Enforcing Schedulability of Multi-Task Systems by Hardware-Software Codesign

Youngsoo Shin Kiyoung Choi
School of Electrical Engineering
Seoul National University
Seoul 151-742, Korea

Abstract

This paper deals with the problem of hardware-software codesign of hard real-time systems. For a given task set, we perform an exact schedulability test to determine whether the task set is schedulable or not. When there is a task that cannot meet the deadline, we compute the amount of time by which the deadline is missed. Then we determine which tasks should reduce their execution time to compensate that amount of time deviation. The reduction of execution time is achieved by implementing parts of the tasks with hardware. With this approach, we can systematically design a hard real-time system which is infeasible with all software implementation. Preliminary experimental results are given to demonstrate the effectiveness of our approach.

I. Introduction

These days, as the complexity of embedded systems increases, a systematic design approach called hardware-software codesign is receiving a lot of attention. Most such embedded real-time systems can be described as a set of tasks with timing constraints. Depending on the strictness of timing constraints, a system can be referred to as a hard real-time or a soft real-time system. In a hard real-time system, it is crucial to satisfy timing requirements as well as its functional correctness. Such systems are easily found in most control systems, avionics, and many other embedded systems.

Usual approaches to a real-time system design involve much ad-hoc style engineering. This includes hand-crafted code tuning, reimplementing core routines, experimenting with various timing parameters, and so on. In the extreme, the entire system may be redesigned to satisfy timing constraints. In designing complex systems, however, these approaches are hard to employ or even are not acceptable. Moreover, systems designed with these approaches are brittle when we should replace certain tasks with their version-ups or when we should add other functionalities. Therefore, we need a systematic approach which will replace the ad hoc approaches.

There have been some research efforts for codesign targeting a multiple task model [1], [2]. Their approaches can be considered as an architectural exploration in that timing parameters for tasks are given for various kinds of processing elements from which they select appropriate candidates. However, it is difficult to extract a priori timing information for various kinds of implementation styles and for various kinds of processing elements. Therefore, their approaches are sometimes hard to employ in realistic system design.

In this paper, we take an approach where the execution time of tasks can be reduced by moving some code fragments to hardware components. In other words, when we cannot schedule the given task set in a specified deadline, we reduce the execution time of some tasks by employing a coprocessor which is a hardware implementation of the code fragments of the tasks. In this approach, the essential problem is a decision about how much portion of which tasks should be implemented with hardware components.

The overall flow of our approach is as follows. First, the system is specified as a set of tasks with their timing attributes/constraints. Each task is modeled as a control data flow graph(CDFG) [3]. If there are tasks to be speeded up to satisfy the schedulability condition, we compute the percentage of the task's execution time to be cut off in order to satisfy the schedulability condition. From this information, a hardware-software partitioner [4] partitions the task into two parts to be implemented with hardware and software, respectively. Then the interface between the two parts is generated and annotated to each partitioned parts [5], [6]. The software part is synthesized by software synthesis process [3] to become an executable and hardware part is synthesized to become an ASIC or FPGA. All these processes are illustrated in Fig. 1.

The rest of the paper is structured as follows. In the next section, we summarize rate monotonic scheduling and its schedulability test which are used in our approach. We propose and describe a novel algorithm for achieving schedulability using schedulability analysis

0-8186-7895-X/97 $10.00 © 1997 IEEE

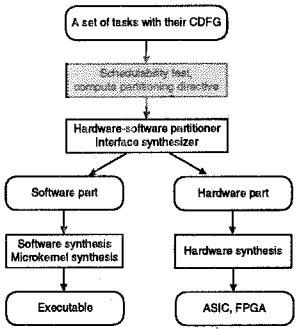

Fig. 1. Hardware-software codesign flow.

in section III. In section IV, we consider extensions to our basic algorithm in order to incorporate deadline monotonic scheduling and discuss the scheduler overhead. We show experimental results in section V and draw conclusions with some remarks in section VI.

II. Rate Monotonic Schedulability Analysis

Given a set of independent periodic tasks, rate monotonic scheduling (RMS) assigns a higher priority to tasks with shorter period or with higher execution rate [7]. RMS is proved to be an optimal static priority assignment in the sense that if a given task set can be scheduled using a certain static priority scheduling algorithm, then it can also be scheduled using RMS. The advantage of RMS lies in its simplicity as well as the existence of schedulability tests.

