WELL-BEHAVED APPLICATIONS ALLOW FOR MORE EFFICIENT SCHEDULING

HO. TRUTMANN

Computer Engineering and Networks Laboratory TIK, Swiss Federal Institute of Technology ETH, CH-8092 Zurich, Switzerland. E-Mail: trutmann@tik.ee.ethz.ch

Abstract. Current trends in embedded applications aim towards implementations with sets of interoperating tasks and elaborate scheduling schemes. The quest for flexibility and universal applicability results in higher resource demands and reduced predictability. If the functional and connection problems are handled separately with the aid of appropriate tools, subsequent code generators can convert design specific issues, such as parallel structures in the functional part and the various periodicities in the connection part to linear and deterministic pieces of code. Such implementations do away with complicated run-time scheduling, make better use of the available resources and can be analyzed exhaustively before installation on a target.

Key Words. Embedded Systems; Specifications; Scheduling; Code Generation

1. INTRODUCTION

Embedded applications used to be tightly interwoven with their run-time environment. Increasing complexity, together with growing requirements and restrictions has favored the tendency to isolate problem concerns in this field, leading to more general results. A collection of singular tasks replaces monolithic programs with the advantage of smaller scale and simpler structures. Tailor-made scheduling for each application has given way to more generic approaches. Modularity and dynamic behavior at the task level should permit system compositions involving new parts as well as legacy code. The drawbacks of these techniques in hard real-time systems are countered with more efficient hardware in the case of speed considerations and testing in the case of unpredictability concerns. In the more cost-sensitive embedded systems segment the claim for additional resources is not well received, because it is mandatory that these systems use the slowest possible processor and a minimum of memory. For these applications progress made in hardware development must result in lower cost or reduced power consumption, rather than compensate implementation deficiencies. Predictable response times, vital in safety-critical systems on the other hand, are hard to achieve without static methods and using components with deterministic properties.

1.1. Current Implementation Techniques

Increasingly, embedded applications that incorporate hard real-time requirements are multi-task implementations scheduled with static or dynamic techniques, the main justification being the possibility to accommodate any number of quasi-parallel tasks on a single processor.

Dynamic scheduling deals with tasks in an arbitrary sequence and unknown start times, where the various tasks operate independently and where interactions are not clearly defined (Liu *et al.*, 1973). Response time requirements are mapped to scheduler priorities, an inadequate measure for systems with non-periodic events. Solving the interaction problem is postponed and these issues are handed down to a later design stage (i.e., run-time). All types of dynamic scheduling are punished with considerable overhead aggravated by conceptual difficulties such as resource depletion and priority inversion; their essential shortcomings with regard to satisfying timing constraints have also been shown (Xu *et al.*, 1993). This is especially true for preemptive methods which create an infinite number of possible schedules for a given problem and introduce additional conflicts concerning resource usage.

Static scheduling on the other hand uses detailed prior knowledge of the tasks' processing requirements to find a feasible schedule (Xu *et al.*, 1993). While this technique can guarantee the correctness of its solutions, it is overly pessimistic regarding asynchronous events which tend to be scheduled too often and therefore waste processing power.

Even combinations of these techniques cannot entirely do away with this disadvantage, as truly asynchronous tasks still have to be scheduled many times only to verify that the associated event has not yet occurred. In both these approaches input and output activities are implemented as tasks and scheduled in the same way as computation. Generally (except for exclusion and precedence relations) no assumptions about the nature of the tasks actually running on a target are made. If such knowledge is available and if the tasks' code confirms to certain limitations, a combination of static and dynamic scheduling techniques is possible that allows for high processor utilization, at the same time guaranteeing low input jitter and predictable behavior. The overhead of a complicated scheduler is avoided, because there are only three basic tasks. No preemption is used and the remaining dynamic decision involves a mere table lookup which hardly introduces any overhead.

