Software Implementation Techniques for Hw/Sw Embedded Systems

J.P. Calvez*, O.Pasquier*, J. Peckol†

* IRESTE University of NANTES, FRANCE

† University of Washington, SEATTLE, USA

Abstract

Our focus in this paper is the software implementation of control oriented systems. Such a task is one of the least automated portions of the contemporary CoDesign process. In systems that must respond to external events, often several asynchronous tasks are implemented on the same processor. We are studying such systems because often, they utilize a dynamic multi-rate scheduling technique using a multitasking real time kernel. Based upon the MCSE functional model as a specification input, we propose a set of transformation rules one can apply to the functional structure to reduce the complexity of the software design prior to implementation. We further show that after such optimizations, the microprocessor interrupt system can often be used as an efficient priority-based scheduler, thereby removing the need for a real time kernel. The resulting implementation is described using a software implementation diagram from which it is easy to prove the timing constraints are satisfied. We use a simplified control system to illustrate our approach and to show a smooth incremental CoDesign path with a better integration of software estimates into the partitioning decision.

1: Introduction

The objective of the CoDesign process is to achieve high-quality designs in less time and at a reduced cost. The process begins with a system-level design that defines the portion of the system for which the performance must be optimized. The first step is to develop the functional and non-functional specifications. From such specifications, designers must decide upon a physical architecture and then proceed with the complete implementation. CoDesign usually consists of two phases: 1)Hw/Sw partitioning and allocation (often a mixture of automatic, semi-automatic, or interactive techniques) and 2) hardware, software and interface synthesis [5].

The state of the art in hardware synthesis permits the complete hardware solution to be implemented using a synthesizable language. By analogy, one can imagine that the software portion could be similarly generated.

Herein, we focus on the software implementation in control oriented systems. Such systems must often respond to external events. Often in such systems, a multitasking solution is selected because of the many asynchronous functions (or processes, or threads). Up to now, most of the research in CoDesign has focused on partitioning, defining the hardware/software interfaces, and on the synthesis of the hardware portion. Research on software side has primarily considered one main thread [10], or the static scheduling of several threads [3],[4],[8],[9]. Such an implementation is too restrictive for systems in which several asynchronous tasks must respond to external events and must be implemented on the same microprocessor.

In this paper, we study the problem of implementing a set of asynchronous tasks on a single microprocessor. We present a technique for efficiently implementing the software portion of embedded systems. The result is a good basis for evaluating software performance and size. Section 2 describes our approach. Section 3 briefly presents an example to describe the specification model from which the design is synthesized using a series of transformations and evaluations. Section 4 describes one implementation technique using a real-time kernel. Section 5 presents a set of optimization techniques for reducing the number of tasks. Section 6 gives an improved implementation without real-time kernel. Section 7 illustrates our proposal in an example. The work is summarized in the last section.

2: Presentation of the method

The final quality of real-time embedded systems depends upon the development process, the description models, and the techniques and tools used. In CoDesign, the objective is to find a global optimal solution that satisfies a given a set of constraints. The design decisions are often interdependent and contradictory. The hardware/software partitioning is an important step and must be based on accurate estimates. One can pose the questions: is it better to start from a skeletal software implementation and move unsatisfiable constraints into hardware or to start with a rough partition and move in both directions? Can we produce an accurate estimation of the software characteristics for partitioning without first defining the software?

Partitioning and implementation activities are highly interdependent. The implementation is produced by tools such as VHDL synthesizers, interface generators or synthesizers, and software generators and compilers. For the software portion, several factors affect the quality of the

0-8186-7895-X/97 $10.00

result. With a multitasking real-time kernel, its size, performance, and cost can have a significant impact on the design. Today, the state of the art in software tools lags the hardware tools. Research into software implementation is becoming increasingly essential.

