Real-time Kernel Support for Coprocessors: Empirical Study of an SoPC

Andrew Morton and Wayne M. Loucks
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

***Abstract*— The system on chip paradigm consists of one or more instruction set processors integrated with custom hardware on a single integrated circuit. A uni-processor real-time kernel is presented that integrates hardware coprocessors by viewing them as system resources to be scheduled in conjunction with the processor. The kernel implements the earliest-deadline first scheduling policy. To demonstrate this "hardware/software coscheduling", an automobile engine idle speed controller and model is implemented. The target platform for this test-case is the Nios[1] system on programmable chip, a soft-core Nios processor embedded in an APEX[2] field programmable gate array. Impact on schedule analysis and application partitioning is discussed.**

Keywords: SoC, Scheduling, EDF, Coprocessor

I. Introduction

The system on chip (SoC) paradigm consists of one or more instruction set processors (ISPs) integrated with custom hardware on a single integrated circuit. A common application domain for SoCs is real-time embedded systems. A real-time kernel is presented here that represents part of a unified approach to the scheduling of both software and hardware on an SoC.

Some research projects have considered the integrated scheduling of hardware and software processes. In most cases a software scheduler is synthesized for the application. Chinook [1] uses watchdogs to implement reactive behaviour. A watchdog watches for an event. At the occurance of an event, a new "mode" is selected. Each mode consists of concurrent operations that have been serialized. A similar approach is taken to scheduler synthesis described in [2]. Concurrent threads are serialized into thread frames. An asynchronous event triggers a thread frame for execution. The run-time scheduler attempts to interleave frame execution in an order that satisfies timing requirements. Both methods offer low run-time scheduler overhead but have sub-optimal processor utilization [3]. In [4] abstract Codesign Finite State Machines (CFSMs) can be implemented in one of hardware, software or peripheral micro-controller. Facilities for communicating events between CFSMs are synthesized. The generated software is scheduled by either cyclic or static priority policies.

This work supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada).

[1]Nios is a trademark of Altera Corporation

[2]APEX is a trademark of Altera Corporation

Higher ISP usage can be achieved in theory by using the Earliest Deadline First (EDF) scheduling policy [5]. However the run-time overhead of EDF schedule analysis discourages application in embedded systems [6]. In this paper, a real-time kernel is presented that implements the EDF policy and supports hardware coprocessor integration in a manner that facilitates compile-time analysis of schedule feasibility. This approach may make EDF scheduling a viable solution for a hardware-software codesign environment.

The kernel is briefly introduced with the concepts of co-processor and auto-processor kernel objects. The concepts are demonstrated in a test-case (automobile engine model and control) running on a SoPC (System on Programmable Chip). Implications for schedule implementation are discussed and resulting analysis and partitioning issues are also considered.

II. Kernel

An object-oriented uni-processor real-time kernel is implemented called cs1 (for CoScheduler1). Three types of system resources are managed by cs1: the real-time clock, inter-task message queues and hardware processors. Two software task types are supported: periodic and aperiodic. (An aperiodic task is triggered by inter-task messages or by hardware interrupts.) All tasks are created statically. Periodic task τ_i has phase s_i, period T_i and relative deadline D_i. A periodic task is released at the start of each period (at time r_i) by the real-time clock. The task must complete by deadline $d_i = r_i + D_i$. A message triggered aperiodic task has a message trigger and deadline. The task is released when it receives a message from the appropriate message queue. A hardware triggered aperiodic task has a hardware trigger and deadline. The task is released when the appropriate hardware processor interrupts the ISP. Tasks are scheduled by the preemptive EDF policy: that is, of all active tasks, the task with the earliest deadline gets to run. In practice, when a task is released, it is inserted into the run list which is sorted by deadline using a min-heap.