1.2. Code Generation

In control system development the most unpleasant problems appear after work on the implementation has begun, when it may be discovered that a design is either wrong, impractical or that alterations to the original intentions need to be made because of implementation constraints. If changes are made at this stage, it is very difficult to keep the design and the implementation synchronized. An obvious solution would be the automatic compilation of a specification into optimized code, thus uniting specification and implementation.

If generated code is to replace manually written implementations altogether, the generators have to meet a number of difficult demands: small code size, fast execution, adaptation to vastly different hardware and operating environments, to just name a few. Adjustments to specific environments should be done automatically from the same specification and tests created in the design phase must remain valid also during the implementation phase. These demands are best met with dedicated models and tools for the two basically unrelated problem concerns in embedded systems:

- correct reaction to events in the environment,
- information transport between this environment and the control system.

If these prospects are exploited, the resulting code can also be made compliant with more restrictive implementation demands. Parallel structures present in the design are implemented by casting them into a single thread with processor loads as uniform as possible and deterministic behavior. With only deterministic components, exhaustive analysis prior to installation becomes possible.

2. PROBLEM DECOMPOSITION

Specifying a system implies creating its abstract model with the purpose of the model defining the abstraction rules. The separation of the three basic activities in control applications, *input, processing* and *output* into two main problem areas permits independent development and also sheds some light on the objective of possible models. The *functional problem* comprises all the behavioral aspects of a control system. The interactions with the controlled processes themselves are accomplished via shared phenomena (events and actions in an event driven environment). The appropriate abstraction for solving this problem is to leave out any details not pertaining to the relation between the course of events in the environment and the inner states of a model. The *connection problem* unites input, the extraction of events contained in the new data, the calling of the functional reactive block, and the output. The functionality of the connector that must be built is implementation-oriented with a regular sequential structure, so that a useful abstraction would concentrate on information transport within a generalized form. Based on these precepts problem-oriented development methods with separate solutions for the functionality and the connections to the physical processes can be built.

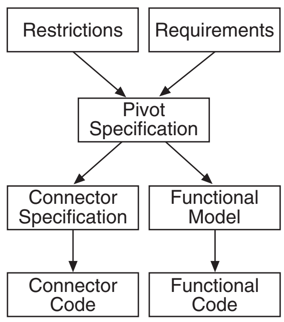

Fig. 1. The design pivot

It is crucial to this approach that the problem areas remain well separated during the entire development. Aside from providing guidelines in the relevant problem areas, architectural restrictions and functional requirements also give rise to an explicit interconnection model, the *pivot specification.* Because it is less prone to frequent changes than the problem areas themselves, development may progress independently behind stable interfaces based on this specification.

2.1. Functional Problem

The functional reactive behavior of a system is modeled with CIP Tool in the form of extended cooperating state machines (Fierz, 1999, and CIP Tool, 1995-99). Control systems in a single- or multiprocessor environment are built with a notation of synchronous processes and asynchronous clusters. Event-based and hybrid systems with integral quasi-continuous processes can both be dealt with. Event messages from the environment cause a transition in the receiving process which activates other processes for his part. Transitions may generate pulses to other processes as well as action messages for the environment, and they can also contain additional code for data manipulation and control functions. A chain reaction initiated by a message cannot be interrupted; its execution depends on the message itself and the internal states of all activated processes.

Consider a simple system with one cluster embodying three processes that model control for an electric motor. It accepts *closeSwitch, openSwitch* (Switch Events), *standstill, sample* (Controller Events) and produces *MotOn, MotOff, MotValue* (Controller Actions), *GreenOn, YellowOn, RedOn, LightOff* (Indicator Actions). The permissible traces in the input stream are determined by the structure and interoperation of the receiving processes; these and other particulars that are of no concern for the generated code will not be considered any further.

