming: it permits the programmer to think exclusively in terms of environment time ("reactivity"), and shifts the burden of reconciliation with platform time to the compiler ("schedulability"). This paradigm shift is in line with the steady move towards higher-level programming abstractions. In fact, the E machine treats platform time as a resource in the way in which most high-level languages treat memory: the programmer assumes there is enough of it; the compiler makes sure there is enough of it (or fails to compile); the runtime system throws an exception in case the compiler was wrong (usually due to incorrect assumptions about the platform and possible contingencies).

Programming in terms of environment time avoids the two central drawbacks of conventional embedded code: nonportability and unpredictability. The former is immediate: conventional embedded code is intrinsically platform-dependent, because it directly or indirectly (say, through priorities) refers to platform time; E code is platform-independent, because it refers only to environment time. Predictability, both in timing and functionality, is a less immediate but powerful consequence of programming in terms of environment time. By *task* we mean a software process (or a fragment thereof) without internal synchronization points [13]. Suppose that the inputs of a task t become

*This research was supported in part by the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, the AFOSR MURI grant F49620-00-1-0327, and the NSF ITR grant CCR-0085949.

PLDI'02, June 17-19, 2002, Berlin, Germany.

available at time x (say, through a sensor interrupt), and its outputs are required at time y (say, as an actuator setting). The programmer and, in turn, the E machine is concerned only with these two times: at environment time x, the inputs are provided to t and the task is turned over to the platform, namely, the RTOS; at environment time y, the outputs of t are read and given to the actuator. The programmer may assume that the task has indeed completed at time y; otherwise the compiler (or, as last resort, the runtime system) will complain. However, the programmer cannot know exactly when in the interval $[x, y]$ the task completes; in fact, she cannot even read the outputs of t "as soon as they become available," as this would introduce an instant of platform time into the program. The strict adherence to environment time allows us to design E code without race conditions: for two concurrent tasks, it does not matter which task completes first, as long as each task completes before its outputs are read.

A computation of the E machine is *time-safe* if each task completes before its outputs are read. Time safety depends, of course, on the platform (performance, distribution, scheduling). A good compiler ensures time safety; in addition, the runtime system monitors time safety. For E code that refers only to environment time, time safety implies environment determinedness. A computation of the E machine is *environment-determined* if the inputs from the environment processes (e.g., the sensor readings) uniquely determine the outputs of the software processes (e.g., the actuator settings). While time safety captures timing predictability (the actuators are written at predictable times), environment determinedness captures, in addition, value predictability (the actuators are given predictable values).

Based on these principles, we defined the language Giotto for high-performance control applications [8]. Giotto supports high-level structuring principles for modern control systems, such as periodic task invocation and multiple control modes. In compiling Giotto, we have found it useful to have an intermediate language, with none of the high-level concepts but the same platform-independent semantics for mediating between the physical environment (typically, sensors and actuators) and software tasks (typically, control law computations). This intermediate language, which has evolved into E code, offers several benefits. First, it separates the platform-independent from the platform-dependent parts of the Giotto compiler, thus enabling reuse. The platform-independent part of the compiler generates E code from a Giotto program; its main purpose is to specify the timing of all interactions among software tasks, and between software tasks and the environment. The platform-dependent part of the compiler checks the time safety of the E code for a given platform with known WCETs and known scheduling scheme (a more ambitious compiler may attempt to synthesize a scheduling scheme that guarantees time safety). Second, E code permits the dynamic implementation of Giotto: code can be patched at runtime, and whenever the controller switches mode, new code can be linked at runtime.

While E code has evolved from compiling Giotto, we have found it of considerable independent interest, as it illustrates the causalities between the underlying semantic principles, and ways to generalize them. One limitation of Giotto, for some applications, is its time-triggered nature: all Giotto time instants are ticks of an external clock, which, in high-performance control applications, minimizes jitter. By contrast, E code may refer to environment events that are not clock ticks, such as sensor interrupts. Also, Giotto cannot refer to the completion time of tasks. For E code, it is a small and natural step to consider inputs not only from the environment processes, but inputs from both the environment and the software processes. Then the completion of a software task becomes an input event, which may influence the execution of E code. As this generalization introduces E code references to platform time, environment determinedness is sacrificed, and replaced by the weaker — but symmetric— notion of *input determinedness*: the inputs from the environment processes (e.g., the sensor readings) and from the software processes (e.g., the scheduling scheme) together uniquely determine the outputs of the software processes (e.g., the actuator settings). It is the study of concepts such as time safety and environment/input determinedness that elevates the E machine from an intermediate language for compiling embedded code to a framework for evaluating embedded programming paradigms.

An Overview of the E Machine

The E machine is a mediator between physical processes and software processes: it interpretes E code, which supervises the execution of software processes (written in, say, C) in relation to physical events, such as clock ticks, and physical states, such as sensor values. The E machine has two input interfaces and one output interface.

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.

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.

Logically, the E machine does not need to distinguish between environment and task ports; they are both input ports, while driver ports are output ports. A change of value at an input port 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. E code, in turn, supervises the execution of both tasks and drivers.

Tasks A task is a piece of application-level code which typically implements a computation activity. When invoked with arguments, a task computes and writes the results to task ports. The execution of a task requires a positive amount of real time, i.e., the results cannot be observed until at least one input event happens (e.g., a clock tick, or a signal that indicates the completion of the task). A task can be preempted but has no internal synchronization points.

Drivers 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. The execution of a driver satisfies the *synchrony assumption* [6], that it can be performed 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.

To protect IP, both tasks and drivers may be given as binary executables; E code refers to tasks and drivers only through symbolic references. E code is interpreted system-level code that supervises the execution of a given set of tasks and drivers relative to input events. For this purpose, E code has essentially three instructions.

Call driver The `call` instruction initiates the execution of a driver. As the driver is synchronous system-level code, the E machine waits until the driver is finished before interpreting the next instruction of E code.

Schedule task The `schedule` instruction hands a task to the operating system. Typically, the task is put into a *ready queue*, from which the scheduler of the operating system chooses tasks for execution according to some scheduling scheme. The 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. Runtime real-time violations are due to a combination of fast physical environment, slow hardware, and inefficient scheduling; they cannot be blamed on any single one of these factors. However, they can be ruled out by a compiler that checks time safety.

Future E code The `future` instruction marks a block of E code for execution at some future time. It has two parameters: a trigger, which is a predicate that is evaluated with every input event; and the address of a block of E code, which is executed as soon as the trigger evaluates to true. In order to handle multiple active triggers, the `future` instruction puts the trigger-address pair into a *trigger queue*. With each input event, all triggers in the trigger queue are evaluated, and the first one to evaluate to true determines the next actions of the E machine.

The E machine is a virtual machine. In an actual implementation of the E machine, E code need not be interpreted, but may be compiled into, say, C code, or even silicon. The difference between E code and equivalent C code lies in the programming discipline imposed by E code. In particular, the fact that E code relates to time strictly through the trigger queue makes time-safety analysis possible. Moreover, the overhead incurred by E code, rather than optimized C code, is minimal, because for code that supervises the timing and interaction of tasks, correctness (i.e., predictability) dwarfs performance as the critical design issue, even in high-performance control applications. For example, we have found that in helicopter control, the entire program contains less than 400 instructions of E code.

