Schedule-Carrying Code*

Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic

EECS, University of California, Berkeley

Abstract. We introduce the paradigm of *schedule-carrying code* (SCC). A hard real-time program can be executed on a given platform only if there exists a feasible schedule for the real-time tasks of the program. Traditionally, a *scheduler* determines the existence of a feasible schedule according to some scheduling strategy. With SCC, a *compiler* proves the existence of a feasible schedule by generating executable code that is attached to the program and represents its schedule. An SCC executable is a real-time program that carries its schedule as code, which is produced once and can be revalidated and executed with each use. We evaluate SCC both in theory and practice. In theory, we give two scenarios, of nonpreemptive and distributed scheduling for Giotto programs, where the generation of a feasible schedule is hard, while the validation of scheduling instructions that are attached to the programs is easy. In practice, we implement SCC and show that explicit scheduling instructions can reduce the scheduling overhead up to 35% and can provide an efficient, flexible, and verifiable means for compiling Giotto programs on complex architectures, such as the TTA.

1 Introduction

Giotto is a high-level programming language for hard real-time applications [4]. A Giotto program consists of a collection of modes, each specifying the release times and deadlines of a set of periodic tasks. In Giotto, the semantics of value propagation between tasks is defined independent of the system scheduler, and therefore deterministic: as long as the scheduler maintains all deadlines, the outputs of all task invocations are determined by the sensor inputs, and do not depend on the task ordering, distribution, or preemption mechanism of a particular RTOS. A Giotto program is executable only if there exists a schedule that meets all deadlines on a given hardware platform, with given resources and performance. An executable combination of Giotto program and platform data (primarily worst-case execution times for all tasks) is called *time-safe* [5]. The Giotto compiler must, in addition to generating code, prove time safety [6]. The proof of time safety establishes the existence of a feasible schedule, and in doing so, produces the schedule. We introduce the idea of *schedule-carrying code* (SCC): once a feasible schedule has been produced, it can be attached to the code

* This work was supported by the AFOSR MURI grant F49620-00-1-0327, the California MICRO grant 01-037, the DARPA grant F33615-C-98-3614, the MARCO grant 98-DT-660, and the NSF grants CCR-0208875, CCR-0085949, and CCR-0225610.

© Springer-Verlag Berlin Heidelberg 2003

to serve as a witness for time safety, in case the code source is untrusted, or if the code is moved to multiple targets. Thus, SCC is the paradigm of proof-carrying code [10] extended from traditional safety properties, such as memory safety, to real-time properties. SCC, however, offers an even more important capability than the reuse of time-safety proofs. If the schedule itself is provided in the form of executable instructions, properly attached to the code generated from Giotto, then SCC renders the system scheduler of the target platform obsolete. This leads to dramatic performance improvements in executing Giotto programs [7].

Let us be more precise. We define two virtual execution engines called the *E(mbedded) machine*, and the *S(cheduling) machine*. The E machine executes E code generated from a Giotto program [5]. E code is reactive code: it manages the release times and deadlines of software tasks in reaction to environment events, such as clock ticks. When an environment interrupt occurs, E code may call a driver that reads a sensor port, or writes an actuator port, or transfers a value between ports, and it may release a task to the system. If an RTOS is used, the released task enters the ready queue, and can be dispatched by the system scheduler. A Giotto compiler that proves time safety is a *schedule-generating compiler*: it generates, in addition to E code, also S code, which represents a feasible schedule for the generated E code. S code is executed by the S machine, which replaces the system scheduler. S code is proactive code: it manages the execution of released tasks on the available CPUs. S code may dispatch a task for a certain amount of time (time-slice preemptive scheduling), or until another task is released (priority preemptive), or until the task completes (nonpreemptive). In other words, the S language is an expressive, executable schedule description language. Our implementation executes intertwined E and S code produced by the Giotto compiler, and thus provides the kernel functionality of an RTOS.

The usefulness of SCC rests on two premises. First, the execution of S code is more efficient, and at least as flexible, as the use of the scheduler provided by an RTOS. This is substantiated by our experiments, where we achieve up to 35% overhead reduction and demonstrate the ability to change scheduling strategies when switching Giotto modes [7]. Second, it is often more efficient to check the feasibility of a schedule than to generate the schedule ("proof checking is easier than proof generation"). Whenever this is the case, then it is beneficial to have the Giotto compiler generate a feasible schedule once, and attach it to the generated E code in the form of S code. Then, before execution, the target platform may check the schedule in order to be sure that it can execute the code without time-safety violations (i.e., without missing any deadlines specified by the original Giotto program). While preemptive single-CPU scheduling for Giotto is simple [6], it is NP-hard to generate nonpreemptive or distributed schedules for Giotto programs, even if the program has only a single mode. We show that these schedules, once expressed in S code, can be checked in time linear in the size of the E code and the frequency of events. More generally, SCC provides an efficient implementation of Giotto in the presence of many scheduling constraints. In particular, in Section 4 we show how Giotto can be compiled onto a time-triggered architecture using SCC.

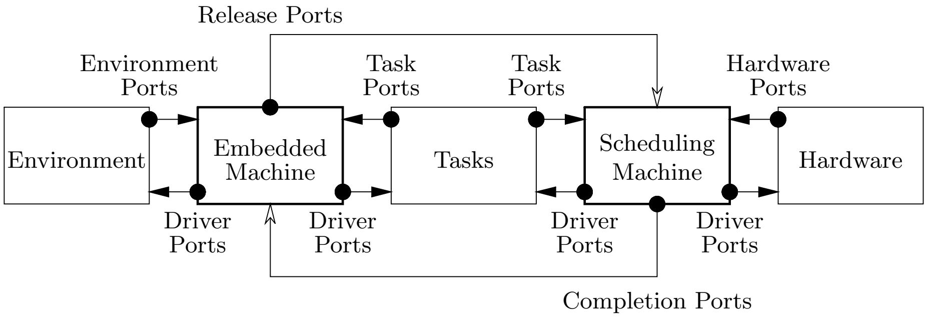

Fig. 1. The E machine and the S machine

2 SCC: Embedded Code Plus Scheduling Code

The E machine [5] is a virtual machine that interpretes E code, which supervises the execution of software tasks in response to physical events. The S machine is a virtual machine that interpretes S code, which specifies the temporal order of task execution. Figure 1 shows how E and S machine interact with the physical environment, software tasks, and hardware platform. We first review the E machine and then introduce the S machine.

