Computational Systems Seminar
SS 2004
University of Salzburg
Department of Computer Science
Jakob–Haringer–Straße 2
A–5020 Salzburg
Austria

The Context of Schedule Carrying Code

Author: Emilia Coste

Academic Supervisor: Christoph Kirsch

1st of July, 2004

Abstract:

This paper represents a survey of the research context of the paper: "Schedule Carrying Code" by Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic, presented at EMSOFT in 2003. The survey presents models for real-time systems, the Giotto methodology, current research in scheduling, and the Schedule Carrying Code (SCC) with an analysis of the contribution.

Table of Contents

1. Introduction

In the information age, computers and information are present everywhere, in systems ranging from household objects such as TV-sets, microwave ovens, to state-of-the-art technology used in space exploration. Although they have different purposes and capabilities, all these systems incorporate control laws for a specific behavior, some environment sensors (e.g., buttons, temperature sensors, etc.), increasingly powerful processing capabilities in form of microprocessors or microcontrollers and various actuators (e.g., LEDs, displays, engines), forming a special class of systems: embedded systems.

In Real-Time Systems the correctness of the system behavior does not depend only on the value of the computation but also on the time at which the results are obtained. A real-time task has a *deadline*, which is the maximum time when its execution must complete. For a *soft* real-time task, meeting the deadline is important for performance but it is not critical for the system behavior. For a *hard* real-time task, missing the deadline may have catastrophic consequences on the controlled environment.

Although hardware technology advances and the computational speed increases, this does not imply that the timing requirements are met. A fast system does not guarantee the individual timing requirements of each task in all circumstances. Therefore rather than being fast, a real-time system should be predictable [But97].

Some of today's real-time control systems are still designed using empirical approaches [But97] and implemented using low-level programming techniques, such as coding in assembly language, changing drivers, timers, and task or interrupt priorities. This approach leads to unpredictable software with difficulties in maintenance, verification and code understanding. If the time constraints cannot be verified, then there is no guarantee that the system will work properly in any situation. The system may work fine for some time, and fail in certain cases of overload. In hard real-time systems, a missed deadline can be catastrophic. Software testing cannot guarantee predictability, as it is impossible to have sufficient test cases for input sensor data and environment state.

A guarantee for real-time systems can be achieved only by more sophisticated design methodologies, static analysis of source code, schedulability analysis and operating systems mechanisms to support computation under time constraints [But97]. We need in embedded systems a shift of parading from the low-level programming to software engineering techniques. The developer should work with abstractions which support determinism, composability, reuse, code generation and verification.

2. Models for Real – Time Programming

The concept of time is the main characteristic that differentiates real-time systems from other computing systems. Physical processes evolve in real-time, while software processes evolve in a so called *soft-time* [Kir02]. Soft-time becomes real-time only at the time instants of input and output, when the software processes get or communicate values to the physical processes. In between, soft-time is discontinuous; it may depend on task priorities, precedence relations, semaphores for shared resources, network communication, hardware utilization, or scheduling strategy.

Real-time programming models differ in the mapping of soft-time to real-time. [Kir02] presents the synchronous, scheduled, and timed models.

2.1. Synchronous Model

The synchronous model (see Figure 1) is based on the assumption that all computations and communications take no time; therefore the soft-time is always zero. The synchronous process is triggered by some event in the environment or other computational processes, and reacts to events in zero time providing instantaneously the output. A synchronous program is *deterministic* if it computes at most one reaction for any event and control state, and *reactive* if it computes at least one reaction for any event and control state [Kir02]. The synchronous program must have instantaneous and deterministic reaction, therefore the compiler has to prove that it terminates, i.e. there are no infinite cycles in the synchronous program.

The implementation of this model may approximate synchrony by reacting to an event before another event appears. A compiler should implement a verification of reactivity and synchrony, based on the computation of the worst case execution time from static code analysis.

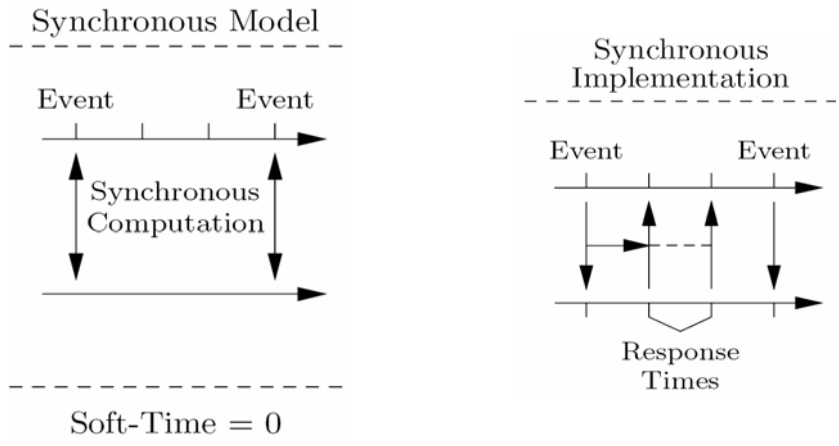

Figure 1. The Synchronous Model and Implementation [Kir02]

Examples of languages based on the synchronous model are Esterel with explicit control flow and Lustre, a dataflow language.

The Esterel language [GB00] is based on the mathematical semantics of Finite State Mealy Machine (FSM) that ensures a deterministic behavior. It is an imperative language that provides a set of primitives for expressing concurrency, communication and preemption. The compiled Esterel program is represented into a Boolean Equation System (netlists) with Boolean registers (latches). The states of the FSM are vectors of the reachable register values and the transitions between states are the input/output predicates (there are no loops).

The dataflow approach is close to the automatic control and electronic circuit design methodologies providing a formal and functional model that is easier to develop and maintain. Synchronous dataflow languages such as Lustre [HCRP91] combine the synchronous with the dataflow programming models for an efficient description of reactive systems with a continuous behavior. Lustre is a declarative language that provides primitives and structures which restrict the dataflow systems to only those that

can be implemented as bounded automata-like programs in the sense of Esterel. However, expressing sequential or evolving behavior with a set of equations is challenging.