A Simple Example with Two Periodic Tasks

We present a highly simplified version of the control program for a model helicopter built at ETH Zürich [10]. Consider the helicopter in hover mode m. There are two tasks, both given in native code, possibly autogenerated from Matlab/Simulink models: the control task t_1, and the navigation task t_2. 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 data communication requires three drivers: a sensor driver d_s, which provides the GPS data to the navigation task; a connection driver d_i, which provides the result of the navigation task to the control task; and an actuator driver d_a, 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. Figure 1 shows the topology of the program: we denote ports by bullets, tasks by rectangles, drivers by diamonds, and triggers by circles. Here is a Giotto description of the program timing:

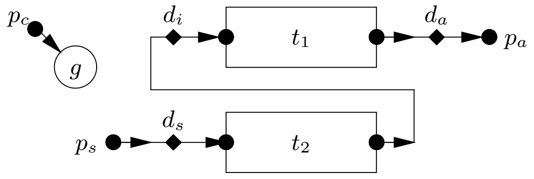

Figure 1: An example with two periodic tasks

```
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); }
```

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. Here is the E code generated by the Giotto compiler:

```
a_1:  call(d_a)          a_2:  call(d_s)
      call(d_s)                schedule(t_2)
      call(d_i)                future(g, a_1)
      schedule(t_1)
      schedule(t_2)
      future(g, a_2)
```

The E code consists of two blocks. The block at address a_1 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 hands the two tasks to the scheduler, and finally activates a trigger g with address a_2. When the block finishes, the trigger queue of the E machine contains the trigger g bound to address a_2, and the ready queue of the scheduler contains two tasks, t_1 and t_2. 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 scheduler takes over and assigns CPU time to the tasks in the ready queue according to some scheduling scheme. When a task completes, the scheduler removes it from the ready queue.

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_2 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_2 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, and is therefore no longer in the ready queue when the a_2 block is executed. In this case, the E code proceeds to put another invocation of t_2 into the ready queue, and to trigger the a_1 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., in the ready queue. In this case, the attempt by the sensor driver to overwrite a port read by t_2 causes a runtime 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, i.e., the ready queue must be empty. 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 time safety 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 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 1. 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 operating system. With this information, time safety for E code produced from Giotto can be checked. However, for arbitrary E code and platforms, such a check may be difficult, and the programmer may have to rely on runtime exception handling.

The time-safe executions of the E code example have an important property: assuming the two tasks compute deterministic results, for given sensor values that are read at the input port p_s at times 0, 10, 20, ... ms, the actuator values that are written at the output port p_a at these times are determined, i.e., independent of the scheduling scheme. This is because each invocation of the control task t_1 at $20n$ ms operates on an argument provided by the invocation of the navigation task t_2 at $20n - 10$ ms, whether or not the subsequent invocation of t_2, at $20n$ ms, has completed before the control task obtains the CPU. Time safety, therefore, ensures not only deterministic output *timing*, but also deterministic output *values*; it guarantees predictable, reproducible real-time code. This is made precise in Section 3.

The helicopter may change mode, say, from hover to descend, and in doing so, apply a different control law. In this case, the control task t_1 needs to be replaced by another task t_1'. In Section 4, we show how to implement different modes of operation using E code with control flow instructions, and how E code can be changed dynamically, at runtime, still guaranteeing determinism if no time-safety violations occur. This capability enables the real-time programming of embedded devices that upload code on demand, of code that migrates between hosts, and of code patches.

Let us summarize how the programming of the helicopter using Giotto and the implementation using E code differs from conventional real-time software design. All executions of E code happen at predefined instants of environment time, as specified by the control model and, therefore, the Giotto program: the sensor is read every 10 ms, the result of the control task is written to the actuator port every 20 ms, etc. The compiler, by checking time safety, ensures that the program can be executed; that is, the compiler matches environment time against platform time. If time safety holds, then deterministic timing and output behavior is guaranteed. Otherwise, platform performance (WCETs) or platform utilization (scheduling) must be improved. Recompilation supports code reuse on upgraded and different platforms. Conventional real-time software design proceeds in the opposite direction: the programmer's model is the platform (e.g., priority-preemptive scheduling). For example, the actuator ports are typically written whenever the control task completes, which is an instant of platform time. Then, code validation is necessary to gain confidence that the application requirements are met (e.g., that the actuator port is updated at least every 20 ms), and that the output jitter is acceptable. Code validation, however, is usually difficult, first, because the code exhibits nondeterministic timing and output behavior, and second, because the application requirements are, unlike time safety, nonuniform. If the application requirements are not satisfied, platform performance or utilization needs to be improved. So programming "the platform" does not necessarily guarantee a better success rate, but at the same time makes platform upgrades and code reuse cumbersome.

2. DEFINITION OF THE E MACHINE

The E machine mediates the timing and interaction between environment and software processes. The software processes fall into three categories: drivers, tasks, and triggers. The processes communicate via *ports*. Given a set P of ports, a P *state* is a function that maps each port in P to a value. The set P is partitioned into three disjoint sets: a set P_E of *environment ports*, a set P_T of *task ports*, and a set P_D of *driver ports*. The read/write access of processes to ports is as follows:

	Environment Ports	Task Ports	Driver Ports
Environment	RW	–	R
Tasks	–	RW	R
Drivers	R	R	RW
Triggers	R	R	R
Input triggers	R	R	–
Environment triggers	R	–	–

The environment, task, and driver ports are updated by the physical environment, by tasks, and by drivers, respectively. All information between the environment and the tasks flows through drivers: environment ports cannot be read by tasks, and task ports cannot be read by the environment. For example, a driver may read an environment port, such as a sensor or a clock, and load the value into a driver port that is read by a task; another driver may read a task port and load the value into a driver port, such as an actuator, which is read by the environment. An *event* is a change of value at a port, say, at a sensor p_s, which is observed by the E machine through an interrupt. Such an *event interrupt* can be characterized by a predicate, namely, $p'_s \neq p_s$, where p'_s refers to the current sensor reading, and p_s refers to the most recent previous sensor reading.

Definition 1. A *program* of the embedded machine consists of (1) a set P of program ports, (2) a set of drivers, a set of tasks, and a set of triggers, and (3) a set of addresses, and for each address, a finite sequence of instructions.

The instructions, discussed below, call drivers, apply scheduling services of the operating system to tasks, and handle interrupts through triggers. We do not prescribe any specific control flow instructions, but rather view part (3) — the E code— of a program abstractly as a set of blocks, each with an address and a finite sequence of instructions. We use a function $Next(a)$ that returns for an instruction at address a the address of the next instruction; if there is no next instruction, then the function returns bottom. 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. The instructions of E code do not manipulate data; all data is handled by drivers, tasks, and triggers, which can be implemented in an arbitrary programming language, such as C. Abstractly, drivers and tasks are functions from ports to ports, and triggers are boolean functions (i.e., predicates) on ports.