In [7], they proposed the following sufficient and non-necessary schedulability test which is based on the processor utilization factor:

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq n(2^{\frac{1}{n}} - 1) \qquad (1)$$

where C_i and T_i are the worst case execution time and period of task τ_i, respectively.

There are also necessary and sufficient schedulability test based on the *critical instant* theorem [8] which tests schedulability on the sets of *scheduling points* defined by the following equation.

$$S_i = \{kT_j | j = 1, \ldots, i; k = 1, \ldots, \lfloor \frac{T_i}{T_j} \rfloor\} \qquad (2)$$

where τ_i's are sorted in the ascending order of the period.

Note that we need to check the schedulability of task τ_i only at those points which are multiples of $T_j (T_j \leq T_i$ for $1 \leq j \leq i)$ and in the interval $[0, T_i]$ as expressed in equation (2). Task τ_i is schedulable if it satisfies the following equation.

$$\min_{\{t \in S_i\}} \frac{\sum_{j=1}^{i} C_j \cdot \lceil \frac{t}{T_j} \rceil}{t} \leq 1 \qquad (3)$$

The above process is illustrated in the following example.

Example 1: Consider the case of the following three tasks with scheduling points computed as shown:

τ_1: C_1=4, T_1=10, $S_1 = \{T_1\}$
τ_2: C_2=10, T_2=16, $S_2 = \{T_1, T_2\}$
τ_3: C_3=7, T_3=25, $S_3 = \{T_1, T_2, 2T_1, T_3\}$

We perform schedulability analysis for all three tasks at each scheduling point as follows.

$$\begin{aligned}
\tau_1 : \quad & C_1 \leq T_1 \\
\tau_2 : \quad & C_1 + C_2 > T_1 \\
& 2C_1 + C_2 > T_2 \\
\tau_3 : \quad & C_1 + C_2 + C_3 > T_1 \\
& 2C_1 + C_2 + C_3 > T_2 \\
& 2C_1 + 2C_2 + C_3 > 2T_1 \\
& 3C_1 + 2C_2 + C_3 > T_3
\end{aligned}$$

From the analysis, we can see that τ_1 is schedulable, but τ_2 and τ_3 do not meet their deadlines.

The basic RMS theory seems to be very restrictive in that they can only be applied to the cases where tasks are independent and periodic, deadlines are equal to periods, and tasks are executing on a uniprocessor. However, there have been many researches on extending the basic RMS theory for more general cases [9], [10].

III. Algorithm for Achieving Schedulability

The schedulability analysis described in the previous section cannot help once the task set is found to be unschedulable. In this situation, a lot of ad hoc style engineering is needed until the modified task set satisfy the schedulability. Our work on algorithm for achieving schedulability is motivated by this shortcoming of the existing methodology for hard real-time system designer.

Our algorithm for achieving schedulability is based on the exact schedulability analysis given by equation (3). In our algorithm, we iterate a process of reducing the execution time of tasks until all tasks meet their deadlines. To this end, we define mrc_i as the maximum value by which we can reduce the execution time of task τ_i and it is computed as follows:

$$\begin{aligned}
mrc_i \quad = \quad & C_i - [input_communication_overhead + \\
& critical_path_length + \\
& output_communication_overhead] \qquad (4)
\end{aligned}$$

where *critical_path_length* is the latency of the task obtained by implementing it with hardware through scheduling[1] and allocation. Note that we need to compute mrc_i for task τ_i only when it is totally or partially implemented with hardware and this will be evident in the following discussion. This computation does not increase much the total computational complexity because tasks determined to be implemented with hardware will be synthesized after all. Alternatively, the designer can control mrc_i by assigning a certain value. For example, we can assign 0 to mrc_1 when we do not want to reduce the execution time of τ_1.