2.2. Scheduled Model

The scheduled model (see Figure 2) is based on the classical scheduling theory, where the operating system provides a scheduler or dispatcher, who decides which process, thread or task to execute at which point in time. The soft-time is the time it takes for a program to execute, the time between release and termination, i.e. the response time. The soft-time is not defined in the scheduling model; it depends on the operating system, processor utilization, and scheduling strategy. The time the task completes depends on the implementation, but the soft-time must be less than the real-time deadline.

The scheduling is implemented in the operating system, but the schedulability analysis requires the computation of the worst case execution time, which is done at compile time. Hard and soft tasks are the same from the compiler point of view, but they need different runtime mechanisms. An operating system which can handle both types of tasks will guarantee the deadlines for hard real-time tasks and will minimize the average response time for the soft ones. Section 4 describes the current research in scheduling.

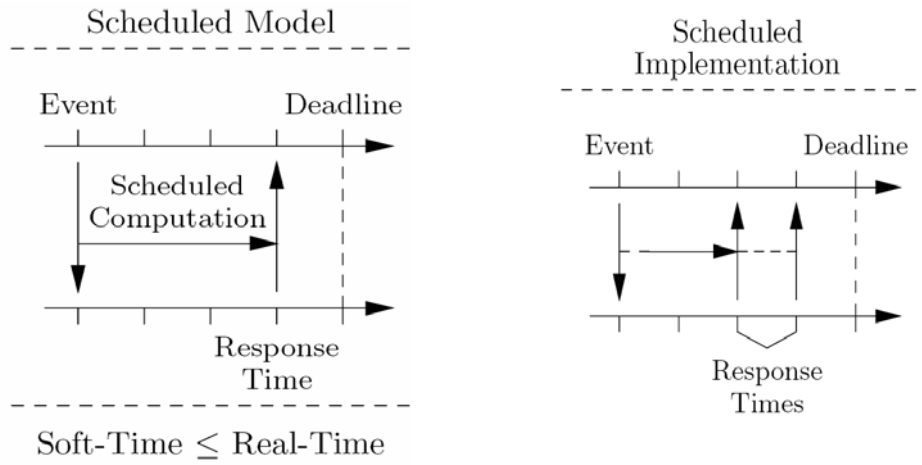

Figure 2. The Scheduled Model and Implementation [Kir02]

2.3. Timed Model

In a timed model (see Figure 3), all computations and communication take logically a fixed amount of time, no matter how much time they actually take; therefore the soft-time is always equal to the real-time [Kir02]. The task will provide the output at the required response time, even if the task has finished earlier and the output is available. The compiler or the runtime system has to check time safety, i.e. there is enough soft-time so that the execution finishes before the required response time.

The timed model assures determinism and predictability. The behavior of the program depends only on the task properties (including control law) and the environment; it does not depend on the platform or a specific implementation. A methodology for embedded software development called Giotto is presented in Section 3.

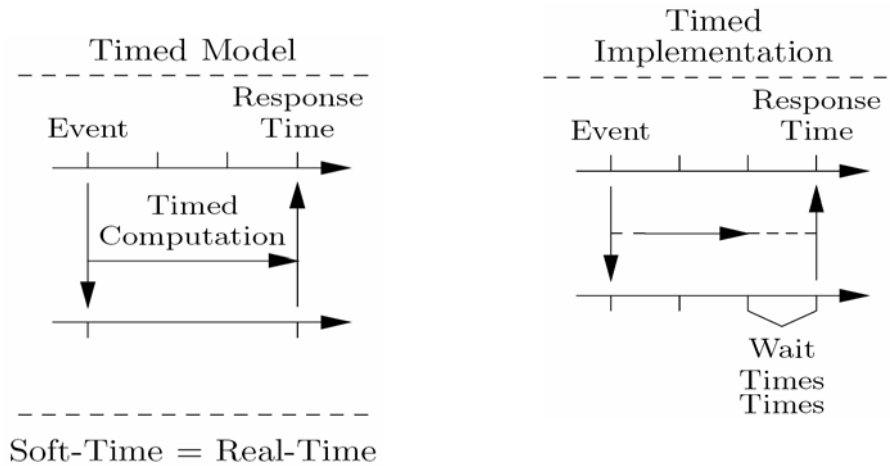

Figure 3. The Timed Model and Implementation [Kir02]

3. Giotto

Giotto is a new methodology for developing hard real-time embedded systems, based on a time-triggered high-level programming language, the corresponding compiler and an embedded runtime system based on virtual machines (e.g., E machine, S-Machine).

The traditional development cycle of an embedded control system begins with the design of the system and its expression as a mathematical model (using some tools such as Simulink). The model is later implemented manually or automatically (via specific code generators) and the resulting code is tested and tuned for the target platform. Most of the time some additional tweaks must be performed on the code to make it work as required on the target platform. In this process the correspondence between the mathematical model and the actual code is often lost. The final code is tightly coupled with the platform (hardware and operating system) and can hardly be reused in similar systems or enhanced with new functionality without major revisions or complete redesign and implementation.

The Giotto development methodology specifically decouples the timing aspects and the functionality aspects of an embedded software control system by introducing an intermediate layer of abstraction between the mathematical model and the corresponding implementation code (see Figure 4).

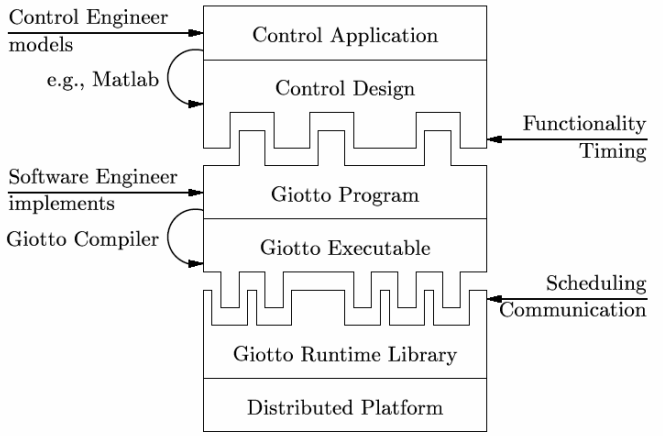

Figure 4. Embedded control systems development with Giotto [HHK01]

The Giotto language expresses the reactivity of the application related to the external environment. It allows a clear separation, in a platform independent way, of the logical behavior of the implemented model with the physical implementation details such as interrupts handlers, CPU mapping and scheduling of parallel jobs.

