Real-Time Multi-Tasking in Software Synthesis for Information Processing Systems*

Filip Thoen, Marco Cornero†, Gert Goossens and Hugo De Man

IMEC, Leuven, B-3001, Belgium
† SGS-Thomson Microelectronics, Crolles, 38921, France

Abstract

Software synthesis is a new approach which focuses on the support of embedded systems without the use of operating-systems. Compared to traditional design practices, a better utilization of the available time and hardware resources can be achieved, because the static information provided by the system specification is fully exploited and an application specific solution is automatically generated.
On-going research on a software synthesis approach for real-time information processing systems is presented which starts from a concurrent process system specification and tries to automate the mapping of this description to a single processor. An internal representation model which is well suited for the support of concurrency and timing constraints is proposed, together with flexible execution models for multi-tasking with real-time constraints. The method is illustrated on a personal terminal receiver demodulator for mobile satellite communication.

1 Introduction

The target application domain of our approach is advanced real-time information processing systems, such as consumer electronics and personal communication systems. The distinctive characteristic of these systems is the coexistence of two different types of functionalities, namely *digital signal processing* and *control functions*, which require different timing constraint support. Specifically, signal processing functions operate on sampled data streams, and are subject to the *real-time* constraint derived from the required sample frequency or throughput. Control procedures vary in nature from having to be executed as soon as possible (like e.g. a man-machine interface), but an eventual execution delay does not usually compromise the integrity of the entire system (*soft deadline*), to having very stringent constraints, like e.g. a critical feedback control loop (*hard deadline*).

Traditionally, real-time kernels, i.e. specialized operating systems, are used for software support in the design of embedded systems [5]. These small kernels, often stripped-down versions of traditional time-sharing operating-system, are in the first place designed to be *fast* (e.g. fast context switch). Above all, real-time kernels provide the run-time support for *real-time multi-tasking* to perform software scheduling, and primitives for inter-process communication and synchronization, and for accessing the hardware resources. Since processes are considered as black boxes, most kernels apply a *coarse grain* model for process scheduling. Most kernels tend to use a fixed priority preemptive scheduling mechanism, where process priorities have to be used to mimic the timing constraints. Alternatively, traditional process scheduling approaches use timing constraints, specified as process period, release time and deadline [11]. From the designer viewpoint however, these constraints are more naturally specified with respect to the occurrence of observable events. Moreover, the scheduler has no knowledge about the time stamps when the events are generated by the processes, and consequently can not exploit this. Assignment of the process priorities, as in the case of the fixed priority scheduling scheme, is a *manual* task to be performed without any tool support. Typically, an iterative, error-prone design cycle, with a lot of code and priority tuning, is required. Not only is this inflexible and time consuming, but it also restricts the proof of correctness to the selected stimuli. Additionally, the behavior of the scheduler under peak load conditions is hard to predict, resulting often in under-utilized systems to stay on the safe side. It is safer to guarantee timeliness pre-runtime, as new family of kernels tend to attain [5]. Moreover, kernels trade optimality for generality, causing them to be associated with run-time and memory overhead.

Software synthesis [1][2][7] is an alternative approach to real-time kernels: starting from a system specification, typically composed of concurrent communicating processes, the aim of software synthesis is the automatic generation of the *source code* which realizes 1) the specified functionalities while satisfying the timing constraints and 2) the typical run-time support required for real-time systems, such as multi-tasking, and the primitives for process communication and synchronization. A better utilization of the available time and hardware resources can be achieved with software synthesis, because the static information provided by the system specification is fully exploited; as a consequence the automatically generated run-time support is customized for and dedicated to each particular application, and does not need to be general, as in the case of real-time kernels. Moreover, an accurate static analysis provides an early feedback to the designer on the feasibility of the input specifications. In this way the iterative design cycle typical for real-time kernels is avoided, and satisfaction of the timing constraints can be guaranteed automatically. Besides, the transformations and optimizations envisioned in the software synthesis approach, try to automate this code tuning. Finally, since the output of software synthesis is source code, portability can be easily achieved by means of a retargetable compiler [6].