Definition 2. A *driver* d consists of (1) a set $P[d] \subseteq P_D$ of driver ports, and a set $I[d] \subseteq (P_E \cup P_T)$ of read environment and task ports, and (2) a function $f[d]$ from the $P[d] \cup I[d]$ states to the $P[d]$ states. A *task* t consists of (1) a set $P[t] \subseteq P_T$ of task ports, and a set $I[t] \subseteq P_D$ of read driver ports, and (2) a function $f[t]$ from the $P[t] \cup I[t]$ states to the $P[t]$ states.

A driver computes on driver ports and may read from environment and task ports; a task computes on task ports and may read from driver ports. Communication to and from a task, like communication to and from the environment, is only possible through drivers. There is a fundamental difference between drivers and tasks. A driver is nonpreemptable, atomic, single-threaded code; a task is single-threaded code that is operationally preemptable but logically atomic, without internal synchronization points. Logically, a driver is assumed to execute instantaneously in zero time whereas the execution of a task takes time. Computation in zero time is called *synchronous computation*; computation that takes time is called *scheduled computation*. Operationally, synchronous computation is performed in kernel context with event interrupts disabled. The WCET of synchronous computation (i.e., drivers) must be included in the administrative overhead for an accurate schedulability analysis. Scheduled computation happens in user context with event interrupts enabled. In order to validate the real-time behavior of E code through schedulability analysis, it is necessary to know the WCETs of scheduled computation (i.e., tasks).

Definition 3. A *trigger* g consists of (1) a set $P[g] \subseteq P$ of monitored ports, and (2) a predicate $p[g]$ on pairs of $P[g]$ states, which evaluates to true or false over each pair (s, s') of $P[g]$ states s and s'. We require that $p[g]$ evaluates to false if $s = s'$. The trigger g is an *input trigger* if $P[g] \subseteq (P_E \cup P_T)$; an *environment trigger*, if $P[g] \subseteq P_E$.

The state s is the state of the ports at the time instant when the trigger is *activated*. The state s' is the state of the ports at the time instant when the trigger is *evaluated*. We assume that all active triggers are evaluated at least at the rate of observed events. An active trigger that evaluates to true may cause a reaction of the E machine. As drivers are executed in logical zero time, a trigger that reads driver ports can evaluate to true at the same logical time instant at which the trigger is activated; such triggers make possible synchronous reactive communication between drivers [6]. Synchronous self-triggering is not possible with input triggers, which read only environment and task ports, nor with environment triggers, which read only environment ports. A program is *input-triggered* if all triggers are input triggers; *environment-triggered*, if all triggers are environment triggers. While input-triggered programs can react to software events such as the completion of tasks, environment-triggered programs can react only to environment events. An important special case of environment-triggered programs are the *time-triggered* programs, whose triggers read only an external clock. For example, every program obtained from a Giotto source is time-triggered.

A program configuration tracks, besides the values of all ports, also the active triggers, and the tasks in the ready queue of the operating system. The active triggers are kept in a FIFO queue, called *trigger queue*, according to their activation order. An active trigger stays in the trigger queue until it evaluates to true, at which point it is removed from the queue. The tasks under OS control are kept in a set, called *task set*, as the organization of the ready queue (e.g., as a priority queue) is unknown to the E machine. A task enters the task set when it is released (i.e., handed over to the OS), and leaves the task set when it completes.

Definition 4. A *program configuration* consists of (1) a P state s', called *program state*; (2) a queue of *trigger bindings* (g, a, s), called *trigger queue*, where g is a trigger, a is an address, and s is a $P[g]$ state; and (3) a set of pairs (t, s), called *task set*, where t is a task and s is a $P[t] \cup I[t]$ state. A trigger binding (g, a, s) is *enabled* if the trigger predicate $p[g]$ evaluates to true over the pair (s, s') of $P[g]$ states. The configuration c is *input-enabling* if the trigger queue contains no enabled trigger bindings; otherwise, c is *input-disabling*.

An E machine instruction is similar to a machine code instruction. It has a unique opcode and a finite number of arguments, all of which can be represented by integers. Using integers as arguments supports the portability of E code. There are only three fundamental instructions of the E machine. The control flow effect of these three instructions is trivial: after executing any of the three instructions, the E machine proceeds to the next instruction in the program. Similar to the execution of drivers, E code interpretation happens logically in zero time: E code interpretation is synchronous computation, which takes place in kernel context.

Definition 5. An E machine *instruction* is one of the following: `call`(d), for a driver d; `schedule`(t), for a task t; or `future`(g, a), for a trigger g and an address a.

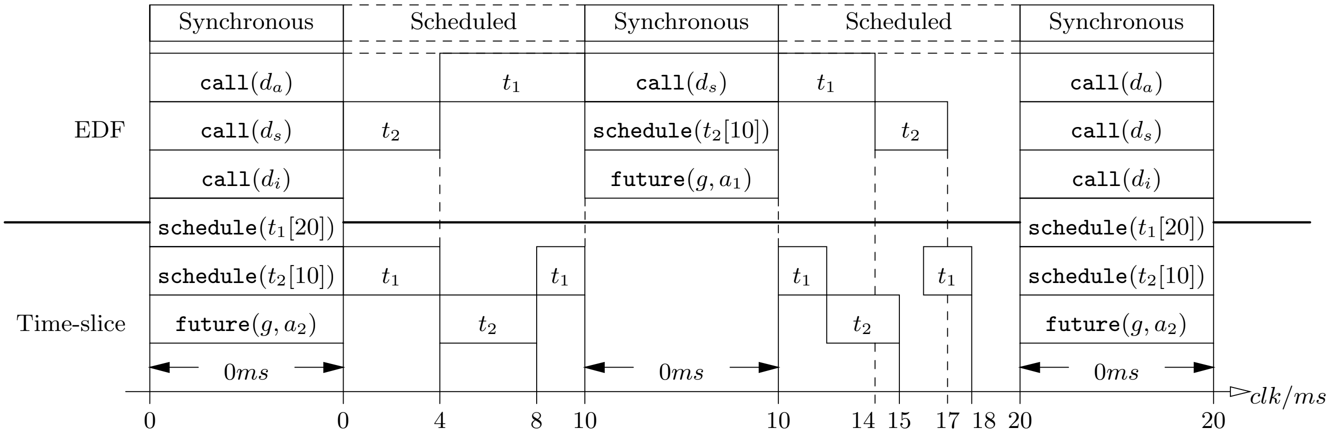

Figure 2: Earliest-deadline-first (EDF) vs. time-slice scheduling of the tasks from Figure 1

The call(d) instruction invokes the binary code of the driver d. The E machine waits until the execution of d is finished, and then proceeds to the next instruction. The schedule(t) instruction marks the binary code of the task t for execution by inserting it into the task set. Then the E machine immediately proceeds to the next instruction. The future(g, a) instruction marks the E code at address a for (possible) execution at the future time when the trigger g next evaluates to true. Operationally, the E machine appends the trigger binding (g, a, s) to the trigger queue, where s is the current program state (this is necessary for evaluating the trigger in the future), and then proceeds to the next instruction. If there is no next instruction, then the E machine gives up control of the CPU and wakes up with each event, to evaluate all triggers in the trigger queue. If the trigger binding (g, a, s) is enabled, then it is removed from the trigger queue and the E code at address a is executed. If several trigger bindings are enabled, then the corresponding blocks of E code are executed consistent with the order of the trigger queue. In other words, the future(g, a) instruction is similar to binding an interrupt handler to an interrupt, where the trigger g defines the interrupt, and the E code at address a defines the interrupt handler.