2.1 The Embedded Machine

Interface. Physical environment processes communicate information to the E machine through *environment ports*, such as clocks and sensors, and application software processes, called *tasks*, communicate information to the E machine through *task ports*. The E machine communicates information to the environment and the tasks by calling system software processes, called *drivers*, which write to *driver ports*, for instance actuators. The E machine releases tasks for execution to the task scheduler (the S machine, or the scheduler of an RTOS) by writing to *release ports*, and the scheduler signals the completion of tasks to the E machine through *completion ports*. Hence, the environment, task, and completion ports are the *input ports* of the E machine. A change of value at an input port is an *input event* and causes an interrupt. The E machine monitors the occurrence of input events through *triggers*. In this paper, we consider only *time triggers*. A time trigger can be specified as a positive integer δ; it watches an environment clock and becomes enabled δ clock ticks after its activation. Tasks, drivers, and triggers are external to the E machine and must be implemented in some programming language like C. Tasks are preemptive, user-level code without internal synchronization points; drivers are system-level code during whose execution all interrupts that correspond to input events are disabled. The task idle is a special task that never completes.

E code. There are three non-control-flow E code instructions. The `call`(d) instruction initiates the execution of a driver d, and the E machine waits until d

```
while $ProgramCounter \neq \bot$ do
  $i := Instruction(ProgramCounter)$
  if call($d$) $= i$ then
    if driver $d$ accesses a port of a task $t$ that has been released but not completed
    then throw a time-safety exception else execute $d$
  else if schedule($t$) $= i$ then
    if task $t$ has already been released but not yet completed
    then throw a time-safety exception else emit a signal on the release port of $t$
  else if future($g, a$) $= i$ then
    append the trigger binding $(g, a, s)$ to $TriggerQueue$, where $s$ is the current
    state of the input ports that occur in $g$
  end if
  $ProgramCounter := Next(ProgramCounter)$
end while
```

Algorithm 1: The E code interpreter

is finished before proceeding to the next E code instruction. The `schedule`(t) instruction releases a task t to be executed, concurrently with other released tasks, and then the E machine proceeds immediately to the next E code instruction. The `schedule` instruction does not order the execution of tasks, nor does it relinquish control of the CPU to the scheduler. The `future`(g, a) instruction activates the trigger g and marks the E code at address a for execution at the future time instant when g becomes enabled. In order to handle multiple active triggers, the E machine maintains a queue of *trigger bindings* (g, a, s), where s is the current state of the input ports watched by the trigger g, which is required for evaluating g in the future. E code has also control-flow instructions such as `if`(c, a) and `return`. In the former case, if the condition c (a predicate on input ports) evaluates to true, then the E machine proceeds to address a; otherwise it proceeds to the next instruction. The `return` instruction terminates the execution of E code.

Algorithm 1 summarizes the E code interpreter. For each input event, the E machine checks the trigger bindings in *TriggerQueue*. The first trigger binding (g, a, s) in the queue with an enabled trigger g is removed from the queue and the interpreter is invoked with *ProgramCounter* set to address a. This is repeated until the queue contains no trigger binding with an enabled trigger. Then, the E machine relinquishes control of the CPU to the task scheduler, which is either provided by an RTOS [5] or implemented as S machine (see below). The goal of the scheduler is to execute the released tasks so that they complete before their deadlines. E code specifies task deadlines in two ways: once released, a task t must complete (1) before any driver accesses a port of t, and (2) before t is released again. If a task violates one of these conditions, then the E machine throws a *time-safety exception*; otherwise the execution is *time-safe*.

Example. Figure 2 shows a Giotto program [4] in the left column and, in the middle and right column, E code generated by the Giotto compiler [6]. The Giotto program is a simplified version of a program that implements the flight

```
start hover {
  mode hover() period 120ms {
    exitfreq 3 do cruise(switch);
    taskfreq 1 do pilot();
    taskfreq 2 do control();
    taskfreq 3 do lieu(); }
  mode cruise() period 120ms {
    exitfreq 2 do hover(switch);
    taskfreq 1 do pilot();
    taskfreq 2 do control();
    taskfreq 4 do move(); }
```

```
  H0: if(switch, C0 + 1)
      schedule(pilot)
      schedule(control)
      schedule(lieu)
      future(40ms, H40a)
      return[h0]
H40a: if(switch, H40b)
      schedule(lieu)
      future(20ms, H60)
      return[h40]
H40b: future(20ms, C60)
      return
 H60: schedule(control)
      future(20ms, H80a)
      return[h60]
H80a: if(switch, H80b)
      schedule(lieu)
      future(40ms, H0)
      return[h80]
H80b: future(10ms, C90)
      return
```

```
 C0: if(switch, H0 + 1)
     schedule(pilot)
     schedule(control)
     schedule(move)
     future(30ms, C30)
     return[c0]
C30: schedule(move)
     future(30ms, C60)
     return[c30]
C60: if(switch, H60)
     schedule(control)
     schedule(move)
     future(30ms, C90)
     return[c60]
C90: schedule(move)
     future(30ms, C0)
     return[c90]
```

Fig. 2. A Giotto program with two modes, and the generated E code

controller of a model helicopter [8]. The program consists of a *hover* mode, in which the helicopter maintains its airborne position, and a *cruise* mode. A Giotto mode specifies a set of periodic tasks. The task periods are specified through frequencies relative to the mode period, which is 120ms for both modes shown here. The *pilot* task is invoked, in both modes, every 120ms to perform path planning. The task outputs flight directions to the *control* task, which controls the servos and is invoked every 60ms in both modes. In the *hover* mode, the *control* task reads the current position estimation from the *lieu* task, which is invoked every 40ms. In the *cruise* mode, the *control* task receives position and velocity information from the *move* task, which is invoked every 30ms. The system can switch mode every 40ms from *hover* to *cruise*, and every 60ms from *cruise* to *hover*. A mode switch is initiated when the *switch* condition evaluates to true. For simplicity, we omitted all sensor and actuator code.