We define $S_{i,j}$ as the jth scheduling point of task τ_i when elements of S_i are sorted in the ascending order. For each task which does not satisfy equation (3), we compute the time deviation by which the task misses its deadline. We define $\Delta c_{i,j}$ as the time deviation of task τ_i at the jth scheduling point. It is given by the following equation.

$$\Delta c_{i,j} = \sum_{k=1}^{i} C_k \lceil \frac{S_{i,j}}{T_k} \rceil - S_{i,j} \tag{5}$$

For each $\Delta c_{i,j}$, we compute the time d_{ijk}, $k = 1, \ldots, i$, by which the execution time of each task τ_k must be reduced in order to make τ_i schedulable. It is computed by the following equation.

$$d_{ijk} = \frac{\Delta c_{i,j}}{\lceil \frac{S_{i,j}}{T_k} \rceil} \tag{6}$$

It can be easily shown that if the execution time of any task τ_k is reduced by the amount given in equation (6), then τ_i can be made schedulable. From this computation, we can compute the minimum required time by which the execution time of τ_k must be reduced in order to make all tasks schedulable as follows.

$$\mathcal{D}_k = \max_i \left[\min_j d_{ijk} \right] \tag{7}$$

For any k, if we reduce the execution time of τ_k by $\mathcal{D}_k$, then all tasks become schedulable. However, if $\mathcal{D}_k$ is larger than mrc_k which is the maximum value we can take off from the execution time of task τ_k, then it is impossible to achieve our goal by only reducing the execution time of task τ_k. We solve this problem through iteration. First, we reduce the execution time of τ_1 by $\mathcal{D}_1$. If $\mathcal{D}_1$ is larger than mrc_1, we iterate the above steps with $\tau_2, \tau_3, \ldots$ until all tasks become schedulable. We start from τ_1 because it is more effective than starting from any other task. Note that $\mathcal{D}_m$ is always smaller than or equal to $\mathcal{D}_n$, provided that m is smaller than n. In the $(k+1)$th iteration, $\Delta c_{i,j}$ computed in the kth iteration should be updated. Note that the execution time of $\tau_1, \ldots, \tau_k$ have been modified during the first k iterations. This computation is performed incrementally using the following equation.

$$\Delta c_{i,j}^{k+1} = \Delta c_{i,j}^{k} - mrc_k \cdot \lceil \frac{S_{i,j}}{T_k} \rceil \tag{8}$$

Example 2: Let's revisit the case of three tasks in Example 1 and assume that mrc_i is 70% of C_i for all tasks. In the first iteration, we compute $\mathcal{D}_1$ as follows.

τ_2	$\Delta c_{2,1} = 4$	$d_{211} = 4$	$d_{212} = 4$	
	$\Delta c_{2,2} = 2$	$d_{221} = 1$	$d_{222} = 2$	
τ_3	$\Delta c_{3,1} = 11$	$d_{311} = 11$	$d_{312} = 11$	$d_{313} = 11$
	$\Delta c_{3,2} = 9$	$d_{321} = 4.5$	$d_{322} = 9$	$d_{323} = 9$
	$\Delta c_{3,3} = 15$	$d_{331} = 7.5$	$d_{332} = 7.5$	$d_{333} = 15$
	$\Delta c_{3,4} = 14$	$d_{341} = 4.7$	$d_{342} = 7$	$d_{343} = 14$
$\mathcal{D}_1 = 4.5$				

Because $\mathcal{D}_1$ exceeds mrc_1 which is 2.8, we iterate the process. After the second iteration, the table is updates as follows.

τ_2	$\Delta c_{2,1} = 1.2$		
	$\Delta c_{2,2} = -3.6$		
τ_3	$\Delta c_{3,1} = 8.2$	$d_{312} = 8.2$	$d_{313} = 8.2$
	$\Delta c_{3,2} = 3.4$	$d_{322} = 3.4$	$d_{323} = 3.4$
	$\Delta c_{3,3} = 9.4$	$d_{332} = 4.7$	$d_{333} = 9.4$
	$\Delta c_{3,4} = 5.6$	$d_{342} = 2.8$	$d_{343} = 5.6$
$\mathcal{D}_2 = 2.8$			

The negative value of $\Delta c_{2,2}$ indicates that τ_2 becomes schedulable. The values of d_{3j1} are not present in the table because we cannot reduce the execution time of τ_1 further after the first iteration. Because $\mathcal{D}_2$ is smaller than mrc_2 which is 7, the iteration completes.

IV. Extension

Our algorithm described in the previous section is based on the schedulability analysis for a system using RMS. However, our methodology can be easily extended in various directions. In this section, we consider some of these extensions.