The Two-Task Example Revisited

Consider Figure 2, which shows the execution of the E code from the example in Section 1. We assume that the initial configuration consists of a state s that sets the clock p_c to -10 ms, a trigger queue with the single trigger binding (g, a_1, s), and an empty task set. When the clock p_c reaches 0 ms, the trigger g evaluates to true, the enabled trigger binding is removed from the trigger queue, and the E machine starts executing the E code at a_1. The first three call instructions execute drivers. The schedule(t_1[20]) instruction schedules task t_1 by inserting it into the task set. The term [20] is an example of an *E machine annotation*, which is handed to the scheduler along with the task. E machine annotations describe information that may be provided by the programmer, or by the compiler of a high-level language such as Giotto, to the system scheduler. For the E machine, annotations have no meaning, but the scheduler may interpret annotations. In particular, an earliest-deadline-first (EDF) scheduler interpretes the term [20] as the relative deadline of task t_1 with respect to the clock p_c. The subsequent schedule(t_2[10]) instruction inserts task t_2 into the task set. The EDF scheduler assigns a priority to t_2 which is higher than the priority of t_1, because t_2 has the earlier relative deadline of 10 ms. The last instruction, future(g, a_2), ensures that the E code at a_2 will be executed at 10 ms. The execution of the E code at a_1 is now finished. The result is a new configuration, which consists of a state s' with new values for task and driver ports, the single trigger binding (g, a_2, s') in the trigger queue, and tasks t_1 and t_2 in the task set. Finally, the E machine enables all interrupts and leaves the kernel context.

Now the scheduler, which is independent of the E machine, takes over. The upper half of Figure 2 shows the resulting timeline. The scheduler invokes task t_2, which is the highest-priority task. Task t_2 executes in user context and completes after, say, 4 ms, when it is removed from the task set. Then task t_1 executes. If task t_1 executes for, say, 10 ms, it is preempted after 6 ms by the trigger g. At 10 ms, the E machine disables event interrupts and executes the E code at a_2 by first calling driver d_s to provide a new value from the sensor port p_s to task t_2. Calling the drivers d_a or d_i at this time would result in a *time-safety violation*, because both drivers access ports that are read or written by task t_1, which has not yet completed; see Section 3. The schedule(t_2[10]) instruction inserts task t_2 into the task set. Since the deadlines of both tasks in the task set are equal, the EDF scheduler may assign a priority to t_2 that is lower or higher than the priority of t_1. In the figure, we assume the priority of t_2 to be lower than the priority of t_1. The final future(g, a_1) instruction marks the E code at a_1 for execution at 20 ms. The new E machine configuration has the single trigger binding (g, a_1, s''), where s'' is the new state, and tasks t_1 and t_2 in the task set. Now the scheduler resumes the execution of task t_1, because it has a higher priority than t_2. Task t_1 executes for another 4 ms and completes. Then task t_2 executes, say, this time only for 3 ms. Then the system is idle for the next 3 ms. At 20 ms, the E machine executes again the E code at a_1, thus closing an infinite loop of periodic invocations of both tasks.

Now consider the timeline of the scheduled computation in the lower half of Figure 2. This shows the task execution according to a time-slice scheduler instead of an EDF scheduler. The purpose of this example is to demonstrate that the E machine is independent of the scheduling scheme.

The E code is executed exactly the same way as before. The only difference is how the tasks are executed between the synchronous blocks of E code. The time-slice scheduler ignores the annotations. Each task gets a time slice of 4 ms, and tasks are executed in the order of their insertion into the task set in a round-robin fashion. Note that the second slice of task t_1 is preempted by a trigger. Also note that for given sensor readings at 0 ms and 10 ms, the value loaded by the actuator driver into p_a at 20 ms is the same no matter which scheduler is used; that is, the output behavior of the program is *deterministic*. This is the main property of E code; see Section 3.

Operational Semantics

We define the semantics of E code operationally using a pseudo-code description of the E machine. Algorithm 1 shows the main loop of the machine as it executes a given program. Initially, the trigger queue contains a single trigger binding, and the task set is empty. After entering the main loop, the machine waits for environment and task ports to change their values, i.e., for the occurrence of one or more events that may enable input triggers. While the machine waits for input events, scheduled computation may be performed, i.e., the scheduler has control of the CPU. Waiting for input events is implemented using event interrupts. The occurrence of an input event wakes up the machine, which immediately disables all event interrupts (thus it is still possible for low-level interrupts to preempt the machine, as long as they do not interfere with the triggering mechanism of the machine). After the while loop, before the machine loops back to wait for new input events, all event interrupts are enabled again and the scheduler is invoked. The main loop is executed ad infinitum.

Algorithm 1 The Embedded Machine

```
loop
  wait for input event
  disable trigger-related interrupts
  while there is an enabled trigger in $TriggerQueue$ do
    $(g, a, s) :=$
      $GetFirstEnabledTriggerBinding(TriggerQueue)$
    $TriggerQueue :=$
      $RemoveFirstEnabledTriggerBinding(TriggerQueue)$
    $ProgramCounter := a$
    invoke Algorithm 2
  end while
  enable trigger-related interrupts
  invoke system scheduler on $TaskSet$
end loop
```

The E machine runs through the while loop of Algorithm 1 as long as there are enabled trigger bindings in the trigger queue, each time executing a block of E code that is bound to an enabled trigger. The termination of the while loop is guaranteed for input-triggered programs such as E code generated from Giotto. In an input-triggered program, newly activated triggers are initially disabled and can only be enabled by environment or task activity. In more general programs with arbitrary triggers, it is possible that calling a driver enables a trigger. In this case an explicit termination proof for the while loop is necessary. Synchronous self-triggering among E code blocks corresponds to the signaling mechanism of Esterel [2], or undelayed data dependency in Lustre [7]. For synchronous reactive languages such as Esterel and Lustre, an explicit termination proof of synchronous computation is necessary (typically by ensuring the existence of finite fixed points).

Each block of E code is interpreted by Algorithm 2. In the while loop of Algorithm 2, the machine fetches the current instruction from the program, decodes and executes the instruction, and then determines the address of the next instruction. This loop terminates, because every block of E code is required to have only a finite number of instructions with sequential control flow. This is true despite the fact that it is often convenient to use control flow instructions such as absolute or conditional jumps in E code, for example, in the generation of E code from Giotto programs with mode switching; see Section 4. Later, we will also introduce additional instructions for manipulating the task set (such as terminating a task) and trigger queue (such as removing a trigger binding), but these have no fundamental impact on the operation of the E machine.