The E code in the middle column of Figure 2 implements the *hover* mode, and the right column implements the *cruise* mode. The execution of the program starts in the *hover* mode, thus the E code execution starts with the `if`(*switch*, `C0` + 1) instruction at address `H0`. If *switch* evaluates to true, then the E machine proceeds to the `schedule`(*pilot*) instruction that follows the instruction at address `C0`. This corresponds to a switch to the *cruise* mode. If *switch* evaluates to false, then the E machine proceeds to the `schedule`(*pilot*) instruction that follows the `if` instruction. The subsequent `schedule` instructions release the *control* and *lieu* tasks for execution. Then, the `future`(40*ms*, `H40a`) instruction makes the E machine execute the E code at address `H40a` after 40ms elapse. For now, the `return`[`h0`] instruction terminates the E code execution and relinquishes control of the CPU to the task scheduler. The expression [`h0`] is an *E code annotation*, which will be explained later.

```
while $ProgramCounter \neq \bot$ do
  $i := Instruction(ProgramCounter)$
  $ProgramCounter := Next(ProgramCounter)$
  if call($d$) $= i$ then
  if driver $d$ accesses a port of a task $t$ that has been released but not completed
  then throw a time-safety exception else execute $d$
  else if dispatch($t, h, a$) $= i$ then
    if there is a thread instance in $ThreadSet$ with a non-idle task then
      throw a time-sharing exception
    else
      insert the thread instance $(t, ProgramCounter, h, a, ReferenceTime)$
      into $ThreadSet$ and set $ProgramCounter$ to $\bot$
    end if
  else if idle($h$) $= i$ then
    insert the thread instance $(\mathsf{idle}, \bot, h, ProgramCounter, ReferenceTime)$
    into $ThreadSet$ and set $ProgramCounter$ to $\bot$
  else if fork($a$) $= i$ then
    insert the thread instance $(\mathsf{idle}, \bot, \mathsf{true}, a, s)$ into $ThreadSet$, where $s$ is the
    current value of the system clock
  end if
end while
```

Algorithm 2: The S code interpreter

2.2 The Scheduling Machine

Interface. The hardware on which the S machine runs communicates information to the S machine through *hardware ports*, such as clocks and message buffers, and the tasks communicate information to the S machine through task ports. The S machine communicates information to the hardware and the tasks by calling drivers. An external task handler (in our case, the E machine) signals the release of tasks to the S machine through release ports, and the S machine signals the completion of tasks to the task handler by writing to completion ports. Hardware, task, and release ports are input ports for the S machine, and changes in their values are input events. The S machine monitors input events through *timeouts*. In this paper, we consider two kinds of timeouts. A *clock timeout* is specified by a nonnegative integer δ; it watches a system clock and expires δ clock ticks into the S code thread that contains the timeout. The *release timeout* Θ expires as soon as any task is released.

S code. There are four non-control-flow S code instructions. The `call`(d) instruction initiates the execution of a driver d. Similar to the E machine, the S machine waits until d is finished before proceeding to the next S code instruction. The `dispatch`(t, h, a) instruction begins or resumes the execution of a task t until the timeout h expires. There are two possible outcomes: (1) the S machine proceeds to the next instruction when t completes, in case this happens before the timeout h expires, or (2) the S machine proceeds to the instruction at address a when the timeout h expires, in case this happens before t completes. Case (1) applies also if

```
RM: dispatch(lieu, +4)        EDF0/60: dispatch(lieu, +4)        EDF40/80: dispatch(control, +4)
    dispatch(control, +3)              dispatch(control, +3)               dispatch(lieu, +3)
    dispatch(pilot, +2)                dispatch(pilot, +2)                 dispatch(pilot, +2)
    idle()                             idle()                              idle()
    fork(RM)                           fork(EDF40/80)                      fork(EDF0/60)
    return                             return                              return
```

Fig. 3. Rate-monotonic (RM) and earliest-deadline-first (EDF) S programs for the *hover* mode of the Giotto program from Figure 2

t has not been released or has already completed when the `dispatch` instruction is encountered. The `idle`(h) instruction makes the S machine idle until the timeout h expires even when there are released tasks that have not been completed. The S machine proceeds to the subsequent instruction when the timeout expires. The `fork`(a) instruction marks the S code at address a for execution in parallel to the S code that follows the instruction. The S code at a is a new *thread* of execution. In order to handle multiple threads, the S machine maintains a set of thread instances. S code may also have control-flow instructions, but we do not consider them here. The `call`, `fork`, and control-flow instructions of S code are *transient instructions*, as they execute, like E code instructions, in logical zero time. In contrast, the `dispatch` and `idle` instructions are *timed instructions*, as they cause a passage of time.

Algorithm 2 summarizes the S code interpreter, which maintains the set *ThreadSet* of thread instances. A *thread instance* has the form (t, b, h, a, s), where either t is the idle task and $b = \bot$, or t is a regular task and b is the address at which the S machine continues executing when t completes before the timeout h expires. When h expires before t completes, the S machine continues executing, instead, the S code at address a. The *reference time* s is the time when the thread instance was created by a `fork` instruction, which is required for evaluating clock timeouts. The S machine is woken up by an input event: a hardware port may signal the completion of a task or the expiration of a clock timeout, or a release port may signal the release of a task. If a task t completes, then the thread instances of the form $(t, b, \cdot, \cdot, s)$ become *enabled*, with *ProgramCounter* set to b, and *ReferenceTime* set to s. If a timeout h expires, then the thread instances of the form $(\cdot, \cdot, h, a, s)$ become *enabled*, with $\mathit{ProgramCounter} = a$ and $\mathit{ReferenceTime} = s$. With each input event, every enabled thread instance is removed from *ThreadSet* and executed until a timed instruction is encountered (at that time, *ProgramCounter* is set to $\bot$ by the interpreter). The execution order for the enabled thread instances in *ThreadSet* is chosen nondeterministically. If control ends at `dispatch` instructions in more than one thread, then a *time-sharing exception* occurs, because only one task can be dispatched on the CPU; otherwise the execution is *time-sharing.*