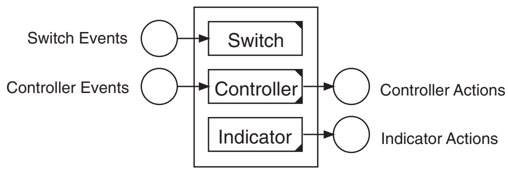

Fig. 2. Motor controller system interface

The transition structure of the controller process (Fig. 3.) shows the motor being either in state *off, running* or *stopping* (the motor runs and stops under continuous control). Its transition triggers are the *start/stop* pulses from the switch process, the *standstill* event and the periodic *sample* event for the controllers from the environment. Its outputs consist of pulses *running*, *stopping*, *stopped* to the other processes and actions *MotOn, MotValue, MotOff* to the environment.

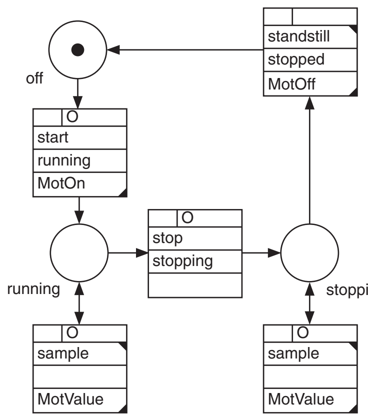

Fig. 3. Controller process

Two different controller algorithms are placed within the transitions triggered by the *sample* event and associated with states *running* and *stopping,* respectively. It follows from the diagram that only one of these controllers can be active at any time, thus effectively limiting the required processing requirements. The transitions leading to these two states contain the initialization parts for the controller algorithms.

All process interactions are based on a specified net (Fig. 4.) which in comprehensive situations also helps understand system dependencies. It does not suffice, however, as a foundation for the later generation of deterministic code.

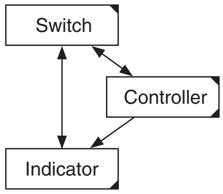

Fig. 4. Motor controller interaction net

To this effect static cascades must be defined that state all possible pulse propagations. In Fig. 5. the cascades for the various processes in the example are shown with the loop *switch-controller-indicator-switch* that is present in the interaction net unraveled. Because the indicator process does not accept events, its cascade is empty.

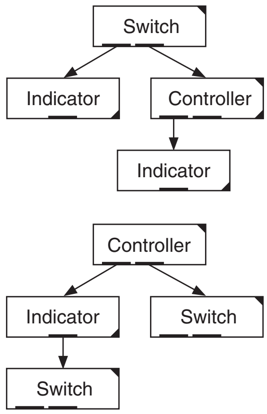

Fig. 5. Motor controller cascades

Code Generation. From a complete model C code can be generated. Along with the modules that implement the functionality, auxiliary code for testing and analysis is provided. One such component is used to verify that and how well an implementation will fit on a specific target. Each specified event sits at the root of possible paths through its pertaining cascade for which a collection of code sequences (minus the action messages) is generated. The worst-case execution path included in this collection must be found by measuring processing times on the target itself. Another generated auxiliary component lists only the sets of action messages belonging to each event with reference to the same possible paths. These will be used to determine upper bounds on the processing needs of the output task.

2.2. Connection Problem

Interfacing functional code to the existent heterogeneous peripheral hardware is a difficult task in itself. Updating input values or the opposite, setting new output values, seem straightforward duties which must be executable within regular structures. On the other hand, input and output operations on some devices are fairly complicated and must be made with multiple accesses, possibly also requiring delay periods in between. In addition, considerable variations with regard to data width, channel number, multiplexer setup and possibly filter and amplifier settings must be accounted for. In present-day systems it is often not possible to exploit all the features of a hardware device, because it would be too tedious to program and impossible to master all potential side-effects. Consider a 16 channel A/D-converter with the possibility to set parameters together with the number of the channel one wishes to convert. Filter and amplifier settings must be issued ahead of the conversion to allow for signal stabilization, i.e., the current setting is influenced by the previous one, introducing dependencies between channels not otherwise related. In practice, this situation is often avoided by using the same settings for all 16 channels and enabling the round-robin conversion option many devices offer.