Algorithm 2 The E Code Interpreter

```
while $ProgramCounter \neq \bot$ do
  $i := GetInstruction(ProgramCounter)$
  if $\texttt{call}(d) = i$ then
    $ProgramState(P[d]) :=$
      $f[d](ProgramState(P[d] \cup I[d]))$
  else if $\texttt{schedule}(t) = i$ then
    $TaskSet := TaskSet \cup \{(t, ProgramState(P[t] \cup I[t]))\}$
  else if $\texttt{future}(g, a) = i$ then
    $TriggerQueue :=$
      $TriggerQueue \circ (g, a, ProgramState(P[g]))$
  end if
  $ProgramCounter := Next(ProgramCounter)$
end while
```

The execution of an E machine program yields a trace. From one configuration of a trace to the next, there is either an environment event, or a software event (i.e., the completion of a task), or the E machine executes one block of E code. The third possibility —E code execution— has precedence as long as there are blocks of E code with enabled triggers.

Definition 6. A *program trace* is an infinite sequence of configurations such that for any two adjacent configurations c and c', one of the following holds:

Environment event c is input-enabling, and c' differs from c in the values of environment ports.

Task completion c is input-enabling, and c' differs from c in the values of the task ports $P[t]$ for some task t, and in the task set: the task set of c contains a pair (t, s), for some state s, the task set of c' results from c by removing the pair (t, s), and the values of the task ports $P[t]$ in c' result from applying the function $f[t]$ to state s.

E code execution c is input-disabling, and c' differs from c in the values of driver ports, in the trigger queue, and in the task set: if (g, a, s) is the first enabled trigger binding in the trigger queue of c, then c' is obtained from c by first removing (g, a, s) from the trigger queue, and then executing Algorithm 2 with program counter a. In this case, we say that the E code at address a is executed at configuration c.

The *input part* of a program trace is the projection of the trace to values for environment and task ports; the *output part* is the projection to values for driver ports; the *environment part* is the projection to values for environment ports.

3. PROPERTIES OF E CODE

The benefits of using E code are due to its strong theoretical properties: time safety is a condition which is satisfied if all real-time requirements are met on a platform, and determinism is a consequence which ensures that the output behavior of E code is predictable.

Time Safety

A compiler that generates E code is not satisfied with every trace that may result from interpreting the code using Algorithm 2. Rather, the compiler expects sufficient performance from the platform so that the computation of a task always completes before drivers access (read or write) ports that are also accessed by the task, and before another invocation of the task is scheduled. A trace that satisfies these conditions is called time-safe, because it meets all timing requirements of the source (e.g., Giotto) program.

Definition 7. A program trace is *time-safe* if for every configuration c, if the task set of c contains the pair (t, s), then E code that is executed at configuration c must obey the following three conditions. (1) For each `call`(d) instruction, $P[d] \cap I[t] = \emptyset$; that is, no driver updates the read driver ports of t. (2) For each `call`(d) instruction, $I[d] \cap P[t] = \emptyset$; that is, no driver reads the task ports of t. (3) For each `schedule(t')` instruction, $P[t'] \cap P[t] = \emptyset$; that is, no scheduled task accesses the task ports of t.

The E machine throws a runtime exception if any of the above conditions is violated; we will discuss the exception handling below. In order to avoid runtime exceptions, it must be shown that the program is time-safe for the environment and the platform, i.e., that all program traces that can occur on the target platform in the target environment are time-safe. Proving time safety requires a schedulability analysis based on the WCETs of all drivers, tasks, and triggers, and if the program is not time-triggered, also requires assumptions about the frequency of input events. As an example, recall the Giotto program from Section 1, which, like all Giotto programs, is time-triggered. For single-CPU platforms with EDF scheduling, the Giotto compiler shows time safety in two steps. First, given the WCETs of all drivers and triggers, the compiler computes the WCETs of all E code blocks. Suppose that the a_1 and a_2 blocks have a WCET of 1 ms each, including the overhead for context switching. This leaves 18 ms CPU time per 20 ms real time for scheduled computation. Second, given the WCETs of all tasks, and having derived the relative deadlines for all task invocations from the Giotto source (relative deadline 20 for each `schedule`(t_1) instruction, and relative deadline 10 for each `schedule`(t_2)), the compiler uses an EDF schedulability test to show that all task invocations complete on time. For instance, assuming a WCET of 10 ms for t_1 and of 4 ms for t_2, the EDF schedule is feasible and achieves a theoretical CPU utilization of 100%.

In addition, as WCET assumptions may be wrong, the Giotto compiler generates E code for handling time-safety violations. Algorithm 3 shows the E code interpreter of Algorithm 2 enhanced with exception handling. For exception handling, a second argument is added to the `schedule` instruction: suppose that the task t is scheduled by the instruction `schedule`(t, e), where e is an address. The block of E code at address e is the *exception handler*, and its address is recorded in the task set. If, before t completes, the ports read by t are updated or the ports written by t are read by a driver or another task, then the E machine discards the instruction that causes the exception and jumps to address e. The task t is not terminated, but for the case that termination is desired, we add an instruction `terminate`(t) to the instruction set of the E machine, which removes t from the task set. This may be the first instruction of the exception handler. When the exception handler finishes, control flow returns to the instruction that follows the instruction that caused the exception. A single `call` or `schedule` instruction may cause multiple exceptions, e.g., because a driver may read ports from multiple tasks in the task set. We do not specify how simultaneous exceptions are prioritized; this is done by the function *ChooseException*. As runtime exceptions can occur inside an exception handler, Algorithm 3 is invoked recursively, so as to implicitly maintain a stack of return addresses.

Algorithm 3 The E Code Interpreter with Exceptions

```
while $ProgramCounter \neq \bot$ do
  $i := GetInstruction(ProgramCounter)$; $E := \emptyset$
  if $\texttt{call}(d) = i$ then
    $E := \{e \mid (t, \_, e) \in TaskSet$:
          $P[d] \cap I[t] \neq \emptyset \lor I[d] \cap P[t] \neq \emptyset\}$
    if $E = \emptyset$ then
      $ProgramState(P[d]) :=$
        $f[d](ProgramState(P[d] \cup I[d]))$
  else if $\texttt{schedule}(t, e) = i$ then
    $E := \{e' \mid (t', \_, e') \in TaskSet$: $P[t] \cap P[t'] \neq \emptyset\}$
    if $E = \emptyset$ then
      $TaskSet :=$
        $TaskSet \cup \{(t, ProgramState(P[t] \cup I[t]), e)\}$
  else if $\texttt{future}(g, a) = i$ then
    $TriggerQueue :=$
      $TriggerQueue \circ (g, a, ProgramState(P[g]))$
  end if
  while $E \neq \emptyset$ do
    $(ProgramCounter, E) := ChooseException(E)$
    invoke Algorithm 3
  end while
  $ProgramCounter := Next(ProgramCounter)$
end while
```