Examples. The left column of Figure 3 shows an S program with the initial address `RM` which implements rate-monotonic (RM) scheduling of the tasks in the *hover* mode of the Giotto program from Figure 2. The S program only works if no mode switching occurs; S code that supports mode switching will be discussed below. We use `idle`() to abbreviate `idle`(Θ) and `dispatch`$(t, +n)$ to abbreviate

```
NP0: dispatch(move)      NP30: dispatch(move)          NP60: dispatch(pilot)  NP90: dispatch(control)
     dispatch(control)         dispatch(pilot, NP60)         dispatch(move)          dispatch(move)
     idle()                    idle()                        idle()                  idle()
     fork(NP30)                fork(NP60)                    fork(NP90)              fork(NP0)
     return                    return                        return                  return
```

Fig. 4. Nonpreemptive (NP) S program for the *cruise* mode

dispatch$(t, \Theta, a + n)$, where a is the address of the instruction itself and n is a relative offset. In the example, the offsets in dispatch instructions always point to fork instructions. The S program dispatches the tasks in a fixed, RM order starting with the task that has the highest frequency. Suppose that the tasks in the *hover* mode have been released by the E machine. Now, the S machine starts executing the dispatch(*lieu*, +4) instruction. The *lieu* task executes until it either completes or is preempted by the release of some other task. If the *lieu* task completes first, then the S machine proceeds to the dispatch(*control*, +3) instruction and executes the *control* task. Otherwise, if some task is released before *lieu* completes, the S machine proceeds to the fork(RM) instruction and forks a new thread starting again at address RM. The following return instruction terminates the current thread. If all tasks complete during the execution of a thread, then the idle() instruction is reached and the S machine waits until some task is released to start a new thread. Note that the execution of the S code is time sharing, and if there exists a feasible RM schedule for the tasks of the *hover* mode, then the S code guarantees the time-safe execution of the E code for the *hover* mode.

As an alternative to RM scheduling, the S code in the middle and right column implements earliest-deadline-first (EDF) scheduling of the tasks in the *hover* mode. The initial address of the EDF S program is EDF0/60. The thread at EDF0/60 initially dispatches the *lieu* task followed by the *control* and *pilot* tasks. However, unlike the RM S program, as soon as the *lieu* task is released again, the S machine forks a new thread at address EDF40/80 and terminates the current thread. Now, at the 40ms instant, the *control* task is dispatched before the *lieu* task, because the *control* task has an earlier absolute deadline than the *lieu* task. If the *control* task already completed before the 40ms instant, then the S machine immediately proceeds to the next instruction and dispatches the *lieu* task. At the 60ms instant, the situation is the same as at 0ms. So, we fork again a thread at EDF0/60. At the 80ms instant, the absolute deadlines of all tasks are the same. Thus we can use the thread at EDF40/80 again.

The S code of Figure 4 implements nonpreemptive (NP) scheduling of the tasks in the *cruise* mode. The initial address of the NP S program is NP0. We use dispatch(t) to abbreviate dispatch(t, false, a), where a is the address of the next instruction. Thus a dispatch(t) instruction executes the task t until completion. Given the worst-case execution time $w(t)$ of each task t, the S program guarantees time safety if $w(move) + w(control) \leq 30ms$ and $2 \cdot w(move) + w(pilot) \leq 60ms$. Time sharing is ensured because the S program dispatches all tasks nonpreemptively, i.e., each task completes before another task is dispatched. Tasks may still be preempted by E code and S code, but not by other tasks.

```
h0/h60:                  h40/h80:                 c0/c30/c60/c90:
dispatch(lieu, +3)       dispatch(control, +3)    dispatch(move, +3)
dispatch(control, +2)    dispatch(lieu, +2)       dispatch(control, +2)
dispatch(pilot, +1)      dispatch(pilot, +1)      dispatch(pilot, +1)
return                   return                   return
```

Fig. 5. Earliest-deadline-first (EDF) S program for both modes

Figure 5 shows an S program that EDF schedules the tasks of both modes. In order to facilitate the interaction of E and S code in the presence of conditional-branch instructions (mode switching) in E code, we allow the E code to fork a new thread instance of S code through E code annotations. Recall the E code of Figure 2. The `return`[h0] instruction of the code block at address `H0` terminates the execution of the code block and, in addition, forks a new thread of S code at address `h0`. We start running the E program and S program by executing the E code at the initial address `H0`. Now the initial address of the S program is not required and thus $\perp$. The E code releases all three tasks of the *hover* mode and then creates a new S code thread starting at `h0`. The thread dispatches the three tasks in EDF order. Suppose that the *lieu* task completes before the 40ms instant, but not the *control* task. Thus, at the 40ms instant, the E code at `H40a` preempts the *control* task. Now, suppose that we switch from the *hover* to the *cruise* mode. We branch to the E code at `H40b`, which does not release any tasks but only jumps to the E code at `C60` after another 20ms elapse. Therefore, after executing the E code at `H40b`, the current S code thread continues where it was preempted by resuming the execution of the *control* task. No new thread is created. Now, suppose that the *control* task completes before the 60ms instant, but not the *pilot* task. Now, at the 60ms instant, the E code at `C60` preempts the *pilot* task. Unlike before, the *control* task is released now, no matter if we switch mode or not. Since the current S code thread becomes enabled upon the release of a task, the thread is terminated by jumping to the `return` instruction of the code block at `h0`. Suppose that we stay in the *cruise* mode, i.e., the code block at `C60` is executed. In addition to the *control* task, the *move* task is released and a new S code thread at `c60` is created, which dispatches now the tasks of the *cruise* mode in EDF order. Note that the S code for the *cruise* mode is equivalent to RM S code, because the task frequencies in the *cruise* mode are harmonic. The execution of the remaining E and S code works in a similar way. The S program is time sharing, and since EDF is optimal for Giotto programs with multiple modes [6], if there exists a feasible, preemptive schedule for the tasks of the *hover* and *cruise* modes, then the EDF S program guarantees the time-safe execution of the E program regardless of any mode switching.