* This work was supported by the European Commission, under contract *Esprit-9138 (Chips)*

The software synthesis approach in the VULCAN framework [7] allows to specify latency and rate timing constraints. *Program threads* are extracted from the system specification, in order to isolate operations with an unknown timing delay. A simple non-preemptive, control-FIFO based run-time scheduler alternates their execution, but provides only a restricted support for satisfying these constraints, since threads are executed as they are put at run-time in the FIFO and are not reordered. Moverover, interrupts are not supported, due to the choice of the non-preemptive scheduler.
The approach taken in the CHINOOK [2] system suffers from a similar restriction: although preemption is allowed based on the watchdog paradigm, resuming at the preemption point is difficult, and hence interrupts are not supported. The system, targetted towards reactive control systems, only supports timing constraints on state transitions and on latency between operations. No rate constraints are supported, as is typical for DSP applications.

The rest of this paper is structured as follows. Section 2 introduces the system representation and the concepts used. In section 3, two different execution models and the steps of a possible software synthesis script are discussed. A real-life illustration of the approach is the subject of section 4. Finally, section 5 draws some conclusions.

2 System Representation - Model

We assume that the target application can be modeled in a concurrent process description, which captures operation behavior, data dependencies between operations, concurrency and communication [8][9]. The precise semantics of such a specification are beyond the scope of this paper. From this specification, a *constraint graph* can be derived that contains sufficient information for the software synthesis problem, as will be introduced below.

We define a **program thread** as *a linearized set of operations which may or may not start with a non-deterministic (ND) time delay operation* [7]. Examples of ND-operations are synchronization with internal and external events, wait for communication and unbounded loops. The purpose of extracting program threads from the concurrent process input specification is to isolate all the uncertainties related to the execution delay of a given program at the beginning of the program threads. Program threads, which can be executed using a single thread of control (as present in most contemporary processors), have the property that their execution latency can be computed statically. Besides being defined by the ND-operations, program threads can also capture concurrency and multi-rate transitions. A new representation model, based on **constraint graphs** [10], is then built up from the extracted threads. This model allows a static analysis, both of the imposed timing constraints and of the thread scheduling. The *vertexes* represent program threads and the *edges* capture the data dependency, control precedence and the timing constraints between threads. Specifically, let $\delta(v_i)$ the execution delay of the thread represented by vertex v_i; a forward edge $e_{i,j}$ with weight $w_{i,j} = \delta(v_i)$, represents a minimum timing constraint between v_i and v_j,

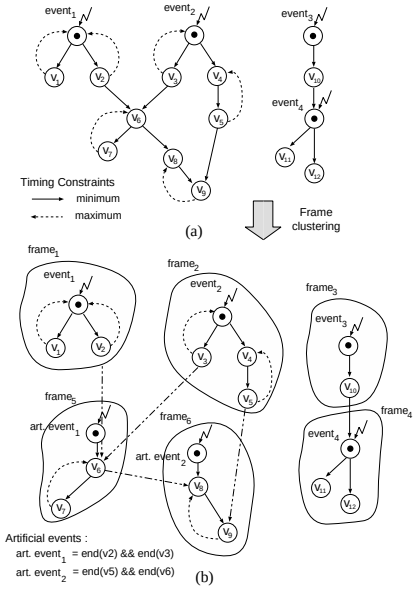

Figure 1: Example of a Constraint Graph

i.e. the requirement that the start time of v_j must occur *at least* $w_{i,j}$ units of time later than v_i. Similarly, a *maximum* timing constraint between two threads v_i and v_j is indicated as a backward edge with negative weight $w_{i,j}$, representing the requirement that the end time of v_i must occur no later than $\mid w_{i,j} \mid$ units of time later than the end time of v_j. Finally, *ND*-operations are represented by separate **event nodes**. An example is given in figure 1 (a).
Our model differs from [10] in the abstraction level of a CG node: in our approach a CG node isolates a group of operations which corresponds to static program parts, while in [10] individual operations are the CG entities. Moreover, in [10] a CG is restricted to being a single connected graph, not able to capture process concurrency. This restriction is lifted in our approach and internal events are introduced to synchronize between concurrent graphs capturing process concurrency. Also, we support multi-rate by placing relative execution rate numbers on control edges.