3.1. Giotto Tasks

The basic functional unit of the Giotto language is the *task* that represents some specific functionality of the implemented model. Any Giotto task is a periodically executed piece of code expressed in some language such as C that can be compiled for the target platform. The execution of the task functionality on the target system is triggered by real time and takes a non-zero bounded time. One important aspect of the Giotto system is that tasks are independent and there are no internal synchronization points, such as waiting for external resources or other events. This way any task will contain only the functionality code that in some cases can be automatically generated from the mathematical model of the control system (e.g., using Simulink). All resource utilization policy and environment reaction is handled by the Giotto runtime system.

Communication between tasks or between tasks and sensors/actuators is realized via *ports* handled by lightweight pieces of code called *drivers*. The ports can be assimilated as buffers that contain the sensor and task output values. The input ports of each task are distinct from any other ports in the Giotto program. However, the output ports of a task may be shared with some other tasks only if the tasks are not running in the same mode. In order to pass some values between different instances of the same task (usually between successive invocations of the same task), the task may have state ports. The drivers transport and convert values between ports and execute logically in zero time (the actual execution time on the target platform is negligible), see Figure 5:

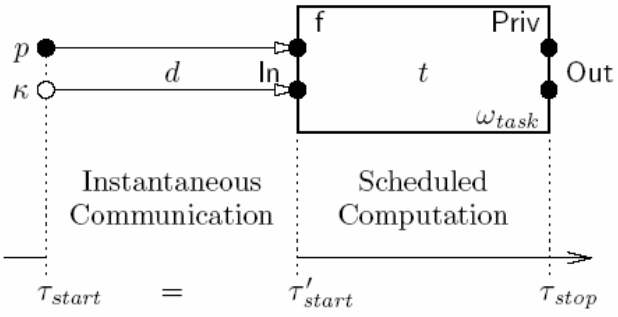

Figure 5. The time line for an invocation of a task *t* [HHK01G]

Any Giotto program will not specify the precise time when a task will run on a specific CPU, or whether it may be preempted. The Giotto program will only specify the release times of each task and the Giotto compiler will automatically generate timing code that models the desired behavior on the target platform. Instantaneous communication and time-deterministic as well as value-deterministic computation are the three essential ingredients of the Giotto logical abstraction [HHK01G].

3.2. Giotto Modes

There can be more tasks that may run logically in parallel with similar or different periods. Tasks may be grouped together into *modes* that denote a specific state of the system. One Giotto program may be in only one mode at a time. However, depending

on the intended functionality the system may switch at runtime between different modes of operation. This allows a great flexibility to build complex application such as fly-by-wire systems.

Each mode of the application has a predefined period and all tasks that run within that mode have their periods derived from the mode period divided by an integer. In this way any task of a mode will be executed an exact number of times during a mode period. The execution of a Giotto mode (along with all its tasks) for a single period determines one round. In distributed environments this round may also be equal or a multiple of the network round.

3.3. FLET

The Giotto methodology conforms to the timed model and is based on the concept of Fixed Logical Execution Time (FLET) that applies to each task of the embedded control system. In contrast to generic real-time software development methodologies, where task releasing and task ending are two non-deterministic moments in the software control system, the FLET concept assumes that the release and end of the task are two exact moments that do not depend on the scheduling strategy of the real-time operating system or the CPU speed but on the control system model; the output of any task will only be available at the end of its FLET. The resulting system is highly deterministic and its behavior can be easily predicted. Moreover, the jitter at the output ports of the system will be close to zero.

The execution time of each task on the target platform is non-zero and is bounded by two values: Worst Case Execution Time (WCET) and FLET, where WCET is smaller than FLET. The WCET of a task is a platform dependent value influenced by the target OS and CPU speed. The FLET is however model-dependent and expresses the logical time when the task get its inputs and when its outputs are available to other tasks or actuators. During the period of FLET the task may be scheduled for execution at any point in time, it may be preempted by some other running tasks or the OS, but logically it will appear as executing for the entire period.

4. Scheduling Theory

The scheduling algorithms depend on the task model, whether tasks must meet their deadline in any circumstances (i.e. they are hard or soft tasks), or whether task arrival is regularly repeated (i.e. they are periodic or aperiodic tasks). Furthermore, sporadic tasks are aperiodic tasks with a known minimum inter-arrival time. Periodic tasks may have late release times, and earlier deadlines in correspondence with the period. Also, tasks may have mutual exclusion constraints to access shared resources, or precedence constraints, i.e. some tasks must execute in a defined order. Also they may allow or not preemption, i.e. they may be interrupted by the scheduler to execute another active task.

Typical task attributes [But97] are (see Figure 6):

- *Arrival time* (a), i.e. request time or release time (r), is the time when a task becomes ready for execution
- *Start time* (s) is the time when a task starts its execution
- *Finishing time* (f) is the time when a task completes its execution
- *Deadline* (d) is the latest time when the task should finish

- *Computation time* (C), i.e. execution time, is the time needed to execute the task without interruption. The *worst case execution time* (WCET) is the maximum computation time needed on a particular processor.
- *Lateness* (L) = f – d. If the task completes before the deadline, its lateness is negative.

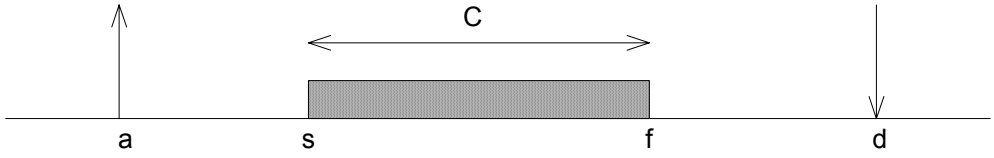

Figure 6. Typical task attributes.