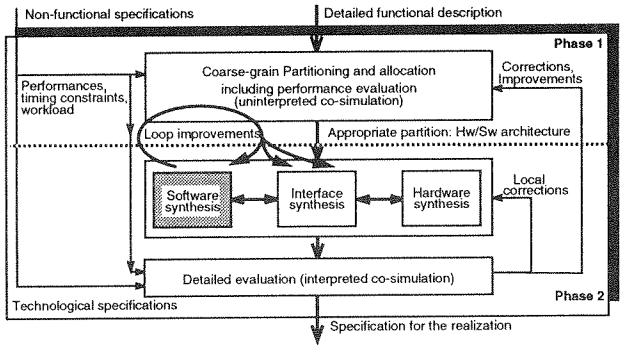
Our approach in this paper considers:

- using MCSE for system design as a CoDesign front-end development process [1],
- using the MCSE functional model as the specification input for the CoDesign activity,
- using interactive and manual coarse-grain partitioning with the help of estimators, [2],
- the tasks on each processor can be implemented independently.

The expected result is an optimized software design on each microprocessor from which accurate timing and size parameters can be evaluated and analyzed.

The CoDesign process is comprised of two phases, in each verification by co-simulation enables one to incorporate corrections or to continue. Our CoDesign flow is shown in Figure 1. The implementation of the functional portion of the software is a part of the synthesis activity.

Partitioning and allocation are based on an interactive coarse-grain procedure. Hardware and software estimations can be used if available. If the grain of the decomposition is too coarse, the process can be continued. Once an appropriate partition is reached, we recommend focusing on the software portion. The hardware and the software are then produced by synthesis. A final verification of the design can be done by co-simulation.



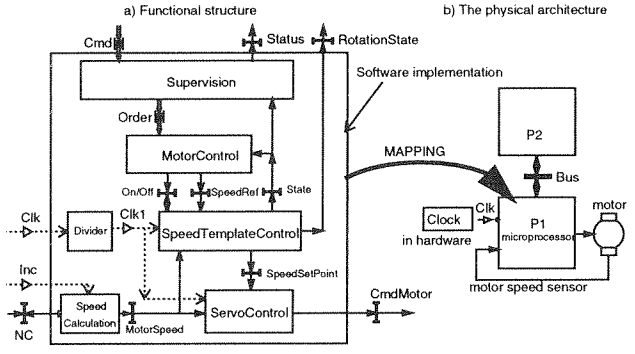
-Figure 1 - The CoDesign process focussing on software implementation.

Our process permits designers to follow a smooth design path with better integration of the software into partitioning decisions. In this way the review cycle becomes shorter. Without such an approach, performance verification is possible only after a full implementation and detailed co-simulation.

3: Recall of the functional model

The specification model recommended in MCSE [1] is a *functional model* describing a system as a set of interacting functional elements. Depicted using a hierarchical, graphical model, functions interact using three types of relations: the shared variable, the synchronization relation, and message transfer. The physical architecture is described with the *executive model* based on physical components and their interconnections. Partitioning leads to a *mapping* between the functional and executive viewpoints. Each micro-processor becomes the software execution resource for implementing the set of functions and relations.

Figure 2 shows the specification model for a simple servo system we will use to illustrate our proposed synthesis technique. On the left hand side is the functional structure that results from the design step, refer to [1]. The physical architecture with two processors is given on the right.



-Figure 2 - Example of a functional specification for the speed control of a motor.

The *Supervision* function receives commands *Cmd* from the application management function (not represented here). Depending on each of the more elementary *Order* messages received from *Supervision*, the *MotorControl* function specifies the command *On/Off* and the speed set point *SpeedRef*. The *SpeedTemplateControl* function generates the speed set point used by the *ServoControl* function to command to the motor (*CmdMotor*) based upon the actual measured speed *MotorSpeed*. These two functions are synchronous to the *Clk1* event. *SpeedCalculation* is necessary for estimating the actual speed from the value *NC* (number of clock periods) and the *Inc.* event generated by a shaft encoder. *Divider* is used to derive the period for *Clk1*. From this functional model, the reader can easily understand the behavior of the design and imagine the algorithmic description of each function. *Supervision* is considered to be a permanent cyclic process. All other functions are temporarily sequential cyclic processes synchronized to their input event (*Clk, Inc, Clk1*) or a message input (*Order*).