In the case of Giotto, the compiler generates an exception handler for each task. If a runtime exception is caused by an instruction that has a conflict with task t in the task set, then the associated exception handler terminates t and restores the most recent valid values to the task ports of t, that is, the values that preceded the terminated invocation of the task. This requires additional ports and drivers. In our example, the new driver d_j, for $j = 1, 2$, stores the result of task t_j in a new driver port p_j. In case of a runtime exception involving task t_j, the exception handler, at address e_j, terminates t_j and calls the new driver d'_j, which restores the value of p_j to the task port of t_j before the program proceeds. The generated E code below produces exactly the same output behavior for time-safe traces as the original E code from Section 1:

```
$a_1$:  call($d_1$)              $a_2$:  call($d_2$)
        call($d_2$)                      call($d_s$)
        call($d_a$)                      schedule($t_2, e_2$)
        call($d_s$)                      future($g, a_1$)
        call($d_i$)
        schedule($t_1, e_1$)
        schedule($t_2, e_2$)
        future($g, a_2$)
$e_1$:  terminate($t_1$)         $e_2$:  terminate($t_2$)
        call($d'_1$)                     call($d'_2$)
```

However, if a time-safety violation occurs, then the new E code handles the associated exception. Suppose that the a_1 block is executed when task t_1 has not yet completed. The attempt to execute d_1 throws a runtime exception, which invokes the E code at address e_1. After t_1 is terminated, the driver d'_1 restores the task ports of t_1, and execution proceeds with the `call`(d_2) of the a_1 block.

Determinism

Figure 2 gave an example where different scheduling schemes (EDF and time slicing) lead to the same output behavior. For either scheduling scheme, the program interacts with the environment at the same constant rate of 10 ms and produces the same actuator settings. We now show that this is a general property of E machine programs, as long as the platform maintains time safety (i.e., there are no runtime exceptions). Recall that by *input* of the E machine, we refer to both environment and task ports (e.g., input-triggered means triggered by events on these ports), by *environment* we refer to the environment portion of the input, and by *output*, to the driver ports (which includes the actuators).

Definition 8. A program is *input-determined* if, whenever two time-safe program traces agree on the initial configurations and input parts, then they agree on the output parts as well. A program is *environment-determined* if, whenever two time-safe traces agree on the initial configurations and environment parts, then they agree on the output parts.

It is easy to see that all E machine programs are input-determined. However, input determinedness is a rather weak property. Input-determined programs are deterministic only with respect to a given behavior of all environment and software processes. In order to decouple environment and software, a program must be environment-determined, i.e., independent of the real-time behavior of the software processes.

FACT. *Every program is input-determined. Every environment-triggered program is environment-determined.*

For example, being environment-triggered, the program of Figure 2 is environment-determined, i.e., it produces the same actuator settings independently of the execution order of the tasks, as long as the platform maintains time safety. In fact, as Giotto is time-triggered, all E code generated from Giotto sources is environment-determined, and so is E code generated from more general sources, whose triggers monitor arbitrary environment events. On the other hand, an input-triggered program is in general not environment-determined, because it may trigger on the completion of a task execution. Thus, depending on the detailed performance (WCETs) and scheduling of the platform, such a program may exhibit very different output behaviors.

Environment determinedness crucially depends on the task model of the E machine. Suppose that tasks were allowed to communicate with each other directly, without going through drivers. In our example, this would mean that task t_1 reads the task port of t_2, instead of calling the connection driver d_i. Now consider the resulting timelines. In the case of EDF, the first invocation of t_1 would read the result of the invocation of t_2 that finishes at 4 ms. By contrast, in the case of time slicing, t_1 would start first and thus read the initial value of the task port of t_2. As a consequence, the result of the first invocation of t_1, which determines the actuator setting at 20 ms, would be different for the two scheduling schemes. Environment determinedness also depends on the instruction set of the E machine. For example, instead of having a compiler ensure time safety based on schedulability analysis, we could obtain time safety trivially by using the `terminate`(t) instruction to terminate any task t before we interact with t. All traces of a program written in this way are time safe, but the program, even if environment-triggered, would not be environment-determined, because the result of a terminated task depends on how long the task has executed.

Time Liveness

Time safety means that all *finite* trace prefixes satisfy the intended real-time constraints of E code. There still may be undesirable *infinite* traces, however, where the E machine is infinitely faster than the physical environment. This is because, even in the absence of infinite loops inside individual blocks of E code, two different blocks of E code may activate and enable each other without intervening input events. A trace without such infinite zero-time behavior is called time-live.

Definition 9. A program trace is *time-live* if for every configuration c, if c is input-disabling, then it is followed by some later configuration c' that is input-enabling.

FACT. *Every trace of an input-triggered program is time-live.*

Time liveness, therefore, is not an issue for E code generated from Giotto, whose triggers monitor only clocks, nor for E code generated from any other source whose triggers do not look at driver ports. However, E code generation from a synchronous reactive language such as Esterel may, like any Esterel compilation, require nontrivial proofs of time liveness.

The execution of a *time-live program*, i.e., a program whose traces are all time-live, may require a trigger queue of unbounded size, which is not desirable in practice. (By contrast, the size of the task set is always bounded by the number of tasks.) For example, two consecutive `future` instructions to the same block of E code may set off a process that doubles the size of the trigger queue with each input event. A similar problem may arise in the presence of control flow instructions such as for loops: while always finite, the execution of a single block of E code may take more time with each new input event, because the block may consist of a ever increasing number of instructions.

Definition 10. A program trace is *bounded time-live* if there exists an integer k such that for every configuration c, if c is input-disabling, then (1) c is followed by at most k input-disabling configurations, and (2) the trigger queue of c contains at most k entries.

Note that bounded time liveness implies time liveness. From Giotto programs, one can always generate E code that is bounded time-live. For E code generation from, say, Esterel, proving bounded time liveness is necessary to ensure that the delay of executing any synchronous reaction is shorter than the time between any two events that can trigger a synchronous reaction. For a fixed bound k, bounded time liveness can be enforced at runtime by exception handling, similar to the handling of time-safety violations. For example, a `future` instruction may cause a runtime exception if it attempts to create a new trigger binding when the trigger queue already contains k entries.

4. E CODE GENERATION AND LINKING

Generating code from standard high-level programming languages is an optimization problem to reduce the time and space requirements of the code. Generating E code is different in the sense that the time and space requirements of E code are usually negligible compared to the efficiency requirements of the task code, even on complex systems such as a helicopter flight-control system. For example, we need less than 400 instructions of E code for the ETH helicopter. However, it is essential to guarantee the correct integration of synchronous and scheduled computation —i.e., time safety— and to ensure bounded E code execution — i.e., bounded time liveness. When generating E code from Giotto, bounded time liveness can be guaranteed by construction, but ensuring time safety requires explicit proof. By contrast, when generating E code from synchronous reactive languages such as Esterel or Lustre, we need only the `call` and `future` instructions, but not the `schedule` instruction, because these languages do not explicitly support scheduled computation. Therefore, time safety is trivial, but achieving bounded time liveness may require proof [6]. Also, for code generation from synchronous reactive languages, more flexibility on the manipulation of the trigger queue may be necessary than is offered by the presented, minimal definition of the E machine. Useful additions to the instruction set of the E machine include the `cancel`(a) instruction, which removes all trigger bindings with the address a from the trigger queue, and the `future`$(true, a)$ instruction, which marks the E code at address a for execution after all currently enabled trigger bindings have been processed.