A schedule is *feasible* if all tasks can be completed according to the given constraints. A set of tasks is *schedulable* if there exists at least one algorithm that can produce a feasible schedule [But97]. An algorithm is *clairvoyant* if it knows in advance the arrival times of all tasks. A *heuristic* algorithm may produce a feasible schedule, but does not guarantee to find it, since it does not explore all possible solutions; heuristic algorithms are used to reduce the complexity. An algorithm is *optimal* if there is no task set that this algorithm can not schedule and another algorithm from the same class (e.g., fixed priority algorithms) can. If an optimal algorithm misses a deadline, then no other algorithms of the same class can meet it. Here the goal is to achieve a feasible schedule and a cost function is not defined, but in a general sense an algorithm is *optimal* if it minimizes some given cost function defined over the task set (e.g. average response time, total completion time, weighted sum of completion times, maximum lateness, and maximum number of late tasks) [But97].

4.1. Approaches

Traditionally there are two approaches in scheduling. In *off-line scheduling* all decisions are computed at compile time, and stored in a dispatch table; at run-time no scheduler is needed, but only a dispatcher which takes the next entry from the table. In *on-line scheduling* all scheduling decisions are taken at run-time, when a new task is released or when a task terminates. The scheduling algorithms are called *static*, if scheduling decisions are based on fixed parameters, assigned to tasks before their activation, and are *dynamic* if scheduling decisions are based on dynamic parameters that might change during system evolution [But97].

Each approach has advantages and drawback and is suited to different domain problems. Off-line scheduling usually is more deterministic, and can use complex algorithms, as it is executed before the system is started, so we don't have to worry about missing deadlines because of the scheduler overhead. It manages distributed applications with complex constraints and has small runtime overhead; it is actually a table lookup. The drawback is that off-line scheduling needs a priori knowledge about all system activities and events, therefore has no flexibility. On-line algorithms take the decisions at run-time and can handle up-to-date information about tasks, therefore are more flexible. But on-line algorithms have run-time overhead which can not be ignored, and should be minimized by avoiding computationally expensive algorithms. Also schedulability analysis might not be trivial.

Usually off-line scheduling is implemented as a search tree, such as in branch-and-bound algorithms. On-line scheduling is usually based on assigning static or dynamic priorities to tasks.

4.2. Algorithms for Aperiodic Tasks

The algorithms for aperiodic tasks are summarized in [But97]. *Jackson's algorithm* (i.e., Earliest Due Date) is optimal in minimizing the maximum lateness for independent aperiodic tasks which have synchronous arrival times. The algorithm schedules the tasks in the order of increasing deadlines. *Horn's algorithm* (i.e., Earliest Deadline First, EDF) extends the concept for tasks with arbitrary arrivals, therefore it allows preemption. The algorithm executes at any instant the task with the earliest absolute deadline among all the ready tasks. EDF may schedule even tasks with precedence constraints, by transforming them into a set of independent tasks; the release times and deadlines are modified so that each task cannot start before its predecessors and cannot preempt its successors.

When preemption is not allowed and tasks can have arbitrary arrivals, it is NP-hard to find a feasible schedule. When arrival times are known a priori, non-preemptive scheduling is usually implemented by branch-and-bound algorithms [But97]. A search tree has the root as an empty schedule and a leaf is a complete schedule. Intermediate nodes correspond to partial schedules, and are extended by inserting a new task. The complexity of the search is exponential in the worst case, and it is reduced by using Bratley's algorithm. It prunes the tree, by abandoning a branch when it finds a feasible schedule, or when adding a node to the current path causes a missed deadline. The Spring algorithm handles non-preemption but also additional constraints such as precedence relations, and resource constraints. The algorithm uses a heuristic function H to select a promising path. At each step of search, the schedule is extended with the task which has the smallest value for function H.

These algorithms have been defined for aperiodic tasks on uniprocessor systems but can be extended for periodic tasks, or even distributed systems.

4.3. Algorithms for Periodic Tasks

For periodic tasks, a wide used schedulability test is the utilization equation. The *processor utilization* factor is the fraction of the processor time used for the execution of the task set: $U = \sum_{i=1}^{n} \frac{WCET_i}{T_i}$, where T_i is the period of task i, and $WCET_i$ is the worst case execution time of task i. Both are constant for each task instance, during system evolution. For an algorithm A we compute the *least upper bound* $U_{\text{lub}}(A)$ of the processor utilization factor. The schedulability test for a task set with the processor utilization U is:

- If $U > 1$, the task set is not schedulable, i.e. no scheduling algorithm can guarantee the schedulability.
- If $U \le U_{\text{lub}}(A)$ the tasks are schedulable by algorithm A (the condition is sufficient but not necessary).
- If $U_{\text{lub}}(A) < U \le 1$, nothing can be said on the feasibility of the task set, and a different schedulability test should be used.

The *Rate Monotonic* (RM) algorithm [LL73] is optimal among fixed (i.e., static) priorities algorithms. It assigns priorities to tasks according to their request rates, which are known before execution and do not change in time. A task with higher request rate, i.e. shorter

period, will get a higher priority. Also tasks can be preempted. For the RM algorithm, the least upper bound is $U_{\text{lub}} = n(2^{1/n} - 1)$, with the limit $\lim_{n\to\infty} U_{\text{lub}}(n) = \ln 2$.

The *Earliest Deadline First* (EDF) algorithm [LL73] is optimal among dynamic priorities algorithms. It assigns priorities to tasks according to their absolute deadline, which depends on the task instance. A task with earlier deadline will get a higher priority. Also tasks can be preempted. Notice that the same algorithm is used for aperiodic tasks, too. For the EDF algorithm, the least upper bound is $U_{\text{lub}} = 1$, the maximum value for the processor utilization. Therefore, EDF is optimal among all algorithms based on priority assignment.