A. Deadline Monotonic Scheduling

Deadline monotonic scheduling(DMS) is an extension of RMS where the deadline of a task is smaller than the period. DMS provides a more flexible model for various situations which include catering for aperiodic events. It is also proved to be optimal in the

[1] This should not be confused with the scheduling of real time tasks. Scheduling in this phrase means assigning a control step to each operation in hardware implementation and is one of the phases performed in a high-level synthesis.

same context of RMS [11]. There is also a schedulability test which is both necessary and sufficient [12], [11]. It finds the worst case response time of task τ_i as follows:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j \qquad (9)$$

where B_i is the maximum duration when we use *priority ceiling protocol* [13] in which task τ_i can be blocked by lower priority tasks when it tries to access shared resources. $hp(i)$ is a set of tasks with priorities higher than τ_i. There are no simple solutions for equation (9) because R_i appears on both sides. However, the solution can be found by using the following recurrence equation.

$$t_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{t_i^n}{T_j} \rceil C_j \qquad (10)$$

R_i becomes t_i^{n+1} when t_i^{n+1} equals to t_i^n.

It is proved that equation (10) converges for a set of tasks when processor utilization is smaller than or equal to 1. However, this is not guaranteed when utilization is larger than 1. Therefore, we cannot use this schedulability test for our purpose. Instead, we can use RMS analysis for testing schedulability. The difference from normal RMS analysis is that we test only at scheduling points that are smaller than or equal to the deadline of a task. For this extension, equation (2) is modified as follows:

$$S_i' = (S_i - \{t | t \in S_i, t > D_i\}) \cup \{D_i\} \qquad (11)$$

Fig. 2 shows the pseudo code for the proposed algorithm that achieves schedulability by adjusting task execution times. It works for both RMS and DMS.

B. Scheduler Overhead

In most schedulability analyses, the cost of scheduler overhead is assumed to be 0 for simplicity. However, we should take into account this overhead in practical system design. In [14], [15], they proposed effective methods for incorporating scheduler overhead into fixed priority schedulability analysis. We can easily accommodate them to our algorithm.

For example, when we use timer-driven scheduling or tick scheduling[2] [14], we can use the following equation instead of equation (3).

$$\min_{\{t \in S_i\}} \left(\sum_{j=1}^{i} \frac{C_j + C_{preempt} + C_{exit}}{t} \lceil \frac{t}{T_j} \rceil \right)$$

[2]A scheduler maintains a *run queue* and a *delay queue*. The former holds the tasks ready for execution and is ordered by priority. The latter holds the suspended tasks and is ordered by due time for release. The scheduler is released at regular intervals by a timer interrupt and moves candidate tasks from the delay queue to the run queue.

```
Calculate_deviation_time() {
    find_scheduling_points();
    schedulability_test();
    for (k = 1, 2, ..., n) {
        for (all tasks τ_i wich are not schedulable)
            compute Δc_{i,j} at S_{i,j};
        D_k = max_i [min_{j,k} d_{ijk}];
        if (D_k > mrc_k) {
            Δc_{i,j} = Δc_{i,j} − mrc_k ⌈S_{i,j}/T_k⌉;
            reduce C_k of τ_k by mrc_k;
        }
        else {
            reduce C_k of τ_k by mrc_k − D_k;
            exit loop;
        }
    }
}
```

Fig. 2. Pseudo code for adjusting task execution times for schedulability.

$$+ \lceil \frac{t}{T_{tic}} \rceil \frac{C_{timer}}{t} + \frac{T_{tic}}{t} \leq 1 \qquad (12)$$

where $C_{preempt}$ is time for handling task preemption, C_{exit} for handling normally completed task, C_{timer} for handling timer interrupt. T_{tic} is period of timer interrupt.

V. Experimental Results

We perform experiments for two kinds of examples. The first example is a case study based upon an inertial navigation system(INS) [14] and the second example is the GAP case study [16]. In the first example, we decrease periods of all tasks by 10% of original periods. Table I and II summarize timing attributes of the two examples, respectively. Note that we can use RMS schedulability analysis for INS example because periods are equal to deadlines. In GAP example, we use DMS test. In both examples, we take into account scheduling overhead. The parameters for scheduler overhead are set as follows.

$$C_{preempt} = 2, C_{exit} = 2, C_{timer} = 2, T_{tic} = 20$$

For all the experiments, we assume mrc_i to be 70% of C_i.