By definition, all the uncertainties related to the timing behavior of a system specification are captured by event nodes. Since the arrival time of an event is unknown at compile time, event nodes limit the extent of analysis and synthesis which can be performed statically.
In a second step, threads are clustered into so-called **thread frames** (figure 1 (b)). The purpose of identifying thread frames is to partition the initial constraint graphs into disjoint clusters of threads triggered by a single event, so that static analysis and synthesis (e.g. scheduling) can be performed for each cluster *relatively* to the associated event. Remark that sequence edge(s) can exist between frames according to the original system specification.
The *event set* $E(v_i)$ of a node v_i is defined as the set of event nodes which are predecessors of v_i. Artificial events are introduced for threads with an event set which contains at least two elements between which there does not exist a path in the graph. These events are in fact internal events, which

must be observed and taken care of by the execution model in a similar way as the external events which are triggered directly by the environment.
The execution model will take care of the activation at run-time of the different thread frames according to the original specification while taking into account the sequence of occurred events and the imposed timing constraints. In this way the unknown delay in executing a program thread appears as a delay in scheduling the program thread, and is not considered as part of the thread latency.

3 Execution Models and Implementation

In this section the execution models and the implementation, i.e. the mapping of the representation model to the single thread of control of the target processor, are described. Although the CG model is target independent, in this paper we focuss on a single processor target.

3.1 Execution Models

Blocking Model - Cyclic Executive Combined with Interrupt Routines A simple, but cost effective solution for the run-time thread frame activation consists of using a simple *event loop* in background combined with tying different thread frames to processor interrupts. The assignment of frames to the event loop and the (internal) scheduling of the frames is done at compile-time. The event loop in the background polls in a round-robin fashion the occurrence of the events triggering the different thread frames and accordingly starts executing the appropriate frames sequentially. Processor interrupts present a cheap way, supported by hardware, to asynchronously start up thread frames which suspend the currently executing frame. The processor interrupt masking and priority levels can be used to selectively allow interruption of time critical thread frame sections and to favor high priority frames.
Only frames which are triggered by an event corresponding to interrupts (either the external hardware or the internal peripheral interrupts) can be started up asynchronously, while the other frames are to be placed in the background event loop. Moreover, a background frame started up by the event loop will block the processor till the end of its execution preventing other frames in the event loop to be started up. Hence, the name "blocking execution model". The execution length of the frames limits the response time of events in the event loop, and therefore limits the scope of this model.

Non-blocking Model using a Run-time Scheduler Figure 2 (a) outlines the execution model which takes a two-level scheduling approach. *Static* scheduling (i.e. at compile-time) is performed after thread clustering to determine a relative ordering of threads *within* each thread frame, and by assumption this ordering is not changed anymore in the dynamic scheduling phase. At *run-time*, a small *preemptive* and *time-driven* scheduler takes care of the composition and interleaving of the different thread frames according to the system evolution and the timing constraints. Additionally, it tries to avoid active waiting by scheduling other frames when the next frame is not ready to be executed, in this way maximizing processor utilization.
This run-time behavior is illustrated in the lower part of figure 2 (a): starting from an idle state, suppose that $\texttt{event}_1$ occurs; this event activates the run-time scheduler, and since no other frames are currently active, the threads of the first frame are executed with the order determined previously with static scheduling (order 1-3-2). Occurrence of $\texttt{event}_2$, while executing thread_3 of the first frame, causes the following actions: 1) thread_3 is interrupted; 2) the run-time scheduler is invoked for determining the subsequent execution order, in the example: A, rest of thread_3, B, 2; and 3) execution proceeds with the newly determined thread ordering. As indicated the *relative ordering* between the threads of the same frame is not changed allowing an efficient implementation of the run-time scheduler, which must be necessarily very fast.