These algorithms consider the deadline equal to the period, but they were extended to support tasks with relative deadlines less than their period. *Deadline Monotonic* extends RM and assigns priorities inversely proportional to the relative deadline. It is still optimal among static priorities algorithms. A sufficient and necessary schedulability test is given by the *response-time analysis*, computing the interference from the higher priority tasks [ABR93]. The analysis is based on the concept of *critical instant*, at which a request for a task will have the maximum response time; the critical instant occurs when the task is released simultaneously with all higher priority tasks. For each task we can compute the maximum response time and check if it is less than the deadline:

$\forall i : 1 \le i \le n, \quad R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \le D_i$, where tasks are sorted by decreasing priority.

For the EDF with deadlines less than periods, the schedulability test is done using the *processor demand approach* ([BRH90], [JS93], [But97]). It is based on the concept of *busy period*, which is the first time instant when all the released tasks are completed.

4.4. Algorithms for Hybrid Sets

[But97] presents a comprehensive survey on algorithms for hybrid task sets consisting of hard periodic tasks and soft aperiodic_tasks. A simple algorithm is *background scheduling*, which schedule aperiodic tasks when there are not periodic tasks ready to execute. The *Slack Stealing* algorithm creates a passive task which tries to make time for servicing aperiodic tasks by "stealing" processing time from the periodic tasks without risking to miss their deadlines.

The other algorithms are based on the concept of a *server*, which is a periodic task used for handling aperiodic requests as soon as possible. The server has a period$_s$ a WCET, and a *capacity*. In general the server is scheduled with the same algorithm like the periodic tasks. Examples of fixed-priority servers: the *Polling Server* serves any pending aperiodic requests within the available capacity, and if none is pending it suspends itself until the beginning of its next period; the *Deferrable Server* preserves its capacity until the end of the period, if no requests are pending upon the invocation of the server; the *Priority Exchange* preserves its high-priority capacity by exchanging it for the execution time of a lower-priority periodic task.

The algorithms presented in this section are for periodic tasks scheduled with RM; they are also available for periodic tasks scheduled with EDF, they just have a different name and schedulability test.

4.5. Existing Systems

Fixed priority scheduling is easy to implement and it is widely use in embedded systems, being the basis for operating systems such as VxWorks, OSE [OSE], or OSEK [OSEK]. Off-line scheduling is used in the Time Triggered Protocol [Kop93].

Hard-real time operating systems were built in research centers aiming at implementing new scheduling strategies and guaranteeing predictability (see [But97] for a comprehensive survey). The MARS System has a static and off-line scheduler (i.e., table driven), and also has phases, i.e. modes. The Spring system has four modules for scheduling: dispatcher, local dynamic scheduler, distributed scheduler which tries to find a node available when a task cannot be locally guaranteed, and meta-level controller which adapts parameters of scheduling algorithms to load conditions. The RK System handles more types of real-time tasks: imperative (scheduled with First Come First Served - FCFS), hard and soft (EDF) and non real-time tasks (scheduled with FCFS in background). The ARTS System use an object oriented paradigm. A scheduling policy object can be selected from: RM plus servers for aperiodic tasks, EDF, Least Laxity First (LLF), FCFS, Round Robin (RR). The HARTIK System uses dynamic preemptive scheduling and on-line guarantee (EDF, and Stack Resource Policy).

4.6. Trends in Scheduling

The fundamental algorithms in scheduling date from the years '70. Since then the focus was to add more flexibility, such as to handle periodic and aperiodic tasks. [Foh94] supports flexibility in off-line scheduling, to serve aperiodic requests and mode changes.

Research is done for mode dynamic task sets with reconfiguration at run-time; components with a more complex structure, which have an internal state, as described by a state machine, or systems with modes; and even for considering the dynamic environment and not the WCET computed by static analysis.

On-going work is done in distributed scheduling, allocation of tasks to nodes [Tin92], considering network effects, bounding worst case communication delay [Tin94].

Future trends are: *feedback-based scheduling* for estimating current workload conditions and tuning parameters; *energy-aware scheduling* which takes voltage into account minimize energy consumption but preserving the timing properties; *elastic task models* which allow some task parameters to support adjustments for limited time. In complex systems it becomes useful to combine more scheduling algorithms, using for example: hierarchical scheduling, dual priority scheduling, or slot shifting [Foh94].

5. Schedule Carrying Code

5.1. Giotto Tool Chain

The Giotto methodology allows streamlined operations from the design phase to the implementation and running phase of and embedded control system. The model designed with a mathematical modeling tool (e.g., Simulink) is translated into a Giotto timed program via an automatic translator (S/G Translator) [GS03]. The functionality code, which in some cases can also be automatically generated from the mathematical model with tools such as TargetLink, and the Giotto program will be compiled to

produce system specific native code for the functionality of the model and machine independent reactivity code called *E code*, see Figure 7.

Time safety checking based on the WCET calculations for each task and the generated E code must be performed before running the resulting executable code on the target system. Time safety checking should ensure that the behavior of the real system will match the model behavior. The execution of a task is regarded as time safe if once released, the task completes before any driver accessed its output ports and before another instance of it will be released.

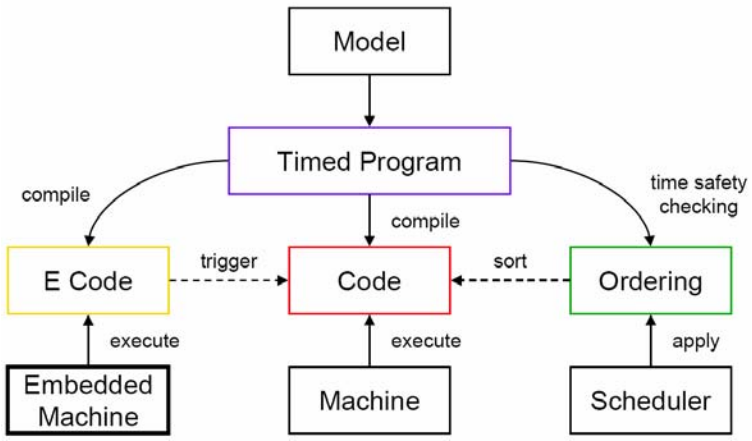

Figure 7. The Giotto development tool chain [Kir02s]

The actual execution of the generated code on the target system is highly influenced by the scheduling strategies for multiple parallel tasks. With Giotto, schedulability analysis can be performed offline in advance and the executable code may be accompanied by scheduling information in the form of *S code*. A run-time scheduler and an ordering policy are not needed anymore, because the Giotto compiler will generate both E code and S code, which are executed at run-time.