2.3 Schedule-Carrying Code

An *SCC program* is a pair $(\mathcal{E}, \mathcal{S})$ consisting of an E program $\mathcal{E}$ that shares a set of tasks with an S program $\mathcal{S}$. If $\mathcal{E}$ contains conditional-branch instructions, then its `return` instructions may be annotated with S code addresses as shown in the example above. The runtime system for SCC is the E machine interacting with the S machine through release and completion ports. The machines are invoked as follows: (1) if there is an enabled thread instance that contains a completed task, then the S machine must handle that thread instance before the E machine handles any enabled triggers; (2) if there is an enabled trigger binding, then the E machine must handle that trigger binding before the S machine handles any expired timeouts. The reason for (1) is that, when a task completes, the dispatching thread must be processed before any E code is executed in order to enable the prompt handling of output data upon task completion, say through driver calls in S code. The reason for (2) is that the E machine must be invoked before the S machine when an E code trigger is enabled at the same time when an S code timeout expires, because the E code may release tasks that require scheduling service from the S code. This will be formalized in the next section.

Implementation. With the help of Marco Sanvido, we have developed a microkernel on a StrongARM SA-1110 processor with 206MHz which executes SCC programs using an integrated implementation of the E and S machine [7]. The microkernel has a footprint of 8kB. We have tested the microkernel with SCC programs that implement four periodic, nonharmonic 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 microkernel is invoked every 1ms by a timer. The periodic release of the tasks is described by E code. We have compared the performance of an EDF scheduler with S code that specifies EDF scheduling. The average time spent in the EDF scheduler is 1.4μs to schedule 4 tasks and 35μs to schedule 100 tasks. The average time to execute, instead, the EDF S code grows from 2.2μs for scheduling 4 tasks to only 3.9μs for scheduling 100 tasks. With 100 tasks, EDF S code performs 35% better than the EDF scheduler (51% vs. 78% CPU utilization). For details, the reader is referred to [7].

3 Generating SCC vs. Checking SCC

We show that checking the time safety of given S code can be exponentially simpler than generating time-safe S code. Checking the time safety of SCC is a program-analysis problem. Following the tradition of path-insensitive program analysis, we define the *abstract* semantics of SCC, which ignores all port values and assumes that both branches of conditionals can be taken. On this abstract semantics, we present an efficient algorithm that provides a sufficient check for the time safety of SCC generated from single-mode Giotto, which specifies a set of periodic tasks.

3.1 Abstract Semantics of SCC

An *abstract E program* $\mathcal{E} = (V, E, \kappa, \lambda, \hat{v})$ over a set T of tasks consists of a control-flow graph (V, E) which is a binary[1] digraph, two edge-labeling functions κ and λ, and an initial node $\hat{v} \in V$. Each edge $e \in E$ is labeled with an instruction $\kappa(e)$ and an argument $\lambda(e)$ as follows:

[1] Each node has at most two successors.

– $\kappa(e) = \texttt{call}$ and $\lambda(e) \subseteq T$. The execution of e calls a driver that accesses ports of the tasks in $\lambda(e)$.
– $\kappa(e) = \texttt{schedule}$ and $\lambda(e) \in T$. The execution of e releases the task $\lambda(e)$.
– $\kappa(e) = \texttt{future}$ and $\lambda(e) \in \mathbb{N}_{>0} \times V$. The execution of e activates the trigger with the binding $\lambda(e) = (\delta, u)$, which means that after δ time units, E code will be executed starting from control location u.

We require that the initial node $\hat{v}$ has a single successor $\hat{v}' \in V$ such that $\kappa(\hat{v}, \hat{v}') = \texttt{future}$ and $\hat{v}'$ is a leaf. We assume that a driver can access the ports of at most two (or any fixed number) of tasks, and that all integers δ can be stored in constant space; hence the size of $\mathcal{E}$ is $|V|$. A *state* $q = (v, r, s, \tau)$ of $\mathcal{E}$ consists of a program counter $v \in V$, a status $s\colon T \to \mathbb{N} \cup \{\top, \bot\}$ for each task, and a queue $\tau \subseteq (\mathbb{N} \times V)^*$ of trigger bindings. For each task $t \in T$, the status $s(t) = \top$ indicates that t has been released at the current time instant and not yet executed; the status $s(t) \in \mathbb{N}$ indicates that t has been released at a previous time instant and executed for $s(t) \geq 0$ time units; the status $s(t) = \bot$ indicates that t has been completed (or not yet released). Note that the number of different status functions, and hence the number of states, is exponential in $|T|$.

An *abstract S program* $\mathcal{S} = (V, E, \mu, \nu, \kappa, \lambda)$ over a set T of tasks consists of a control-flow graph (V, E) which is a binary digraph, two node-labeling functions μ and ν, and two edge-labeling functions κ and λ. Each control location $v \in V$ is labeled by one of the following:

– $\mu(v) = \texttt{dispatch}$ and $\nu(v) \in T$ and v has a successor v_1 such that $\lambda(v, v_1) = \bot$, and possibly a second successor v_2 such that either $\lambda(v, v_2) = *$ or $\lambda(v, v_2) \in \mathbb{N}$. The execution of v dispatches the task $\nu(v)$. If $\lambda(v, v_2) = *$ and a task is released before $\nu(v)$ completes, control proceeds to v_2; if $\lambda(v, v_2) \in \mathbb{N}$ and $\nu(v)$ does not complete within the first $\lambda(v, v_2)$ time units from the time at which the current thread was created, control proceeds to v_2; otherwise, control proceeds to v_1 when $\nu(v)$ completes.
– $\mu(v) = \texttt{idle}$ and v has a single successor v' such that either $\lambda(v, v') = *$ or $\lambda(v, v') \in \mathbb{N}$. The execution of v idles the processor until a task is released (if $\lambda(v, v') = *$), or until $\lambda(v, v') \in \mathbb{N}$ time units pass from the time at which the current thread was created.
– $\mu(v) = \nabla$. This indicates that control is at a transient instruction.

If $e = (v, v')$ and $\mu(v) = \nabla$, then the edge $e \in E$ is labeled by one of the following:

– $\kappa(e) = \texttt{call}$ and $\lambda(e) \subseteq T$. The execution of e calls a driver that accesses ports of the tasks in $\lambda(e)$.
– $\kappa(e) = \texttt{fork}$ and $\lambda(e) \in V$. If $\lambda(e) = u$, then the execution of e creates a new thread, which starts at control location u.