Compiling Giotto

A Giotto program consists of a functionality part and a timing part. The functionality part contains port, driver, and task declarations, which interface the Giotto program to a functionality implementation, typically written in C. The Giotto compiler generates so-called *functionality wrappers* —parameter-less procedures— for each driver and task implementation, and stores the wrappers in a table similar to a symbol table. A wrapper calls the corresponding implementation with the proper arguments. The E code, however, refers to the wrapper using only its table index, a portable integer value. From the timing part of the Giotto program the compiler generates annotated E code, where each `schedule` instruction is annotated with the relative deadline of the scheduled task.

The E code generation from multi-mode Giotto programs illustrates the use of conditional jumps. Consider the following timing part of a Giotto program with two modes, m_a (representing the helicopter in hover mode) and m_b (descend mode):

```
start $m_a$ {
  mode $m_a$() period 20 {
    actfreq 1 do $p_a(d_a)$;
    exitfreq 1 do $m_b(c_b)$;
    taskfreq 1 do $t_1(d_i)$;
    taskfreq 2 do $t_2(d_s)$; }
  mode $m_b$() period 20 {
    actfreq 1 do $p_a(d_a)$;
    exitfreq 1 do $m_a(c_a)$;
    taskfreq 1 do $t'_1(d_i)$;
    taskfreq 2 do $t_2(d_s)$; }}
```

The program begins by executing mode m_a, which is equivalent to the (single) mode m of the Giotto program from Section 1 except for the mode switch to mode m_b. A *mode switch* in Giotto has a frequency that determines at which rate an exit condition is evaluated. The exit condition c_b of mode m_a is evaluated once every 20 ms. If c_b evaluates to true, then the program switches to mode m_b, which is similar to mode m_a except that task t'_1 replaces task t_1. Task t'_1 computes a different control law on the same ports as t_1. The mode switch back to m_a evaluates the exit condition c_a also once every 20 ms. In order to express mode switching in E code, we use a conditional branch instruction `if`(c, a). The first argument c is a *condition*, which is a predicate on some ports. The second argument a is an E code address. The `if`(c, a) instruction evaluates the condition c synchronously (i.e., in logical zero time), similar to driver calls, and then either jumps to the E code at address a (if c evaluates to true), or proceeds to the next instruction (if c evaluates to false). Here is the E code that implements the above Giotto program:

```
$a_1$:  call($d_a$)                    $a_3$:  call($d_a$)
        call($d_s$)                            call($d_s$)
        call($d_i$)                            call($d_i$)
        schedule($t_2[10], e_2$)               schedule($t_2[10], e_2$)
        if($c_b, a'_3$)                        if($c_a, a'_1$)
$a'_1$: schedule($t_1[20], e_1$)       $a'_3$: schedule($t'_1[20], e'_1$)
        future($g, a_2$)                       future($g, a_4$)

$a_2$:  call($d_s$)                    $a_4$:  call($d_s$)
        schedule($t_2[10], e_2$)               schedule($t_2[10], e_2$)
        future($g, a_1$)                       future($g, a_3$)
```

The two E code blocks in the left column implement mode m_a; the two blocks on the right implement m_b. The exception handlers for the three tasks (e_1, e'_1, and e_2) are omitted. Note that, no matter which conditional branches are taken, the execution of any block terminates within a finite number of E code instructions. This concludes the first, platform-independent phase of the Giotto compiler.

The second, platform-dependent phase of the Giotto compiler performs a time-safety check for the generated E code and a given platform. For single-CPU platforms with WCET information and an EDF-based scheduling scheme, and for the simple code generation strategy illustrated in the example, the time-safety check is straightforward. For distributed platforms, complex scheduling schemes, or complex code generation strategies, this, of course, may not be the case. The code generation strategy has to find the right balance between E code and E machine annotations. An extreme choice is to generate E code that at all times maintains a singleton task set, which makes the scheduler's job trivial but E code generation difficult. The other extreme is to schedule tasks as early as possible, with precedence annotations that allow the scheduler to order task execution correctly. This moves all control over the timing of software events from the code generator to the scheduler. In other words, the compiler faces a trade-off between static (E machine) scheduling and dynamic (RTOS) scheduling. Our strategy, which schedules tasks and computes deadlines according to the "logical semantics" of the Giotto source, chooses a compromise that suggests itself for control applications. To achieve controller stability and maximal performance, it is often necessary to minimize the jitter on sensor readings and actuator updates. This is accomplished by generating separate, time-triggered blocks of E code for calling drivers that interact with the physical environment. In this way, the time-sensitive parts of a program are executed separately [12], and for these parts, platform time is statically matched, at the E code level, to environment time as closely as possible. On the other hand, for the time-insensitive parts of a program, the scheduler is given maximal flexibility.

Dynamic Linking

Software modularization is an important concept in the non-real-time world for improving software reusability and reducing software complexity. Software modularization requires the use of symbolic references in executable code rather than direct references. Resolving symbolic references at compile time and runtime is called *linking* and *dynamic linking*, respectively. Since E code uses only symbolic references, E code (and tasks, drivers) can be linked statically as well as dynamically.

For an example of static linking, the two columns of E code that are generated for the two modes of the Giotto program above can be compiled separately and then linked to a complete E code executable. For dynamic linking, we leverage the dynamic nature of the trigger queue by maintaining trigger bindings to unloaded E code in the queue. This approach requires control over the queue as provided by the `cancel`(a) instruction, which removes all trigger bindings with the address a from the trigger queue. Hence a `cancel` instruction may negate the effect of several `future` instructions. Consider the following program (annotations and exception handling are omitted), which implements the dynamic loading and linking of the two Giotto modes m_a and m_b during helicopter flight:

```
$a_1$:  cancel($a_3$)          $a_3$:  cancel($a_1$)
        call($d_a$)                    call($d_a$)
        call($d_s$)                    call($d_s$)
        call($d_i$)                    call($d_i$)
        schedule($t_1$)                schedule($t_1'$)
        schedule($t_2$)                schedule($t_2$)
        future($g, a_2$)               future($g, a_4$)
        future($h_b, a_3$)             future($h_a, a_1$)

$a_2$:  call($d_s$)            $a_4$:  call($d_s$)
        schedule($t_2$)                schedule($t_2$)
        future($g, a_1$)               future($g, a_3$)
```

Suppose that only the E code in the left column is currently loaded in the E machine. We begin by executing the a_1 block with an empty trigger queue and an empty task set. Thus the initial `cancel`(a_3) instruction has no effect. The final `future`(h_b, a_3) instruction activates a new trigger h_b and binds it to the still unknown address a_3. The trigger predicate of h_b is $(p'_c = p_c + 20) \wedge c_b$, that is, h_b becomes enabled at 20 ms if the condition c_b evaluates to true (h_a is defined analogously using condition c_a). At 10 ms, the E machine executes the a_2 block. The `future`(g, a_1) instruction appends the trigger g to the trigger queue after the trigger h_b, which is still in the queue. At 20 ms, if c_b is not true, then h_b is not enabled and thus skipped, but g is enabled, which causes the a_1 block to be executed again. Now the first `cancel`(a_3) instruction removes the h_b trigger from the queue. On the other hand, if c_b is true, then the E machine attempts to execute instead the a_3 block. As the a_3 block is not available, the E machine starts the loader and linker to retrieve it. This overhead needs to be taken into account by time-safety analysis. Once the E code is available, the E machine begins by executing the `cancel`(a_1) instruction, which removes the g trigger from the queue. The rest of the E code at a_3 and a_4 is analogous to the E code at a_1 and a_2, except that task t'_1 (whose code can also be loaded and linked dynamically) replaces task t_1.