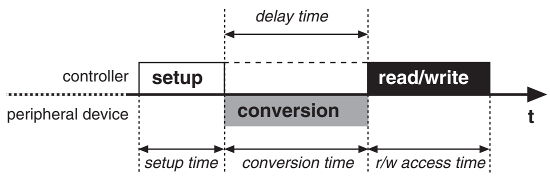

Fig. 6. Device access sequence

Selective specific multi-stage accesses combined with delays as shown in Fig. 6. offer a potential for increasing performance, with the conversion delay unfortunately presenting a dilemma, because the required delay times usually are too short to justify rescheduling at the task level. These delays thus remain periods where the processor idles until the next read or write operation is allowed.

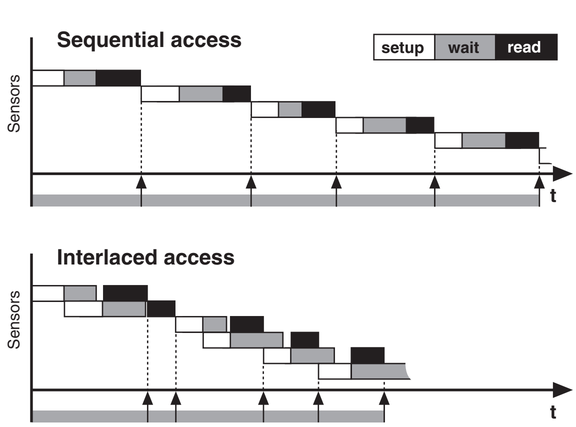

Fig. 7. Optimizing device accesses

Fig. 7. illustrates the benefits of an alternative solution, where the access routines of several devices are interleaved and the processor idle time minimized. This clearly is not a convenient method for manual implementations, but fortunately the affinity to general scheduling problems make it suitable for code generation.

Input. Next to minimizing absolute processor loads the objective of an input system is to distribute work loads evenly among all invocations, a demand even more difficult to fulfill manually. The input task is itself a collection of tasks with precedence and exclusion relations. A feasible schedule is found statically and leads to a single sequential structure. The scheduler algorithm eliminates possible idle times and ensures practically equal processor loads for each invocation.

A textual specification containing the timing requirements for all input signals expressed in terms of the highest timing requirement must be given. Only periodic inputs can be dealt with, and it is desirable to reduce the different timing requirements to just a few frequency classes (e.g. F_{high}, F_{med}, F_{low}). Low level functions (or macros) for setup, access, conversion, verifying and filtering along with their relevant profile data (execution times and required delay periods) that were measured previously on the target are also required. The schedule, consisting of a switch-statement with F_{high}/F_{low} cases, is then generated heuristically. The structure of the specification follows the hardware architecture and therefore also serves as an adequate and up-to-date documentation. In the created schedule the access time of one signal is used to guarantee the necessary delay times for another signal, according to Fig. 7. If no more access routines are available in a slot or where they cannot be used because of exclusion relations (when signals must be acquired by the same input device) delays are inserted into the schedule. Unsatisfactory schedules and wasted processor time reflect deficiencies at the hardware architecture level.

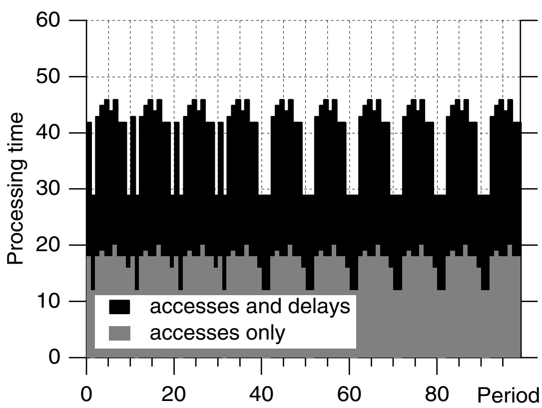

Fig. 8. Processor utilization 42%
(15 input signals, class high: 2, med: 9, low: 4)

In the example in Fig. 8. (with a frequency class relation $F_{high} : F_{med} : F_{low}$ of 1 : 10 : 10) a small collection of input signals with unfavorable exclusion relations causes the accumulation of delay time, resulting in very low processor utilization (42%). A more advantageous setting with five times as many signals including the original ones makes the required processing time rise a mere 30%, practically eliminating the idle delay time. The example also shows that the input task presents a very uniform processor load.