By our assumptions, the size of $\mathcal{S}$ is $|V|$. A *state* $q = (s, \theta)$ of $\mathcal{S}$ consists of the status $s\colon T \to \mathbb{N} \cup \{\top, \bot\}$ for each task, and a set θ of threads. Each thread (u, δ) consists of a program counter $u \in V$ and a number $\delta \in \mathbb{N}$ of time units for which the thread has been executed. If u is a leaf, then the thread (u, δ) has terminated and may be removed from θ.

An *abstract single-processor SCC program* $\Pi = (\mathcal{E}, \mathcal{S}, \Phi)$ over a set T of tasks consists of an abstract E program $\mathcal{E}$ over T, an abstract S program $\mathcal{S}$ over T, and an *annotation function* Φ that maps each leaf in the control graph of $\mathcal{E}$ to a node in the control graph of $\mathcal{S}$. When the E code execution arrives at a leaf v, this creates a new thread of S code which starts at control location $\Phi(v)$. A *state* of Π is a tuple $q = (v, s, \tau, \theta)$ such that (v, s, τ) is a state of $\mathcal{E}$, and (s, θ) is a state of $\mathcal{S}$. The state q violates *nonpreemption* if there exist two different tasks $t_1, t_2 \in T$ such that $s(t_1), s(t_2) \notin \{\top, 0, \bot\}$. The state q violates *time sharing* if there exist two different threads $(u_1, \cdot), (u_2, \cdot) \in \theta$ such that $\mu(u_1) = \mu(u_2) = \texttt{dispatch}$ and $s(\nu(u_1)) \neq \bot$ and $s(\nu(u_2)) \neq \bot$. The state q violates *time safety* if there exists a task $t \in T$ with $s(t) \neq \bot$ and one of the following: (1) v has a successor v' in $\mathcal{E}$ with either $\kappa(v, v') = \texttt{call}$ and $t \in \lambda(v, v')$, or $\kappa(v, v') = \texttt{schedule}$ and $t = \lambda(v, v')$; or (2) there exists a thread $(u, \cdot) \in \theta$ such that $\mu(u) = \nabla$ and u has a successor u' in $\mathcal{S}$ with $\kappa(u, u') = \texttt{call}$ and $t \in \lambda(u, u')$. The state q has a *transition* to the state $q' = (v', s', \tau', \theta')$ if one of the following:

Completion S transition The state q is *completion enabling*, that is, there exist a thread $(u, \delta) \in \theta$ and a successor u' of u in $\mathcal{S}$ such that $\mu(u) = \texttt{dispatch}$ and $s(\nu(u)) = \bot$ and $\lambda(u, u') = \bot$. Then $(v', s', \tau') = (v, s, \tau)$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

Transient S transition The state q is *transient enabling*, that is, there exist a thread $(u, \delta) \in \theta$ and a successor u' of u in $\mathcal{S}$ such that $\mu(u) = \nabla$. Then $(v', s', \tau') = (v, s, \tau)$ and one of the following:
- $\kappa(u, u') = \texttt{call}$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.
- $\kappa(u, u') = \texttt{fork}$ and $\lambda(u, u') = \hat{u}$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta), (\hat{u}, 0)\}$.

E transition q is neither completion nor transient enabling but *E enabling*, that is, either (1) v has no successor and $(0, u) \in \tau$ for some u, or (2) v has a successor v' in $\mathcal{E}$. If (1) let $(0, u')$ be the first such pair in τ. Then $v' = u'$ and $s' = s$ and $\tau' = \tau \setminus \{(0, u')\}$ and $\theta' = \theta$. If (2) then one of the following:
- $\kappa(v, v') = \texttt{call}$ and $s' = s$ and $\tau' = \tau$.
- $\kappa(v, v') = \texttt{schedule}$ and $\lambda(v, v') = t$ and $s'(t) = \top$ and $s'(t') = s(t')$ for all tasks $t' \in T \backslash \{t\}$, and $\tau' = \tau$.
- $\kappa(v, v') = \texttt{future}$ and $s' = s$ and $\tau' = \tau \circ \{\lambda(v, v')\}$.

In all three cases, if v' is a leaf, then $\theta' = \theta \cup \{(\Phi(v'), 0)\}$; otherwise $\theta' = \theta$.

Timeout S transition The state q is neither completion nor transient nor E enabling but *timeout enabling*, that is, there exist a thread $(u, \delta) \in \theta$ and a successor u' of u in $\mathcal{S}$ such that $\mu(u) \in \{\texttt{dispatch}, \texttt{idle}\}$ and either (1) $\lambda(u, u') \in \mathbb{N}$ and $\lambda(u, u') \leq \delta$, or (2) $\lambda(u, u') = *$ and $s(t) = \top$ for some task $t \in T$. Then $(v', s', \tau') = (v, s, \tau)$ and $\theta' = (\theta \backslash \{(u, \delta)\}) \cup \{(u', \delta)\}$.

Time transition The state q is neither completion nor transient nor timeout nor E enabling. Then $v' = v$. For all tasks $t \in T$, if there exists a thread $(u, \delta) \in \theta$ with $\mu(u) = \texttt{dispatch}$ and $\nu(u) = t$, then either $s(t) = \top$ and $s'(t) = 1$, or $s(t) \in \mathbb{N}$ and $s'(t) = s(t) + 1$, or $s'(t) = \bot$; if no such thread exists, then either $s(t) = \top$ and $s'(t) = 0$, or $s'(t) \neq \top$ and $s'(t) = s(t)$. In case $s'(t) = \bot$, we say that on the transition (q, q'), task t *completes* after execution time $s(t) + 1$. The queue τ' results from τ by replacing each trigger binding (δ, u) by $(\delta - 1, u)$. The set θ' results from θ by replacing each thread (u, δ) by $(u, \delta + 1)$.

Note the priorities implied by this definition: transient S code that is enabled by the completion of tasks has priority over E code, which has priority over all remaining S code. A *trace* of the abstract single-processor SCC program Π is a sequence $\gamma = q_0, q_1, \ldots, q_n$ of states of Π such that (1) $q_0 = (\hat{v}, \hat{s}, \emptyset, \emptyset)$, where $\hat{v}$ is the initial node of $\mathcal{E}$, and $\hat{s}(t) = \perp$ for all tasks $t \in T$, and (2) for all $i \geq 0$, there is a transition from q_i to q_{i+1}. The trace γ is *time-safe* (resp. *time-sharing*; *nonpreemptive*) if no state of γ violates time safety (time sharing; nonpreemption).