Together the E code and S code executed by the appropriate virtual machines will determine the exact moments in time when tasks will be released, preempted or drivers will be executed to transport values between sensors, tasks and actuators, in other words when each piece of native executable code will be executed on the target CPU. The E code and S code are platform independent and have instructions for activating external functions such as tasks or drivers and flow control.

5.2. The Embedded Code

The embedded code (E code) [HKM03] is a sequence of instructions that are interpreted by a virtual machine called Embedded Machine (E machine) [HK02] in order to produce the desired behavior of the target system. It represents the reactivity/timing code that manages the release and deadline times of software tasks in reaction to external events, and may be regarded as the closest relation with the mathematical model of the system.

However, the E code is highly portable, predictable and moreover composable. Multiple E codes can be summed together to extend the functionality of the system in a modular way. By specifying the logical times when drivers should be called, tasks should be released or finish, the E code is totally decoupled from the real time (and the system time of the platform).

The E code instructions recognized by the E machine are the following:

- Call(d) – calls a driver to transfer values between different ports
- Schedule(t) – schedules a task for execution (release task)
- Future(g,a) – prepares a later jump to a different section of E code
- If (c,a) – controls the flow via logical functions on current values of sensor/task ports.
- Return – ends the current section of E code.

5.3. The Scheduling Code

The scheduling code (S code) [HKM03] has some similar instructions with the E code but their purpose is slightly different:

- Call(d) – the driver calls are intended for distribution when values must be passed transparently between different nodes.
- The flow control is used to implement arbitrary scheduling that does not conform to standard scheduling algorithms (such as Rate Monotonic or Earlier Deadline First). Fork (a) – allows parallel execution of sequences of S code in order to run multiple applications on the same CPU or provide load balancing on multi CPU systems (e.g., SMP machines).
- Dispatch(t,h,a) – begins or resumes the execution of a task *t* until timout *h* expires. A schedule instruction in the E code will have as correspondent one or more dispatch instructions in S code, because the E machine will just release the task for execution (or flag it as ready for execution) and actually the S machine will dispatch that task on CPU for a specific amount of time.
- Idle(h) – makes the S machine idle until timeout *h* expires.

In contrast with the E machine where all timing is related to logical time, the S machine when interpreting the S code will take into account the real time of the system or the global time in a distributed environment.

5.4. Scheduling for Giotto Programs

If S code is not used, then the compiler generates only E code and has to check the time safety. [HKM02] proves that for E code generated from Giotto programs, EDF scheduling is well-defined and optimal. If all modes of the Giotto program are reachable (i.e., where each mode may be executed for a full period), then the schedulability may be checked with the processor utilization equation for each mode independently. Time safety checking is exponential for arbitrary E code, but it is polynomial for E code generated from Giotto programs [HKM02]. Time Safety checking for Giotto generated S code is also polynomial (see Section 7.3).

Flexibility in scheduling Giotto programs was introduced by [Hor03] making the following extensions to the original Giotto model:

- All jobs have a WCET, including sensors readings, drivers, actuators
- All jobs, not just task invocations, may be preemptible
- The Spillover concept - The execution of jobs of one round of a Giotto program should be allowed to continue into the next round of the program, if all the other constrains and precedence constraints are met.

6. Examples of S code Execution

We take as example the hover mode of a flight-by-wire system. The mode has a period of 120ms and contains a set of three tasks, for which the frequency relative to the mode period is specified, as described in the following Giotto program [HKM03] (the code is simplified here, omitting for example the mode switching conditions). Therefore the *pilot* task is invoked every 120ms, the *control* task is invoked every 60ms, and the *lieu* task every 40ms:

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

6.1. RM

Rate Monotonic scheduling of the *lieu*, *control*, *pilot* tasks in this flight-by-wire system is denoted by the following section of S code [HKM03]. The actual execution of the functionality code for these tasks on the CPU is depicted in Figure 8. We selected the values for WCET to emphasis preemption (e.g. the *control* task has a higher WCET and cannot terminate before the arrival of the *lieu* task).

```
Hover Mode S Code

RM:     dispatch(lieu, +4)
        dispatch(control, +3)
        dispatch(pilot, +2)
        idle()
        fork(RM)
        return
```

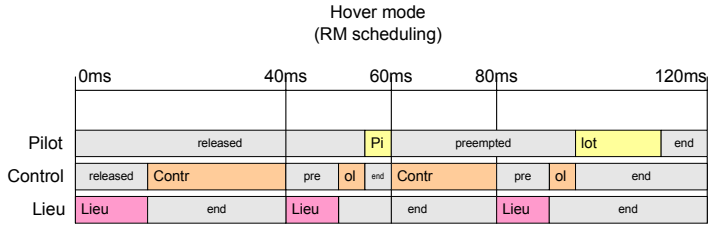

Figure 8. The RM scheduling

6.2. EDF

A more efficient scheduling of the same task set, using the Earlier Deadline First algorithm is denoted by the following sections of S code [HKM03]. There are less preemptions and the remaining time for the system idle task is generally larger and in a continuos block. However the S code for EDF is larger than the RM correspondent. The red circles from Figure 9 represent the deadlines considered for tasks scheduling with the EDF algorithm.