In the first experiment, $\mathcal{D}_1$ is 280.812 after the first iteration. The results are that we should reduce the execution time of task1 by 23.8% to satisfy timing constraints and the resulting processor utilization is 0.893. Recall that the reduction of the execution time is achieved by implementing part of task1 with hardware. In the second experiment, $\mathcal{D}_1$ is 1046.25 after the first iteration. The results are that we should reduce the computation time of task1 by 34.9% to satisfy

TABLE I

TIMING ATTRIBUTES OF INS TASKS

	Timing attributes			
	T_i	D_i	C_i	B_i
task1	2250	2250	1180	0
task2	36000	36000	4280	0
task3	562500	562500	10280	0
task4	900000	900000	20280	0
task5	900000	900000	100280	0
task6	112500	112500	25000	0

TABLE II

TIMING ATTRIBUTES OF GAP TASKS

	Timing attributes			
	T_i	D_i	C_i	B_i
task1	200000	5000	3000	300
task2	25000	25000	2000	600
task3	25000	25000	5000	900
task4	40000	40000	1000	1350
task5	50000	50000	3000	1350
task6	50000	50000	5000	750
task7	59000	59000	8000	750
task8	80000	80000	9000	1350
task9	80000	80000	2000	450
task10	100000	100000	5000	1050
task11	200000	200000	1000	450
task12	200000	200000	3000	450
task13	200000	200000	1000	450
task14	200000	200000	1000	1350
task15	200000	200000	3000	0
task16	1000000	1000000	1000	0
task17	1000000	1000000	1000	0

timing constraints and the resulting processor utilization is 0.845.

VI. Conclusions

We have proposed in this paper a systematic design approach for hard real-time systems. Our work was motivated by the following two facts.

- Design of hard real-time systems involve a lot of ad hoc engineering. Systematic design approach is needed to design a complex system.
- Most schedulability analysis can do little when tasks can not meet their deadlines. We need systematic strategy for this situation.

To satisfy strict timing constraints, we computed a time deviation by which tasks miss their deadlines. This computation can be used as a directive for hardware-software partitioning tool.

Our future work includes synthesis of predictable microkernel and incorporating our approach into a design environment as shown in Fig. 1.

References

[1] T. Yen and W. Wolf, *Hardware-Software Co-Synthesis of Destributed Embedded Systems*, Kluwer Academic Publishers, 1996.

[2] C. Lee, M. Potkonjak, and W. Wolf, "System-level synthesis of application specific systems using A* search and generalized force-directed heuristics," in *Proc. Int. Symposium on System Synthesis*, Nov. 1996.

[3] Y. Shin and K. Choi, "Software synthesis through task decomposition by dependency analysis," in *Proc. Int. Conf. on Computer Aided Design*, Nov. 1996, pp. 98–102.

[4] J. Jeon and K. Choi, "An effective force-directed partitioning algorithm for hardware-software partitioning problem," submitted to 5th Int. Workshop on Hardware/Software Co-Design, Dec. 1996.

[5] K. Kim, "Generation of interface module in hardware-software codesign," M.S. thesis, Seoul National University, Feb. 1996, in Korean.

[6] K. Kim, Y. Kim, Y. Shin, and K. Choi, "An integrated hardware-software cosimulation environment with automated interface generation," in *Proc. 7th IEEE Int. Workshop on Rapid Systems Prototyping*, June 1996, pp. 66–71.

[7] C. L. Liu and James W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[8] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proc. IEEE Real-Time Systems Symposium*, Dec. 1989, pp. 166–171.

[9] L. Sha, R. Rajkumar, and S. Sathaye, "Generalized rate-monotonic scheduling theory: A framework for developing real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 68–82, Jan. 1994.

[10] L. Sha and J. Goodenough, "Real-time scheduling theory and Ada," *IEEE Computer*, vol. 23, no. 4, pp. 53–62, Apr. 1990.

[11] N. Audlsey, A. Burns, A. Richardson, and A. Wellings, "Hard real-time scheduling: The deadline monotonic approach," in *IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[12] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer J.*, vol. 29, no. 5, pp. 390–395, Oct. 1986.

[13] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[14] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 920–934, Sept. 1993.

[15] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 21, no. 5, pp. 475–480, May 1995.

[16] C. Locke, D. Vogel, and T. Mesler, "Building a predictable avionics platform in Ada: A case study," in *Proc. Real-Time Systems Symp.*, Dec. 1991.