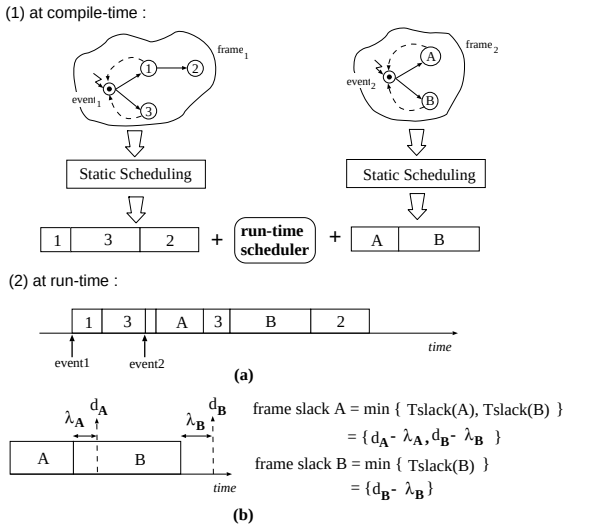

Figure 2: The run-time execution model (a) and the *frame slack* scheduling metric (b)

The scheduling metric used by the run-time scheduler is the *frame slack* time. This information is derived statically based on the imposed timing constraints and on the relative thread ordering within each frame. The frame slack indicates the amount of time the end of an individual thread in a thread frame can be postponed, relatively to its static schedule, before violating a timing constraint. As illustrated in figure 2 (b), the frame slack is defined as the minimum of all *thread slacks*, i.e. the remaining time between the end of the thread and its timing constraint, of all the succeeding threads in the static schedule. The frame slack derived at compile time, is used and updated at run-time. For a more formally description of this model, we refer to [3].

3.2 Script

Figure 3 gives an overview of the proposed approach. From the concurrent process specification, the different program threads are extracted and the non-deterministic timing delay is isolated in event nodes. During this step, a code generator can provide a static estimate of the thread execution times. These execution times are placed together with the timing constraints in a constraint graph, the abstraction model used in the sequel of the approach. The assignment of external processor interrupts to event nodes in the constraint

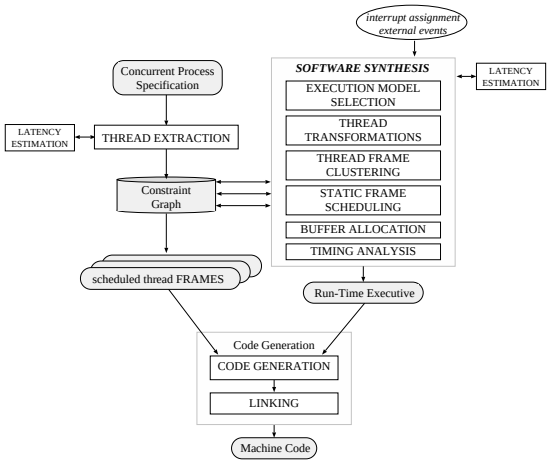

Figure 3: Possible software synthesis script

graph, which is determined by the context of the system, is to be provided by the user.

After selection of one of the two execution models explained above, five tasks, which are phase coupled, have to be performed. *Thread frame clustering* tries to cluster the constraint graph into disjoint groups of threads which are triggered by the same event set. These thread frames are to be activated *at run-time* by the selected execution model. Since events introduce an overhead during frame scheduling, we also want to minimize the number of clusters, without violating timing constraints. *Static frame scheduling* will determine *at compile-time* the relative order of the threads *inside* each of the identified thread frames. Occasionally, the timing constraints can not be met by the identified frames. In this case, *a transformation step* on the threads or the frames can resolve the problem or can provide a more optimal solution. An example of these transformations will be given in the illustration in section 4. *Buffer allocation* will insert the required buffers in between communicating frames, by deriving the buffer sizes from the execution rates of the frames. *Timing analysis* is used in different phases : once upon entry of the tool, to check for consistency of the user specified timing constraints, and subsequently during the execution of the tool, to verify whether a result of a synthesis task still satisfies all constraints.