4: Implementation with a real-time kernel

The quickest software approach for implementing of a set of asynchronous functions is to use a real-time kernel (RTK). A real-time kernel manages the tasks according to a scheduling policy. The simplest and most frequently used is rate-monotonic scheduling [11].

The semantics of the MCSE functional model make the implementation of such a system very simple. From the functional model, the relative priority of each task is easily defined. Observe that the 3 functional relations have direct procedures in the RTK library. The only procedures needed for a set of static tasks are:

- *Signal(Ev)* and *Wait(Ev)* for event synchronization. A boolean semaphore is used.
- *WriteSharVar(Val, V)* and *ReadSharVar(V, Val)* for the exclusive accesses to the shared variable V. A shared variable is a common resource protected with a semaphore.
- *Send(Mess, Pt)* and *Receive(Pt, Mess)* for the message transfer through the port Pt. Pt is implemented as a fixed-size mailbox or a message queue.

The solution for the example in Figure 2 is clear: *Supervision* and *MotorControl* are low priority tasks, *SpeedTemplateControl* and *ServoControl* have higher priority, *Divider* and *SpeedCalculation* are the highest ones. The scheduling and verification of the timing constraints are also straight forward; the RMA (Rate Monotonic Analysis) details the complete technique [11].

It may be the case, however, that a real-time kernel is not the most effective solution. The cost or the memory requirements may be reasons for considering an alternate approach, however, let's examine system performance. The execution of each RTK procedure takes time. If we consider the *Order* message transfer between the two tasks *Supervision* and *MotorControl*, the important time is the delay between entering the Send procedure in *Supervision* and exiting the Receive procedure in *MotorControl*. In the case of a task switch, our experience shows us that the delay ranges between 20 and 50 µs or higher. Another problem with RTKs is the difficulty of estimating an upper bound on the execution time of each thread or task. Such values are necessary to verify that tasks can be scheduled and all timing constraints can be satisfied.

5: Optimization of the implementation

In this section we propose applying a set of transformation rules to the functional structure to reduce complexity before deciding on a software implementation. We interpret complexity as the number of software tasks and interdependencies on each processor. Even when using an RT kernel, our method should be considered as a step for simplifying the implementation, enhancing the resulting performance, reducing cost, and allowing one to reconsider the need for an RT kernel.

The first step is to correctly identify the links between the software tasks and the hardware environment; an accurate specification of the hardware /software interface is essential. The next step tries to reduce the number of asynchronous tasks. For clarity, we use the word Function for a cyclic sequential process in the functional specification and the word Task for the same kind of process as an implementation unit.

5.1: Transformations on Hw/Sw Relations (Rule 1)

Based upon the functional model, the links between each software task and its hardware environment are: event synchronization, shared variable, and message transfer. In this section we show now a transformation rule aids in the implementation of hardware/software interfaces.

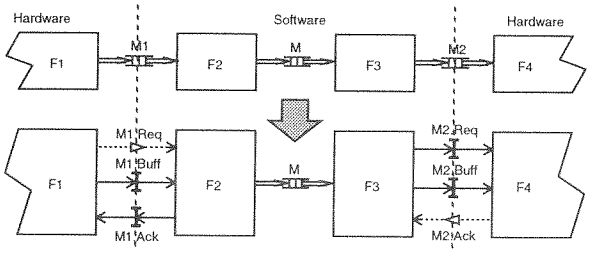
a- Case of an event and a shared-variable

An event input is implemented as a Boolean input to the microprocessor that can be polled or used as an interrupt input. An event output is also a Boolean. A shared-variable is implemented as a shared register or shared memory.

b- Case of a message transfer

A port linking two functions on different processors requires a hardware interface. The result of a transformation of the port into the two preceding relations is shown in Figure 3. A shared variable Mx.Buff is used to buffer one or several messages. Synchronization is implemented as a boolean variable in the software to hardware direction and as an event in the opposite direction. Thus software tasks F2 and F3 can be activated by interrupt signals M1.Req and M2.Ack.