Inter-task message queues are encapsulated by message objects. A message object has storage for a user defined number and size of message buffers. It also has lists for tasks that block waiting to send or receive. Message objects are accessed via their *send* and *receive* methods. If a task blocks on a message send or receive, it is removed from the run list and placed on the appropriate wait list. When an action on the

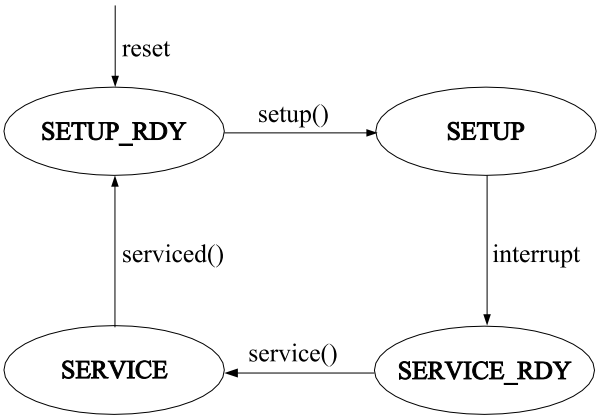

Fig. 1. Coprocessor Object FSM

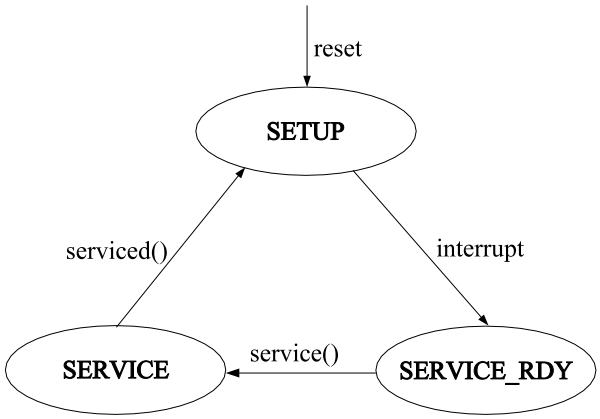

Fig. 2. Auto-Processor Object FSM

message object enables the task, it is placed back on the run list.

For this work, a coprocessor is defined as a hardware device to which software may dispatch a job and which will interrupt the ISP upon completion of the job. The coprocessor is required to have a software mask-able interrupt and a status register that can be queried to determine interrupt status of the device.

Coprocessors are encapsulated by kernel coprocessor objects. A coprocessor object contains the interrupt priority, status and control register addresses, and masks for testing and disabling interrupts. The vector table entries for all coprocessors are linked to the kernel routine. Therefore coprocessor events are handled by the kernel, which also manages message and real-time clock events. Each coprocessor object also has lists for tasks that block on setting up or servicing the coprocessor. The coprocessor object finite state machine (FSM) is shown in Figure 1. Software tasks access coprocessor objects via the *setup*, *service* and *serviced* methods. Before a task uses the coprocessor, it must first gain exclusive access via the *setup* method. If that task is blocked on coprocessor setup, it is removed from the run list and placed on the coprocessor setup wait list. After the coprocessor is setup, it may interrupt the ISP at any time to indicate "ready for service". The task which has set up the coprocessor can invoke the *service* method which may require the task to be blocked until the coprocessor has interrupted the ISP. After servicing the coprocessor, the task invokes the *serviced* method which puts the coprocessor back into the "ready for setup" state.

In addition to coprocessors, which implement a subset of a task's functionality, there may be autonomous hardware processors (termed auto-processors) that function independent of software tasks. For example, an auto-processor may sample an input data stream at a fixed frequency and pass it on to a software task. The primary difference between the auto-processor and the coprocessor is that the auto-processor requires a software task to set it up. The auto-processor object FSM, shown in Figure 2 does not have a *setup_rdy* state. After servicing, the FSM moves directly to the *setup* state to be ready for the next interrupt. The kernel auto-processor object is inherited from the kernel coprocessor object. It does not implement the *setup* method.

To integrate the auto-processor into the kernel, an aperiodic task is defined that is triggered by the auto-processor. The software task may be the consumer task for data from the auto-processor, or it may act as a wrapper that simply sends the data to a message queue from which other tasks may access it. This "wrapper" enables the auto-processor to be integrated into the kernel with little custom programming.

III. Idle Engine

The idle engine problem was selected as a test-case because it consists of several tightly interacting processes. The interaction between environment and controller is expected to be typical of many real-time applications. The test-case is not entirely authentic because the environment and the engine model are actually simulated on the same processor as the control application. The idle engine model, as described in [7] is summarized here.