Current E Machine Implementations

We have three implementations of the E machine. The simplest one is written in C for Lego Mindstorm robots using the open-source LegOS operating system. The interesting aspect of this implementation is that it is a kernel patch, which makes the E machine part of the kernel, rather than the highest-priority thread outside the kernel. The E machine implementation for the ETH helicopter [10] is written in Oberon using a custom-designed RTOS on a StrongARM embedded processor. The interesting feature of this implementation, besides high performance, is that tasks are implemented as subroutines rather than as coroutines [13]. In this case, the E machine is an interrupt handler bound to a real-time clock. Preemption works through reentrant interrupts. The third implementation is in C under Linux using POSIX threads and semaphores. The E machine and each task runs in its own thread. Each task thread runs at a lower priority than the E machine thread and uses a unique semaphore on which it waits until the E machine signals the semaphore. For example, upon executing a `schedule`(t) instruction, the E machine signals the semaphore of the thread that implements task t. When t completes, the thread loops back and waits on the semaphore for the next `schedule`(t) instruction. The goal of the Linux implementation is to port it to the RTOS VxWorks, which also features POSIX threads, however, with real-time guarantees. The Linux version also includes a dynamic loader and linker for the binary format of E code. Moreover, it features a distributed E machine implementation that runs interacting E machines on each host of a distributed system. The hosts communicate using the UDP protocol on BSD sockets. The distributed version is also supported by the Giotto compiler, which can generate E code separately for each E machine. The goal is again to port the implementation to a network with real-time guarantees, such as a time-triggered bus.

5. CONCLUSION

E code is predictable, portable, hard real-time code. E code is hard real-time, because it relates to environment (physical) time, rather than platform (CPU) time; this simplifies code validation at the expense of code generation. E code is predictable, because the timing and behavior of a program depends only on the external inputs; there are no internal race conditions. E code is portable, because it is independent of the platform, in particular, of the scheduler. The combination of these attributes is made possible through the notion of *time safety*: E code is time-safe if its timing requirements can be met on the chosen platform. In some cases, such as Giotto source programs, time safety can be checked statically, by the compiler; in other cases, time-safety violations are handled dynamically, by the runtime system.

The problem of real-time programming is to define abstractions that capture the interaction between physical processes and software processes. Two major research communities have approached this problem from different directions: the synchronous reactive language community has studied zero-delay synchronous computation [6], and the real-time systems community has focused on the theory of scheduled computation [4] and corresponding programming languages [3]. Both fields have had a big impact on the design of the E machine, which attempts to bring together the concepts of synchronous and scheduled computation. Synchronous computation relates software processes to physical processes by modeling software processes as instantaneous reactions to physical stimuli. The main challenge is to prove the analogue of time liveness, i.e., the existence of finite reactions [2]. However, large blocks of synchronous computation naturally exhibit delayed reactivity, which may deteriorate the determinism of the synchronous model. Real-time scheduling theory, on the other hand, relates software processes to physical processes by imposing a constraint system of release times and deadlines on the software processes. The main challenge is to prove the analogue of time safety, i.e., the existence of a feasible schedule. However, the arrangement of computation according to a schedule that depends on the behavior of both physical and software processes leads to an inherently nondeterministic model.

From the E machine perspective, the synchronous reactive language community deals with organizing a time-live trigger queue, and the real-time systems community worries about scheduling a time-safe task set. This can be seen more clearly from various intermediate languages that have been proposed for synchronous and for scheduled computation. Examples from the synchronous realm include the automata-based portable object code [9] for Esterel and Lustre, and halt point graphs [11] for Esterel. Examples from the scheduled realm include the abstract machines JAM and BEAM for code generation from the functional real-time language Erlang [1]. Each of these formalisms is richer than the E machine in some respects, as they permit more general manipulations of the trigger queue, or of the task set, or more general control flow and data handling. The E machine attempts to exploit and, at the same time, restrict the possibilities in both the synchronous and the scheduled realm by identifying a set of primitives that (1) guarantee strong determinism properties of the code and (2) are sufficiently rich to be useful in practice. An important practical question raised by the E machine asks which parts of embedded software are best modeled by synchronous computation, and which by scheduled computation. For example, E code generated from Giotto uses synchronous computation to implement the data transport (I/O) from and to the physical system, and between software processes, while time-consuming data computation is implemented as scheduled computation. (In a distributed Giotto system, data communication across networks with nonnegligible latency is also implemented as scheduled computation.)

It should be noted that Wirth already emphasized in 1977 the value of separating logical programs from physical platforms to obtain a discipline of real-time programming [12]. He suggested that the time-dependent program parts, i.e., the blocks of synchronous computation, are "few, simply structured, and without loops with an unknown number of repetitions" in order to maintain correctness independently from the execution time of the non-real-time code, i.e., the units of scheduled computation. In particular, he proposed a "ban on the notion of interrupt" unless "interrupts can be ignored in considerations about a system's computational state and can be confined to timing considerations only."

Acknowledgments. We thank Marco Sanvido for help with the helicopter implementation, Arkadeb Ghosal and Slobodan Matic for help with the Giotto compiler, and Edward Lee and Jie Liu for many valuable discussions.

6. REFERENCES

[1] J. Armstrong. The development of Erlang. *Int. Conf. Functional Programming*, pp. 196–203. ACM, 1997.

[2] G. Berry. The foundations of Esterel. In G. Plotkin and M. Tofte, eds., *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[3] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[4] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.

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

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

[7] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. IEEE*, 79:1305–1320, 1991.

[8] T. Henzinger, B. Horowitz, C. Kirsch. Giotto: A time-triggered language for embedded programming. *EMSOFT*, LNCS 2211, pp. 166–184. Springer, 2001.

[9] J. Plaice and J.-B. Saint. *The Lustre-Esterel Portable Format*. Tech. Rep., INRIA Sophia-Antipolis, 1998.

[10] M. Sanvido. *A Computer System for Model Helicopter Flight Control, Part 3: The Software Core*. Tech. Rep. 317, Inst. Computer Systems, ETH Zürich, 1999.

[11] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, J. Pulou. Efficient compilation of Esterel for real-time embedded systems. *Int. Conf. Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 2–8. ACM, 2000.

[12] N. Wirth. Toward a discipline of real-time programming. *Comm. ACM*, 20:577–583, 1977.

[13] N. Wirth. Tasks versus threads: An alternative multiprocessing paradigm, *Software: Concepts and Tools*, vol. 17, pp. 6–12. Springer, 1996.