The outcome of software synthesis are scheduled thread frames and possibly (depending on the execution model chosen) a small run-time executive, which activates the appropriate frames at run-time; both have to be compiled with the code generator and linked together afterwards.

4 Illustration of the Approach

System Description - Concurrent Communicating Process Specification Figure 4 outlines the process specification of a mobile terminal receiver *demodulator* to be used in the MSBN satellite communication network [4]. This network allows a bi-directional data and voice communication in a star network consisting of a fixed earth station and multiple mobile stations. Two different data channels, called *pilot* and *traffic* channel, are sent over on the same transmission carrier using the CDMA technique, i.e. correlating the channels with orthogonal pseudo-noise codes enabling them to use the same frequency spectrum without interference. The former channel carries network system information (e.g. average channel bit error rate), the latter carries the actual user data. Acquisition and tracking of the transmission carrier is performed on the pilot channel in cooperation with an intelligent antenna.

Triggered by an external interrupt, the `read_decorr` process reads periodically (at a rate of 3.4 kHz) the memory mapped decorrelator FPGA. This process sends data to the `track_pilot&demod` and the `traffic_demod` processes, which perform the tracking of the transmission carrier and the demodulation (i.e. gain, carrier phase and bit phase correction). After a 1:3 rate conversion the demodulated traffic data is formatted by the `traffic_manage_data` process and via the `send_vocoder` process transmitted to a second, memory mapped processor. In contrast, the demodulated pilot data will be further processed on the same processor.

The `track_pilot&demod` process not only delivers its demodulated data to the `pilot_manage_data` process, it steers the frequency of the NCO *(numerical controlled oscillator)* in the preceeding analog demodulation part through use of the on-chip *serial peripheral*. Moreover, together with `traffic_demod` process it sends information concerning carrier synchronization to the `display_LEDs` process and `write_antenna` process. The channel decoding of demodulated pilot data is carried out by the `pilot_DSP_functions` process, which operates on a 1024 element *frame* basis, so a multi-rate transition is present between the `pilot_manage_data` and this latter process. The output data of the pilot channel decoding is sent to a PC computer using the on-chip DMA engine. The `setup_DMA` process is triggered when output data is available from the `pilot_DSP_functions` process and sets up and starts the DMA process.

Asynchronously with this chain of periodic processes, the `read_sys_cmd` and `read_antenna` process control the internal parameters of the demodulation processes. They respectively perform the man-machine interface connected to the system using a memory mapped flag, allowing the user to alter the system operating parameters, and the interface with the antenna controller which is connected via an external interrupt. The former is a *sporadic* process, since a user will adapt the parameters only once in a while, and is allowed to have a large response time. The latter is a *time-critical* process: when the antenna controller looses the beam, it will signal this immediately to the demodulator, which must take special re-tracking actions.

Constraint Graph Representation Figure 5 outlines the constraint graph (after thread frame clustering) for the demodulator capturing the threads, their dependency and their timing constraints. For reasons of clarity, the thread execution times are not indicated.

Three event nodes are introduced to capture the timing uncertainty of the periodic interrupt from the decorrelator (ev_{decorr}), the interrupt from the antenna controller (ev_{ant}) and the setting of the polling flag of the man-machine in-