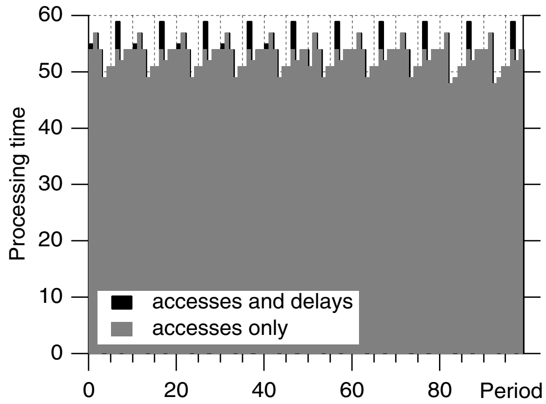

Fig. 9. Processor utilization 99%
(75 input signals, class high: 8, med: 39, low: 28)

Event extraction. Events required to drive the functional reactive block will be extracted at run-time from the newest input data. To generate the tables used by the extractor algorithm another specification must be supplied which contains the conditions for each event. For discrete signals this implies binary patterns, for events buried within continuous signals limit values with an added hysteresis are used. If events shall be extracted according to priorities at run-time, an arbitrary number of priority lists can be supplied in the specification. The currently active scheme will be chosen from within the functional model, where all information regarding system status and desired behavior is available.

Output. Contrary to the input, system output load cannot be distributed statically, as the number of outputs depends on the specific transition structures encountered in a cascade as well as on the occurrence of multi-stage hardware accesses. Stubs for calling the conversion and/or expansion routines from the functional model are generated using the same hardware-oriented specification as for the input schedules, the profile data in this case also including information on access expansions if required. With this information together with the action sets from the functional model (see end of section 2.1.) upper bounds on the processing demands of the output task can be found. A static table with the relevant values is computed to be used by the run-time scheduler.

3. IMPLEMENTATION

The code generators yield code that can be implemented as three distinct tasks either together on a single processor or separately on different processing entities. On a single processor they operate in sequence in every period. The input task with its uniform execution time on each invocation presents a constant load, while the time demands for the processing and the output tasks depend on the occurring events and hence the cascades that must be run through. The performance measurements are used to check if the code can run on a certain target and they will also be needed for run-time decisions.

3.1. Performance analysis

Processing time is present neither in the functional model nor in the specifications for the connector. It is only by examination of the generated code that the performance on a specific target can be evaluated. Because this code is deterministic, the procedure gives the necessary assurance that an implementation does perform according to the requirements. If bottlenecks are detected, a way out can be sought by distributing the work load onto different processors, or by softening the requirements on some events. In this case the measured data together with the event priorities permit quantitative statements regarding the performance degradation.

The time demands for processing the events in the motor controller example in Fig. 10. show that aside from the periodic controllers which are triggered by the *sample* event and must execute first in every period, only a single other event can be handled during the same period, if the worst case paths are taken. Considering the time scale and the improbability of these events arriving at the same time, this may still be acceptable.

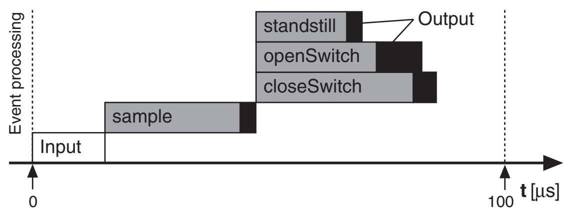

Fig. 10. Motor controller processor time demands

3.2. Run-time scheduling

When the input task yields the processor, the arrival of a new set of input data in the processing task is interpreted as a time event and used by the event extractor to generate the *sample* event for those quasi-continuous processes which run at the highest frequency available in this system. The processing task keeps track of the current time and also records the time demands of all pending outputs, i.e., the ones associated with ongoing event processing. Whenever a new event has been extracted, the expected processing time from the table is compared to the remaining time balance of the current period. If it conforms, the event is fed to the reactive block and the procedure repeated until an event occurs whose execution would not fit any more. It is important to note that in this case the event is withdrawn and not put in a queue, because this particular event may not be present in the input data the next time the extractor runs (this characteristic, combined with sufficiently low sample frequencies can also be used as low-pass filter, e.g., for input key debouncing). Not having a queue for events at the same time effectively prevents processor overloads and is also a prerequisite for switching between different event priority lists.