Using this technique, we conclude *Rule 1:* a functional description to be implemented on each processor is only linked to its environment through events and shared variables. Therefore the implementation of the set of tasks can be determined separately from the environment.

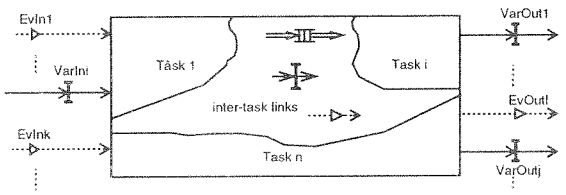


-Figure 3 - Transformation rule for Hw/Sw message transfers.

5.2: Minimum number of software tasks

Examining the result after applying Rule 1, an interesting question arises: for a given problem, is there a minimum number of tasks and if yes, what is the value? Figure 4 represents the general case. Let the functional description contain N functions and K event inputs. Normally all event inputs are mutually asynchronous because each is generated by the environment. Thus, the minimum number of tasks is K + 1. The added task is the processor background task.

Consider the example in Figure 3, F1 and F4 are asynchronous external functions. Applying Rule 1 gives the 2 asynchronous events M1.Req and M2.Ack. F2 and F3 are therefore asynchronous and cannot be merged. This justifies the need for the relation through the port M acting as a FIFO buffer. Removing the need for M2.Ack permits F2 and F3 to be merged into a single task.



-Figure 4 - General case for a functional description to be implemented on a microprocessor.

It is thus observed, that multitask decomposition of software depends only upon the asynchronous nature of event inputs. This is important because it gives designers a guide for functional decomposition. Designers tend to decompose a design into too many tasks because they have no termination rules. Later, the tasks must be merged to reduce implementation complexity.

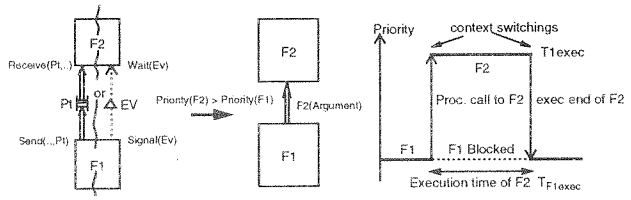
Asynchronous software tasks can be linked through shared variables. In such cases only the coherency of each variable has to be ensured. For small data structures this is accomplished by task prioritization or by interrupt masks. This is a major reason why the MCSE methodology recommends finding essential shared data and then functions during the functional design step.

5.3: Function merging (Transformation rule 2)

Based on the previous results, we observe that when the number of functions in the functional solution is greater than the minimum, transformations can be applied to reduce the number of tasks. Because an elementary function is a sequential cyclic process and thus equivalent to a single thread, the reduction means the serialization of several threads [6],[7].

If we consider 2 functions F1 and F2 linked through an event or a port (Figure 5), according to rate-monotonic scheduling, 2 different priorities must be assigned. Two possibilities exist. If Priority(F2) > Priority(F1) (Rule 2), the 2 functions can be merged in a single task because when F1 sends a message in Pt or signals the event EV to F2, the processor must be released and given to F2. The most efficient implementation is a procedure call in F2. Figure 8 illustrates a solution, the notation and the behavior. The interpretation of the timeline is exactly the same as for an interrupt except that task switching is controlled by the software. As explained in [1], a similar technique can be applied to merging several functions activated by the same event in a single temporary task, or several permanent functions in the background task.