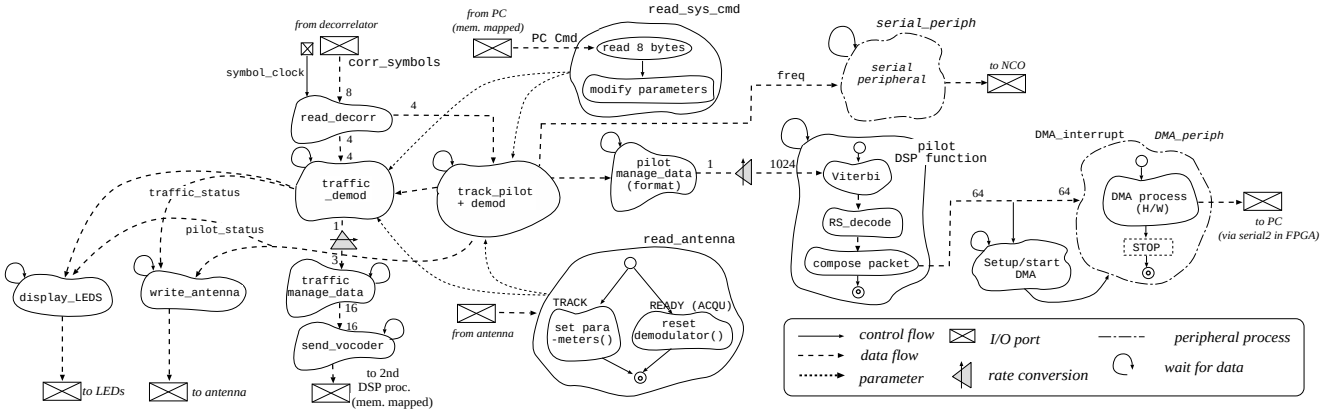

Figure 4: Concurrent Process Specification of the MSBN demodulator

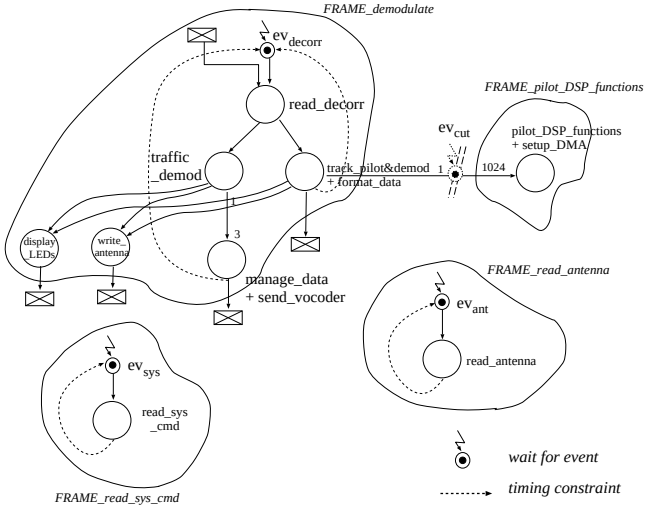

Figure 5: The constraint graph of the MSBN demodulator (after frame clustering)

terface (ev_{sys}). The extra event ev_{cut} was not present in the original CG, but was introduced during frame clustering (see below). Remark that the event nodes make abstraction from whether they are implemented as an interrupt or as a polling loop. Program threads capture also concurrency and the rate conversion. Some processes in the original user specification have been combined into one program thread (e.g. `manage_data+send_vocoder`).

Timing constraints are added as backward edges, e.g. the edge from the `manage_data+send_vocoder` to the (periodic) event node expresses that the end of that thread must be executed before the start of the occurrence of the next event, and thus the next period.

Thread Frame Clustering and Transformations In first instance, the cyclic executive based execution model was tried, which proved satisfactory for this application. *Thread frame clustering* and *transformation* were already added in the CG of figure 5. Four different thread frames are identified, three according to the original events and one by an artificial event introduced by the *frame cutting transformation*. Although the event set of the `pilot_DSP_functions+setup_DMA` thread is the same as e.g. `track_pilot&demod+format_data` thread, it had to be placed in a separate frame because of timing constraints: the execution time of the frame triggered by the decorrelator event in its 1024th execution (according to the relative rate of 1:1024 of the last thread) would become longer than the period of the periodic event and thus conflicts with the timing constraints. Cutting the `pilot_DSP_functions+setup_DMA` frame off and introducing an artificial event ev_{cut} which checks for the 1024th execution of preceeding frame, will allow the execution model to overlap the 1024 executions of the `FRAME_demodulate` with the `FRAME_pilot_DSP_functions`.