A 4-cylinder automobile engine is modeled in the idle state (out of gear). It is a hybrid model, combining a continuous time system (CTS) with a discrete event system (DES). The components of the CTS are manifold pressure, crankshaft speed and piston position. The cylinders are modeled by a FSM that represent the interleaving between spark ignitions and dead-centers (when a piston reaches the top of its stroke). The DES variables track the torque generated by the pistons. The CTS, FSM and DES are tightly coupled: the FSM transitions in response to CTS events, the DES updates variables based on CTS events and FSM state, and the CTS is affected by DES output.

The model has control inputs (throttle angle and spark advance) and environment inputs (load torque). Both increasing the throttle angle and increasing the spark advance increase the generated torque. The throttle has greater control authority while the spark advance has less authority but has a faster response time. The load torque disturbances for an ideal engine

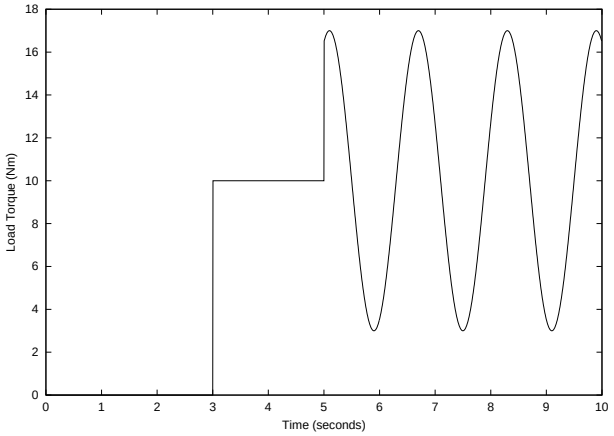

Fig. 3. Environment Input

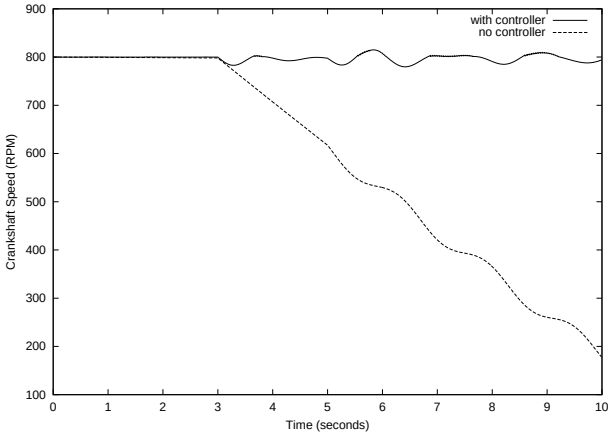

Fig. 4. Engine Output

are generated by such phenomena as the air conditioning system and the steering wheel servo-mechanism.

The goal is to maintain the crankshaft speed in the range of 800 ± 30 RPM (rotations per minute) under load torque disturbances. The controller devised for this test-case consists of a PID controller for the throttle angle and a P controller for the spark advance. The test-case consists of four blocks: idle engine model, environment, controller and user interface. The simulated environment input and the resulting crankshaft speed are shown in Figures 3 and 4. The simulated load torque is similar to that shown [7]. In Figure 4, the resulting crankshaft speed is shown with and without the controller.

The idle engine application consists of six tasks configured as shown in Figure 5. The directed arrows indicate data dependency. s is start-time (phase), T is period, D is deadline, C is worst-case execution time on the Nios soft-core processor, running at 33 MHz on an Altera APEX 20KE FPGA. All times are given in units of seconds. Task FSM+DES is an aperiodic task that is triggered by *dc* (dead-center) and *spk* events which are generated by Task CTS. This aperiodic task is represented as a sporadic task with minimum inter-arrival period T. A task set, such as this, consisting of periodic and sporadic tasks, is denoted a hybrid task set.