When applicable, this rule is effective for reducing complexity and increasing performance. Note, the task switching overhead is approximately 1 µs. Applicability is based upon examining response times and CPU utilization ratio. For more details see [1],[11]. The execution end of F2 is $T_{F2end} = T_{F2exec}$ with T_{F2exec} its execution time and T2 the highest priority task. $T_{F1end} = T_{F1exec} + T_{F2exec}$, and so on from the highest task down to the background task.



-Figure 5 - Process merging of two dependent functions.

5.4: Function splitting (Transformation rule 3)

A function is activated by two or more events or messages, is difficult to implement even with an RT kernel. If K is the number of activation events and all are asynchronous, the *Rule 3* splits the function into K threads, each activated by one event. Being in the same function, the threads are interdependent, thus a shared variable is used to express the dependencies. One must be aware that such a case is relatively rare, occurring mostly in communication systems.

6: Hardware scheduling

After applying the preceding rules, a question arises: when the number of tasks is M>1, is it necessary to use a RT kernel? The answer depends on the application, the value of M, task interdependencies, and required performance. The designer should be able to determine these factors and easily decide upon the best implementation.

We now show that a microprocessor interrupt system can be used as an efficient priority-based scheduler. With such a scheme, an interrupt mask is used to control the order of (interrupt) task execution, i.e. order task execution according to some priority. Now, let the interrupts be used for task activation. The only difficulty is ensuring proper software task synchronization and satisfying any remaining post-optimization interdependencies between a task T1 to a lower-priority task T2 (opposite case in Fig 5). In such a case, T1 can use a digital output of the microprocessor connected to the appropriate interrupt input according the relative priorities of all tasks. T2 has to be linked to the selected interrupt input or vector in the vector table. If the connection cannot be direct, the few simple gates can be used. Consider the Motorola 68K microprocessor family, for example, which has 7 interrupts. Such a device permits one to schedule 7 tasks directly. Such complexity is infrequent in small embedded systems after the above optimizations.

7: Solution for the motor control system

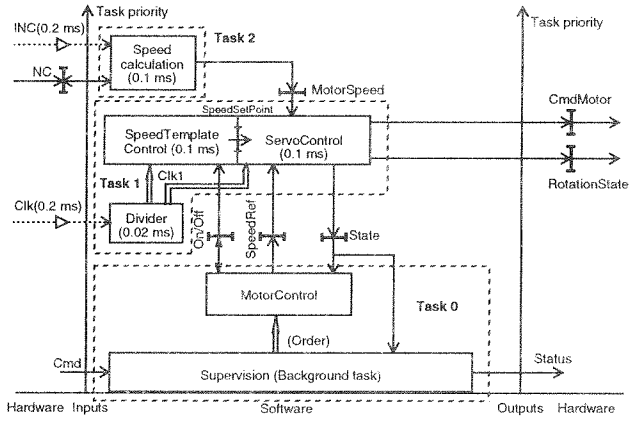
The techniques we have described are now illustrated using the example introduced in Section 3. The first step is to optimize the functional description by reducing the number of functions and thereby the number of tasks. The minimum task number is here 4: the two asynchronous events *Clk* and *Inc*, the message from port *Cmd*, and the background task.

The temporal constraints are used to determine the implementation and to verify the task scheduling. Realistic timing values are given. TA is the minimum time between two consecutive occurrences of an event; note, this is also the activation period of the dependent function. TE is an estimation of the maximum execution time of a function.

TA *Clk* = 0.2 ms;	TE *Divider* = 20 µs;
TA *Inc* = 0.2 ms;	TE *ServoControl* = 100 µs;
TA *Clk1* = 2 ms;	TE *SpeedTemplateControl* = 100 µs;
TA *Order* > 1 s;	TE *MotorControl* = 100 µs;
TE *SpeedCalculation* = 100 µs;	TE *Supervision* = 200 µs;

The rates of the *Cmd* and *Order* messages are here very low (<1Hz); therefore the CPU utilization for *Supervision* is null. The above values enable us to verify the scheduling of all functions. The utilization ratio is: Tuse max = 0.02/0.2 + (0.1+0.1)/2 + 0.1/0.2 = 0.7.