In the period shown in Fig. 11. processing for the event *standstill* would not fit into the remaining time budget in the current period and is therefore rejected. This type of scheduling is achieved with practically no overhead and it is worthwhile because worst-case execution times are hardly ever exhausted, so that in reality more events can be dealt with than predicted by the static analysis. The slack time nevertheless remaining can be exploited by non real-time tasks, which themselves must be preemptable.

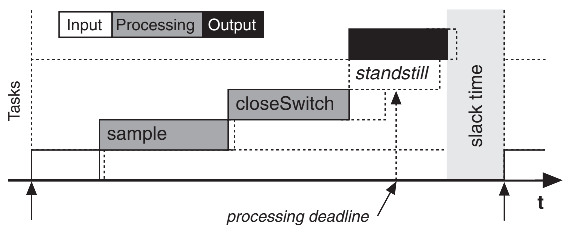

Fig. 11. Task schedule

Output from the processing task is fed to buffers to allow for possible asynchronicities of the output system. The majority of output signals results in a single hardware access, but sometimes expansion to several accesses with time lags in between may be necessary. For this reason two different buffers are provided that may or may not be located on the same processing entity. In the case of a simple output signal, a plain FIFO buffer is used, while for multi-stage accesses an ordered queue with relative delays is necessary. These buffers are read in a round-robin fashion, either until they are empty, a new timer tick occurs or, in case of the second buffer, a first element is encountered with a delay time that has not yet elapsed.

Jitter. Algorithms for quasi-continuous periodic control tasks are sensitive to time variations concerning their input data. This input 'jitter' is usually difficult to control with priority-based or even preemptive schedulers. Because a controller's input data is sampled at precisely the same instant in each invocation, and because the periodic *sample* events are always processed first, jitter is reduced to inaccuracies inherent in the clock hardware (if no other interrupts interfere). Asynchronous events on the other hand may have tight response requirements, but are insensitive to jitter.

4. SUMMARY

Introducing parallel entities into the design of control systems is a well established method when the focus is on specific aspects in a complicated system. While the idea of parallelism is helpful for disentangling complex problems in a conceptual stage, there is hardly any reason for the common practice to implement such a solution using different tasks when they all run on a single processing entity. If the conceptual parallelism cannot be exploited by parallel hardware, it is advantageous to replace it with a single corresponding sequential task. While this may not be easy or even feasible for a set of parallel tasks written in a programming language, it poses no problems for a code generator. Input and output generally use common hardware resources that often impose sequential use. With specifications and code generators efficient solutions for hardware access become possible. Provided that deterministic code results from these efforts, what remains to be done is verifying that the available processing time is sufficient; there is no need for scheduling more than necessary!

The presented approach has been shown to scale well in a number of real-world applications, some with more than 300 I/O signals, sub-ms response requirements and distributed implementations.

5. REFERENCES

CIP Tool - User Manual (1995-99). *CIP System AG.* Solothurn.

Fierz, H. (1999). *The CIP Method: Component- and Model-based Construction of Embedded Systems.* Submitted to European Software Engineering Conference ESEC'99.

Liu, C. L. and J. W. Layland (1973).*Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, 20:46-61.

Trutmann, HO. (1998). *Generation of Embedded Control Systems.* WRTP'98. Shantou.

Xu, J. and D. L. Parnas (1993). *On Satisfying Timing Constraints in Hard-Real-Time Systems.* IEEE Transactions on Software Engineering 19(1):70-84.