An *abstract multiprocessor SCC program Π* over a set P of processors and a set T of tasks is a function that assigns to each processor $p \in P$ a pair (T_p, Π_p), where $T_p \subseteq T$ such that $\{T_p \mid p \in P\}$ is a partition of the task set T, and Π_p is an abstract single-processor SCC program over the set T_p of tasks. A *trace* γ of Π is a function that assigns to each processor $p \in P$ a trace γ_p of Π_p such that all single-processor traces γ_p contain the same number of time transitions. The trace γ is time-safe (resp. time-sharing; nonpreemptive) if γ_p is time-safe (time-sharing; nonpreemptive) for all $p \in P$. A *wcet map* w: $P \times T \to \mathbb{N}$ assigns to every processor p and task t a worst-case execution time $w(p,t) > 0$. There are a number of techniques for obtaining wcet maps, e.g. [2]. If P is a set of *identical* processors, then $w(p_1, t) = w(p_2, t)$ for all processors $p_1, p_2 \in P$ and tasks $t \in T$. The trace γ of Π is an *w-trace* if for all processors $p \in P$, tasks $t \in T$, and $i \geq 0$, if t completes on the transition (q_i, q_{i+1}) of γ_p, then it completes with execution time at most $w(p,t)$. The abstract (single- or multiprocessor) SCC program Π is *time-safe* (resp. *time-sharing*; *nonpreemptive*) for wcet map w if all w-traces of Π are time-safe (time-sharing; nonpreemptive). The time safety (time sharing; nonpreemption) of an abstract SCC program can be checked by searching the state space, but the number of states is exponential in the number of tasks. We will see that the check is simpler for SCC programs of a special form.

3.2 Giotto-Generated SCC

The E programs generated from Giotto programs have a special form [6]. Let G be a Giotto program [4] with task set T and a single mode m, let π_m be the period of m, let f_t be the frequency in mode m of task t, and let f_m be the least common multiple of all task and actuator frequencies in m. Then $d_m = \pi_m / f_m$ denotes the time interval between consecutive input events. The abstract E program $\mathcal{E} = (V, E, \kappa, \lambda, \hat{v})$ over a set $T' \subseteq T$ of tasks is *G-generated* if the control graph (V, E) consists, in addition to the initial location $\hat{v}$ and its successor $\hat{v}'$, of a set of f_m acyclic digraphs $\mathcal{E}_i$, for $0 \leq i < f_m$, such that every node in $\mathcal{E}_i$ has at most one successor, and $\mathcal{E}_i$ consists of a source node v_i followed by (1) a sequence of $O(|G|)$ edges (v, v') with $\kappa(v, v') = \texttt{call}$, followed by (2) a sequence of edges (v, v') with $\kappa(v, v') = \texttt{schedule}$ and $\lambda(v, v') = t$ for each task $t \in T'$ for which $(i \cdot f_t / f_m) \in \mathbb{N}$, followed by (3) a single edge (v, v') with $\kappa(v, v') = \texttt{future}$ and $\lambda(v, v') = (d_m, v_{(i+1) \texttt{mod} f_m})$. Moreover, $\lambda(\hat{v}, \hat{v}') = (0, v_0)$. If all numbers in G

(mode period as well as task and actuator frequencies) are bounded by n, we have $|V| = O(|G| \cdot n)$. An abstract single-processor SCC program $\Pi = (\mathcal{E}, \cdot, \cdot)$ is *G-generated* if $\mathcal{E}$ is a G-generated abstract E program over the set T of all Giotto tasks. An abstract multiprocessor SCC program Π is *G-generated* if for each processor $p \in P$, if $\Pi_p = (\mathcal{E}_p, \cdot, \cdot)$, then $\mathcal{E}_p$ is a G-generated abstract E program over the set T_p of tasks that are assigned to processor p.

Proposition 1. *Let P be a set of identical processors, let G be a single-mode Giotto program with task set T, and let w be a wcet map for P and T. It is NP-hard in the strong sense to decide if there is a G-generated abstract multiprocessor SCC program over P and T which is time-safe and time-sharing for w.*

Proof. Reduction from BIN PACKING [3]. Given an instance of BIN PACKING, we choose as many processors as there are bins, and construct a single-mode Giotto program whose tasks have periods equal to the bin capacity. □

Proposition 2. *Let G be a single-mode Giotto program with task set T, and let w be a wcet map for T. It is NP-hard in the strong sense to decide if there is a G-generated abstract single-processor SCC program over T which is nonpreemptive, time-safe, and time-sharing for w.*

Proof. Reduction from NSPT (nonpreemptive scheduling of periodic tasks) [1]. □

Checking time safety becomes simpler if we restrict also the shape of S code. Let G be a Giotto program with task set T and numbers bounded by n, as above. The abstract S program $\mathcal{S} = (V, E, \mu, \nu, \kappa, \lambda)$ is *simple* if the control graph (V, E) is acyclic and for every node $v \in V$, (1) if $\mu(v) = \triangledown$, then v has at most one successor, and (2) if $\mu(v) = \triangledown$ and v' is a successor of v with $\kappa(v, v') = \texttt{fork}$, then v' is a leaf. Condition (1) ensures that the S code does not contain conditional branching. Condition (2) ensures that the S code is single-threaded, i.e., during execution, there is always at most one thread in the thread set. Single-threadedness, in turn, implies time sharing. An abstract single-processor SCC program $\Pi = (\mathcal{E}, \mathcal{S}, \Phi)$ is *simple G-generated* if (1) $\mathcal{E}$ is a G-generated abstract E program over T, (2) $\mathcal{S}$ is a simple abstract S program of size $O(|G| \cdot n)$ which does not contain numbers (clock timeouts) larger than the time step d_m, and (3) for each leaf v of $\mathcal{E}$, if v is not the successor of the initial node of $\mathcal{E}$, then $\Phi(v)$ is a leaf of $\mathcal{S}$. An abstract multiprocessor SCC program Π is *simple G-generated* if for each processor $p \in P$, the single-processor program Π_p over T_p satisfies conditions (1)–(3). Note that for the Giotto program G from Figure 2, the E code from Figure 2 together with the S code samples from Figures 3 and 4 yields simple G-generated SCC programs.