Another transformation, called *rate matching* is applied on the `manage_data_send_vocoder` thread: by inserting a rate counter which checks for the relative event occurrence, and based on this causes a conditional execution, the rate of this thread is matched to its frame rate.

Implementation The final implementation after static *frame scheduling* and *introduction of the communication buffers* and with inclusion of the *execution model* is shown in figure 6. Both the `FRAME_read_sys_cmd` and `FRAME_pilot_DSP_functions` frame are placed in the background event loop of cyclic executive based execution model, and thus in a round-robin schedule. For the man-machine frame, this is possible because of its non-stringent timing constraint. Its response time to a user setting new system commands will be limited by the execution time of `FRAME_pilot_DSP_functions`. The two other frames are triggered and activated by the environment using the corresponding hardware interrupt. Remark that in the frame `FRAME_demodulate` the interrupts are unmasked again (the target processor by default did not allow interrupt nesting) to allow interrupt by `FRAME_read_antenna`, in order to reduce the response time to the antenna loosing the carrier. At the bottom of figure 6 the behavior of both the CPU and the processor peripherals are outlined on a time-axis. It can be seen clearly that while the `FRAME_pilot_DSP_functions` frame is processing the previous data frame in background, the `FRAME_demodulate` frame (activated by a hardware interrupt) is processing consecutively the 1024 data samples of the next frame. This can be considered as a kind of *process time-folding*. It can also be seen that in-