The most constrained task is *SpeedCalculation* which occurs only when the motor is at its maximum speed. Figure 6 gives the optimized software implementation diagram.



-Figure 6 - Software implementation diagram.

Using rate monotonic scheduling, *SpeedCalculation* has the highest priority. *SpeedTemplateControl* and *ServoControl*, which are synchronous to the same event *Clk1*, are grouped in a procedure called by *Divider*. *Clk* is a hardware interrupt; the three functions are merged in a single task *Task1* (Rule 2). *Supervision*, as the lowest priority function, is implemented as the background task. The *Cmd* message from another processor is polled by *Supervision* (Rule 1). The message transfer between *Supervision* and *MotorControl* is implemented as a procedure call because of the higher priority for *MotorControl* (Rule 2). Couplings between the tasks are done by the shared variables implemented as global variables in the software.

The result is a multiple task implementation (3 tasks) without using a RT kernel. The satisfiability of timing constraints (here no event loss) is easy to prove based upon the same execution times as before. The worst case CPU utilization ratio is still 0.7.

8: Conclusion

In this paper, we have described several powerful embedded software implementation techniques. The proposed strategy is twofold: first start from a good specification model and perform the partitioning and allocation, second optimize the software architecture to simplify and shorten the development, reduce cost, improve performance and more effectively evaluate and satisfy hard real-time constraints. The resulting software design is represented in a software implementation diagram that simplifies the optimization process and the analysis of temporal constraints. Performance estimations to aid in partitioning can easily be made using our techniques. One can easily hypothesize of an interactive tool to help the designer in applying the above rules. The example is interesting because it shows how one can optimize a design to achieve the best partitioning, function allocation, and hardware and software implementations. The proposed method is fully integrated into the MCSE system-level methodology.

9: References

[1] J.P. Calvez, Embedded Real-time Systems. A specification and Design Methodology, John Wiley, 1993

[2] J.P. Calvez, D. Heller, O. Pasquier, Uninterpreted Co-Simulation for performance evaluation of Hw/Sw systems, Proceedings of the Fourth International Workshop on Hardware/Software CoDesign, Pittsburgh, USA, March 18-20, 1996, pp 132-139

[3] P. Chou, E.A. Walkup, G. Borriello, Scheduling for reactive real-time systems, IEEE Micro, August 1994, pp 37-47

[4] P. Chou, G. Borriello, Interval scheduling: fine-grained code scheduling for embedded systems, Proceedings of the 32nd Design Automation Conference, San Francisco, USA, June 12-16, 1995, pp 462-467

[5] D.D. Gajski, F. Vahid, S. Narayan, J. Gong, Specification and Design of Embedded Systems, Prentice Hall, Englewood Cliffs, New Jersey, 1994

[6] R.K. Gupta, C.N. Coelho, G. De Micheli, Program implementation schemes for hardware-software systems, IEEE Computer, Vol 27, N1, January 1994, pp 48-56

[7] R.K. Gupta, Co-Synthesis of hardware and software for digital embedded systems, Kluwer Academic Publishers 1994

[8] R.K. Gupta, G. De Micheli, Constrainted software generation for Hardware-Software systems, Proceedings of the Thirth International Workshop on Hardware/Software CoDesign, Grenoble, France, Sept 22-24, 1994, pp 56-63

[9] R.K. Gupta, A framework for interactive analysis of timing constraints, Proceedings of the Fourth International Workshop on Hardware/Software CoDesign, Pittsburgh, USA, March 18-20, 1996, pp 44-51

[10] A. Kalavade, System-level CoDesign of Mixed Hardware-Software Systems, PhD dissertation, University of California, Berkeley, Sept 1995

[11] M.H. Klein et al., A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems, Kluwer Academic Publishers, 1994