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

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

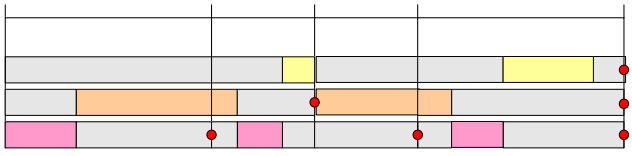

Figure 9. The EDF scheduling

7. The Contribution of Schedule Carrying Code

7.1. Proof Carrying Code

In the SCC approach, the compiler proves the existence of a feasible schedule by generating S code [HKM03]. S code is attached to the program (E code) and represents its schedule. A SCC executable code is a real-time program that carries its schedule as code, which may be validated at each use [HKM03]. SCC is an innovative approach by extending the paradigm of proof carrying code to the scheduling theory.

Proof carrying code (PCC) [Nec97] was introduced as a mechanism by which a code consumer can determine with certainty that it is safe to execute a program (generally in binary form) from an untrusted code producer. The procedure of establishing "trust" between the two entities is based on a formally defined safety policy expressed by the code consumer, a safety proof from the code producer attesting the fact that the program respects the safety policy, and a proof validator for the code consumer to check the proof. It is required that changing the program or the proof will result in a validation error. In the rare case when the program and the proof are changed so that the validation succeeds, the resulting PCC is considered also safe.

One of the most important aspects of the PCC is the formal safety policy provided by the code consumer that encapsulates the conditions it considers for the execution of an untrusted program. The safety policy consists of two components: safety rules – describing authorized operations and their preconditions; interface – describing the calling conventions between the code consumer and the untrusted program and their environment.

Producing PCC requires a certification process from the code producer that generates a proof accompanying the program with regards to the safety policy expressed by the code consumer. The proof and the functional binary code of the program are encoded together and delivered to the code consumer for validation and later use. Before running the untrusted code, the code consumer having the proof and the code in the PCC bundle is able to perform an offline validation of the proof that if successful guarantees the adherence of the program to the requested safety policy.

It is to be noted that producing the proof for an arbitrary program may be hard or even impossible. Some programming languages that are used mostly for mathematical problems (equations, boolean predicates, etc) are generally better suited for PCC, than the general purpose ones. Large programs are difficult to check for loop invariants and interfaces for all functions, but an advantage of PCC certification is that it can be performed offline assisted by special certifying compiles accompanied by program analyzers and code optimizers.

7.2. Table Driven & On-line Scheduler vs. S Machine

As discussed in Section 4.1, there are several approaches for running a set of tasks in an embedded system. In the case of a low performance CPU available on the target system every cycle matters and context switching is very expensive in terms of time and general performance. In this case the most common execution strategy relies on a precompiled dispatch table that was generated offline together with the binary code of the application. A more flexible approach for scheduling tasks on the target system relies on on-line scheduling during system runtime. It allows dynamic operations, on-the-fly system

changes and simpler compilation. The drawback is that CPU utilization is higher because of the scheduler overhead and without an offline time safety checking the system may fail in the case of CPU overload.

Using the S code and a virtual machine for interpreting its instructions merges the benefits of both approaches presented above. The S code is generated offline in a similar way to PCC and is considered as a schedulability proof for the target program. Contrary to table driven approach it may express more powerful scheduling strategies and with the help of loop/flow control instructions it can reduce the memory footprint of the schedule. The role of the on-line scheduler is taken by the lightweight S machine that will interpret the S code and dispatch the tasks at the right moments without any overhead incurred from standard scheduling algorithms, because the schedule is already generated in the form of S code. The system can easily be extended or changed even during runtime by changing the task set with the accompanying S code.

7.3. Generating vs. Checking SCC

Similar with PCC generating S code is a non trivial task as the S code will represent a feasible schedule for the system. Finding such schedule for an arbitrary set of tasks is NP hard but the process can be optimized when the S code is generated according to some optimal algorithms such as EDF.

On the other hand checking S code is a polynomial task by exploring every path of the possible running graph, presuming that all branches are taken [HKM03]. Checking the proof in form of S code for a given system can be accomplished both offline when the properties of the known in advance or online when adding another functionality to an existing running system. In the last case the safety policies expressed by the real-time constraints of the platform will be matched with the S code proof during the idle time of the system. In case that the S code represents a feasible schedule for the current running system, it may be loaded and executed together with the accompanying functionality code.

7.4. Advantages

Flexibility. It may be argued that S code is nothing more than an accompanying dispatch table for a given piece of code but the implementation aspects are less important than the concepts behind SCC. By the use of fork instruction parallel pieces of S code can be executed transparently on one or multiple CPUs without any changes to existing code. This allows greater flexibility in choosing the right hardware platform for an embedded system.

In the case of larger applications where there are possibly hundreds of tasks invocations per mode round the S code has a significant advantage in reducing the memory footprint of the same schedule. A typical table will have hundreds of entries (one for each invocation), but with the power of flow control and looping or parallel execution via fork the resulting S code could be a couple of lines. Swapping tables at runtime in order to implement a dynamic system is much harder than simply executing a different section of S code.

Composability and support for distribution. Via the call and fork instructions of the S code any given schedule can be implemented transparently on a single CPU system or distributed system. The S machine on one CPU system will run parallel sections of the S

code attached to a program via some time sharing mechanism, but the S machine running on a SMP system will simply for the same S code run each section of S code on a different CPU. Driver calls at the S code level will transparently interface any communication layer and may be used to transport values between the nodes of a distributed embedded system.

7.5. Measurements – Microkernel vs. Traditional On-line Scheduler

The following figures illustrate some measurements on an embedded system based on a StrongARM CPU running a microkernel of 8KB and various task sets (4, 10, 50 and 100 tasks) [KHS03].