Proposition 3. *Let P be a set of (nonidentical) processors, let G be a single-mode Giotto program with task set T and numbers bounded by n, and let w be a wcet map for P and T. It can be checked in time $O(|G| \cdot n)$ if a given simple G-generated abstract multiprocessor SCC program is time-safe (resp. nonpreemptive) for w.*

```
h0:                     h40:                     h60:                      h80:

dispatch(lieu, 10, +2)  dispatch(control, +2)    dispatch(lieu, +2)        dispatch(lieu, 30, +4)
idle(10)                dispatch(lieu, +1)       dispatch(control, +1)     dispatch(control, 30, +2)
call(ctrl_in)           return                   return                    idle(30)
dispatch(lieu, +2)                                                         call(lieu_out)
dispatch(control, +1)                                                      dispatch(control, +1)
return                                                                     return
```

```
c0:                     c30/c60:                 c90:                       p0:

dispatch(move, 10, +2)  dispatch(move, +2)       dispatch(move, 20, +4)     call(pilot_in)
idle(10)                dispatch(control, +1)    dispatch(control, 20, +2)  dispatch(pilot, +3)
call(ctrl_in)           return                   idle(20)                   idle(120)
dispatch(move, +2)                               call(move_out)             call(pilot_out)
dispatch(control, +1)                            dispatch(control, +1)      return
return                                           return
```

Fig. 6. Distributed, TDMA-based S code for a two-processor TTA

Proof. For unconditional single-threaded S code, if all traces in which each task completes with an execution time equal to the time given by the wcet map w is time-safe, than all w-traces are time-safe. Therefore all transitions are deterministic, and it suffices to check the time-safety of a trace whose duration corresponds to one mode period. The number of states thus explored is $O(|G| \cdot n)$. $\Box$

For multimode Giotto programs, we know that if each mode in isolation is time-safe under EDF scheduling, then the whole program is time-safe under EDF [6]. Furthermore, we can check the EDF schedulability of a single mode by solving a utilization equation [6]. Hence, for SCC, it remains to be checked if a given abstract SCC program represents an EDF schedule, such as the example from Figure 5. This can be done in time linear in the size of the Giotto program.

4 SCC for Time-Triggered Networks

We illustrate how to generate distributed SCC programs that run on a *time-triggered architecture* (TTA) [9], whose nodes are connected by a bus on which all communication is scheduled according to a collision-free TDMA protocol. Each time slot assigns exclusive network access to one of the nodes to send data. Each node has a host and a network processor connected by a send and a receive buffer. Thus the host processor can execute programs while data is being sent or received. To send data, the host processor loads the send buffer before its time slot arrives. Similarly, to receive data, the host reads the receive buffer after a time slot ends. There are an E machine and an S machine running on each host processor. The E code portion of a distributed SCC program may be generated from Giotto. The S code portion specifies the execution order of released tasks, and also calls drivers that transport data between message buffers and tasks before and after the appropriate time slots. Thus the TDMA protocol imposes additional timing constraints on the tasks.

Example. Figure 6 shows distributed, TDMA-based S code for two host processors, p_0 and p_1, which execute the Giotto program from Figure 2. Suppose that

p_0 executes the *pilot* task, while p_1 takes care of the other tasks by executing the E code in Figure 2 with the `schedule`(*pilot*) instructions removed. For p_1, we use the E code `P0`: `schedule`(*pilot*); `future`(120, `P0`); `return`[`p0`]. Suppose that the output of the *pilot* task is read by the *control* task, while the outputs of the *lieu* and *move* tasks are read by the *pilot* task. For this purpose, we use a TDMA-schedule with two time slots, l_0 and l_1, where l_0 is from 0ms to 10ms, and l_1 is from 110ms to 120ms. Processor p_0 sends during l_0, and p_1 sends during l_1. The *pilot_out* driver writes the output of the *pilot* task into the send buffer of p_0. Similarly, the *pilot_in* driver reads the input for the *pilot* task from the receive buffer of p_0. On processor p_1, the *ctrl_in* driver reads the input for the *control* task from the receive buffer. The *lieu_out* and *move_out* drivers write the outputs of the *lieu* and *move* tasks, respectively, to the send buffer. Note that tasks may be executed during the time slots l_0 and l_1, because in a TTA each host has a separate network processor that handles the network traffic. For example, the *pilot* task may run for its full period of 120ms. The *control* task, on the other hand, cannot start before 10ms elapse from the beginning of the mode period, because its inputs depend on values received during the time slot l_0. Likewise, the *lieu* and *move* tasks must complete before 110ms. The S code represents an EDF schedule *under the constraints* imposed by the TDMA protocol. As the constraints are the same for all modes of the Giotto program, it can be shown that this schedule is optimal.

References

1. Y. Cai and M.C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. *Algorithmica* 15:572–599, 1996.
2. 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. In *Embedded Software*, LNCS 2211, pp. 469–485. Springer, 2001.
3. M.R. Garey and D.S.Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.
4. T.A. Henzinger, B. Horowitz, C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proc. IEEE* 91:84–99, 2003.
5. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. Programming Language Design and Implementation*, pp. 315–326. ACM, 2002.
6. T.A. Henzinger, C.M. Kirsch, R. Majumdar, S. Matic. Time-safety checking for embedded programs. In *Embedded Software*, LNCS 2491, pp. 76–92. Springer, 2002.
7. C.M. Kirsch, T.A. Henzinger, M.A.A. Sanvido. *A Programmable Microkernel for Real-Time Systems.* Technical Report CSD-03-1250, UC Berkeley, 2003.
8. C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, W. Pree. A Giotto-based helicopter control system. In *Embedded Software*, LNCS 2491, pp. 46–60. Springer, 2002.
9. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer, 1997.
10. G.C. Necula. Proof-carrying code. In *Proc. Principles of Programming Languages*, pp. 106–119. ACM, 1997.