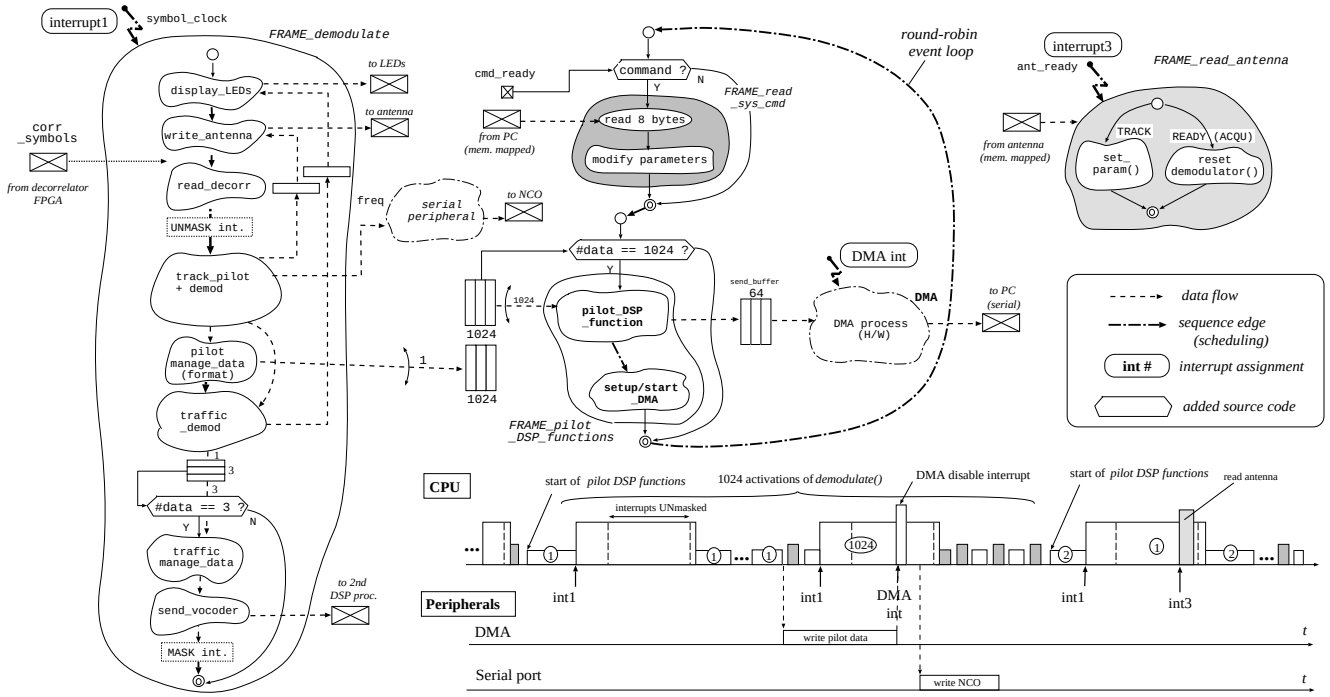

Figure 6: The Final Implementation of the MSBN demodulator

terrupt nesting of both the FRAME_read_antenna frame and the DMA interrupt routine (which disables the DMA engine after transfer completion), can occur in the FRAME_demodulate frame.

Results The overhead implied by the use of the blocking execution model is *minimal*: it only requires an infinite loop and two conditional tests for the background event loop, and a (register) context save/restore (supported by the processor hardware) for each interrupt routine.

This overhead has to be compared the situation when a real-time kernel is used to implement the run-time behavior of figure 4. The original specification consists of twelve concurrent user processes, and when this hierarchy is straightforwardly implemented, using the kernel's semaphore primitives to signal between two tasks when data is available, this will result in a considerable (run-time) overhead compared with the solution proposed above. Additionally, it requires extra program memory to hold the kernel's program code. However, with careful manual tuning of the orginal specification and by collapsing a number of user processes, the kernel solution could approach ours. This tuning is however a manual task in contrast the automated tuning and transformation process in our approach, which works across the division of the specification into processes by the user.

5 Conclusions

The approach in this paper tackles the software support problem at the source level in contrast to contemporary coarse-grain, black-box approaches.

The proposed approach tries to exploit the knowledge of the application at hand, applies transformations and optimizations to the specification and generates an application specific solution, with the automatic support for timeliness. The method, based on a representation model composed of program threads and constraint graphs, features a selectable execution model which combines a detailed static analysis of the input specifications, resulting in a static partitioning of the input specifications and a static schedule for each partition, and a run-time activation for the dynamic composition of the specification partitions.

References

[1] M. Chiodo, et al., "Hardware-Software Co-design of Embedded Systems," IEEE Micro, Vol. 14, No. 4, Aug., 1994.

[2] P. Chou, G. Borriello, "Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems," Proc. DAC-94, Jun., 1994.

[3] M. Cornero, et al., "Software Synthesis for Real-Time Information Processing Systems," Code Generation for Embedded Processors, Kluwer, 1995.

[4] European Space Agency (ESA), "Mobile Satellite Business Network (MSBN) - System Requirement Specification," Issue 3.1, ESA-Estec, Nov. 17, 1992.

[5] K. Ghosh, et al.,"A Survey of Real-Time Operating Systems," report GIT-CC-93/18, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, Feb. 15, 1994.

[6] D. Lanneer, et al., "CHESS: Retargetable Code Generation for Embedded DSP Processors," Code Generation for Embedded Processors, Kluwer, 1995.

[7] R. K. Gupta, "Co-Synthesis of Hardware and Software for Digital Embedded Systems," PhD. Dissertation, Stanford University, Dec., 1993.

[8] N. Gehani, W. D. Roome, "The Concurrent C Programming Language," Prentice Hall, 1989.

[9] IEEE Inc., "IEEE Standard VHDL Language Reference Manual," IEEE Standard 1076-1987, Mar., 1982.

[10] D. Ku, G. De Micheli, "Relative Scheduling Under Timing Constraints," Proc. DAC-90, Orlando, FL, Jun., 1990.

[11] J. Xu, D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations," IEEE Trans. on Softw. Eng., Vol. 16, No. 3, Mar., 1990.