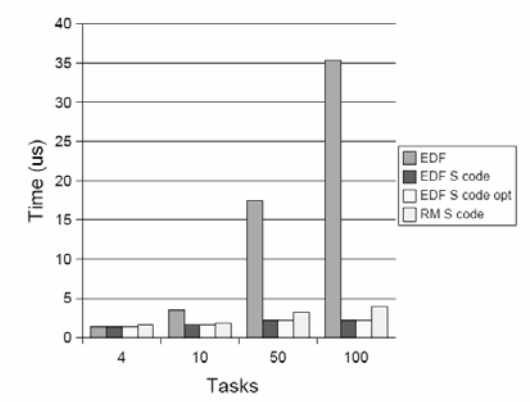

Figure 10. Kernel overhead [KHS03]

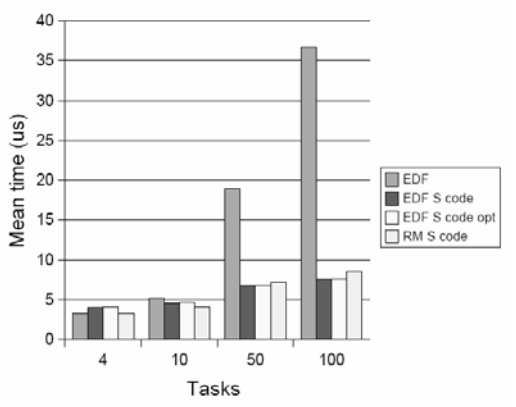

Figure 11. E + S code overhead [KHS03]

The Figure 10 illustrates the scheduling overhead of the microkernel in four cases: EDF online scheduler, and three S code based schedulers with EDF, optimized EDF and RM. The online scheduler keeps a sorted list of tasks to decide which task should run next. In the case of S code this is no longer needed as the sequence of instructions will choose the next task to be executed. The RM based S code is slightly slower because it dispatches more often already completed tasks.

The Figure 11 illustrates the overhead of the scheduler when running E code and S code. The overhead of the online scheduler can be drastically reduced by using any kind of S code (and it this case maintaining the system overhead under 10us even for 100 tasks). Optimized S code may have close to O(1) performance.

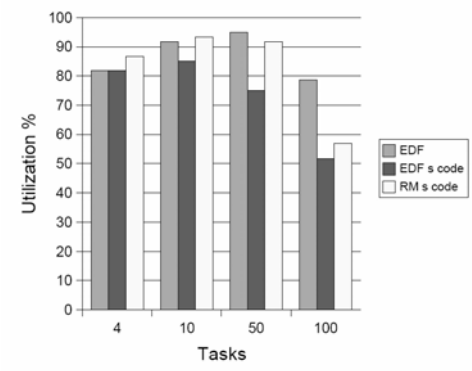

Figure 12. Global CPU utilization [KHS03]

Using the S code the system global utilization is lower and more tasks can be scheduled for additional functionality (see Figure 12).

7.6. Drawbacks

Any technology has both advantages and disadvantages. In the case of SCC with a reduced task set the memory footprint of a table containing the schedule may be smaller than non-optimized S code. Also generating SCC is a non-trivial task and requires advanced compilers and code analysis for time safety checking and then searching for a possible optimal schedule which may be NP hard.

8. References

[ABR93] N. Audsley, A. Burns, K. Tindell, M. Richardson, A. Wellings, *Applying New Scheduling Theory To Static Priority Pre-emptive Scheduling*, Software Engineering Journal, 8(5) (1993) 284-292, September 1993.

[BHR90] S. Baruah, R. Howell, and L. Rosier. *Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor,* Real-Time Systems 2 (1990), pp. 301-324.

[But97] G. Buttazzo*, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer, 1997

[Foh04] G. Fohler, *Flexibility in Statically Scheduled Hard Real-Time Systems*, Dissertation Technische Universität Wien, April 1994.

[GB00] G. Berry, *The Foundations of Esterel*, In C. Stirling, G. Plotkin and M. Tofte editors: Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, 2000

[GS03] Gerald Stieglbauer, *Model-based Development of Embedded Control Systems with Giotto and Simulink*, Master Thesis, Institute of Computer Science, University of Salzburg, April 2003.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, *The synchronous dataflow programming language Lustre,* Proc. of IEEE 79(9), 1991

[HHK01] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. *Embedded Control Systems Development with Giotto*. In Proc. SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, ACM Press, 2001.

[HHK01G] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. *Giotto: A Time-triggered Language for Embedded Programming*. In Proceedings of the First International Workshop on Embedded Systems, ACM Press, 2001.

[HK02] 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.

[HKM02] 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.

[HKM03] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic, *Schedule Carrying Code*, EMSOFT, 2003

[Hor03] Benjamin Horowitz, *Single-mode, single-processor Giotto scheduling*, Report No. UCB/CSD-03-1238, EECS University of California Berkeley, April 16, 2003

[JS93] Kevin Jeffay, Donald L. Stone, *Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems*, Proceedings of the 14 IEEE Real-Time Systems Symposium, pages 212-221, December 1993

[KHS03] C. M. Kirsch, T. A. Henzinger, M. A. A. Sanvido, *A Programable Microkernel for Real-Time Systems*, Technical Report CSD-03-1250, UC Berkeley, 2003.

[Kir02] C. Kirsch: *Principles of Real-Time Programming*, EMSOFT 2002

[Kir02s] C. Kirsch: *Principles of Real-Time Programming*, EMSOFT 2002, presetation slides.

[Kop93] H. Kopetz and G. Grünsteidl. *TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems.* In Proceedings of the 23rd International Symposium on Fault-Tolerant Computing, pages 524–533, 1993.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*., Kluwer, 1997.

[LL73] J. Layland and C Liu, *Scheduling algorithms for multiprogramming in hard real-time environments*, Journal of the ACM, January 1973.

[Nec97] G. C. Necula, *Proof-carrying code*, In Proc. Principles of Programming Languages, 1997

[OSE] http://www.ose.com/prodserv/Default.asp. OSE Systems.

[OSEK] OSEK/VDX Operating System Specification, Version 2.2, September 2001

[Tin92] K. Tindell, A.Burns and A.J.Wellings, *Allocating Real-Time Tasks: An NP-Hard Problem made Easy*, Journal of Real-Time Systems, Vol 4 pp145-165, 1992

[Tin94] K. Tindell, J. Clark, *Holistic Schedulability Analysis for Distributed Hard Real-Time Systems*, Microprocessing and Microprogramming, volume 40, pages 117–134, 1994.