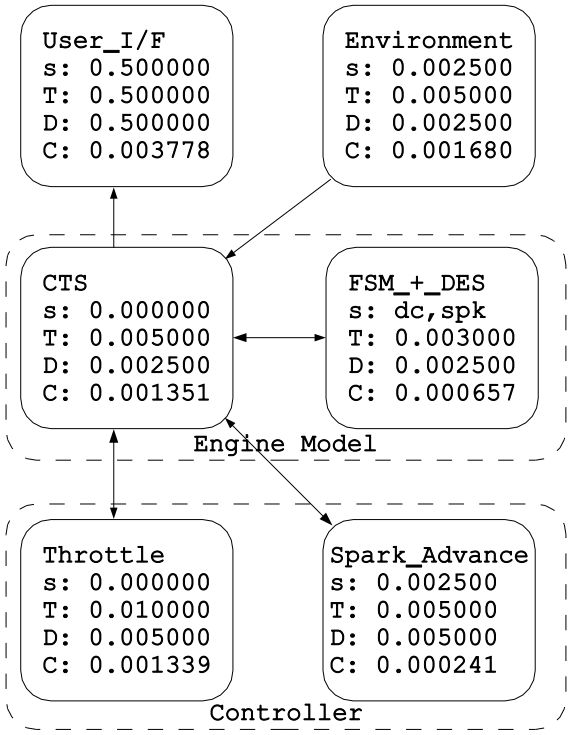

Fig. 5. Application Configuration

Processor utilization U is defined by Liu and Layland [3] as

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Baruah *et. al.* demonstrate [8] that a necessary condition for feasibly scheduling a hybrid task set under EDF is

$$U \leq 1$$

For the idle engine task set $U = 1.01486$ indicating that it is infeasible to schedule this task set by EDF. (Note that EDF is optimal, in the sense that it will only fail to meet a deadline if no other scheduling algorithm can meet the deadline [9].) In practise, it was found that Task Throttle was late 22 times in a 10 second interval.

A. Cordic Coprocessor

A manual examination of the task code revealed that Task Environment invokes the C cos() library function each time it runs. The worst-case execution time of this function was found to be $0.001435s$. If this function is moved to a coprocessor, the schedule may become feasible. A hardware coprocessor was therefore implemented that calculates the cosine using the Cordic algorithm [10]. This algorithm calculates the cosine of an n-bit fixed point number in n iterations using 3 adders, 2 shifters and a ROM for constants. The implemented coprocessor converts an IEEE floating point double to a 64-bit fixed point number, applies the Cordic algorithm, and converts the result back to floating point. The coprocessor

TABLE I
COSINE WORST-CASE EXECUTION TIMES

cos() func		0.001435 s
cordic cosine		0.000081 s
Environment	with cos()	0.001680 s
	with cordic	0.000343 s

TABLE II
FPGA RESOURCE USAGE

	Logic Cells	Registers (bits)	Memory (bits)
cordic	3840	1639	23808
SoPC	7246	3110	50304

is implemented with microprogrammed control. The resulting worst-case execution times are shown in Table I and the FPGA resource usage is shown in Table II. These numbers are for an Altera APEX 20KE FPGA. The SoPC consists of the CPU, ROM, timer, UART and cordic coprocessor. The SoPC uses 86% of the logic elements and 47% of the embedded system blocks (memory).

To integrate the cordic coprocessor into the kernel, a coprocessor object is created for it. The call to the cos() C library function in Task Environment is then replaced with code such as this (C++):

```
system.coproc[0].setup();
cordic->data = angle;
cordic->go = 0; // start calculating
// enable interrupt
cordic->statCtrl = Cordic::DoneMask;
system.coproc[0].service(status);
cosine = cordic->data;
system.coproc[0].serviced();
```

The processor utilization is $U = 0.747456$, indicating that it may be feasible to schedule the idle engine task set under EDF. It was found that the application executed with no late tasks. It was also possible to increase the model resolution by decreasing the simulation time from $0.005s$ to $0.00383s$.

No auto-processor has been incorporated into the idle engine application. However, if the idle engine controller were implemented in an automobile, an auto-processor could be used to interface a crankshaft speed sensor to the control tasks running on the ISP.

IV. COPROCESSOR SCHEDULING

The method used to account for coprocessor execution time in calculating deadlines is illustrated here by example (Figure 6). τ_1 has a worst-case execution time $C_1 = 5$ and a relative deadline $D_1 = 7$. The execution time of τ_2 is broken down into 3 parts: C_{2a} is time spent before invoking the coprocessor, C_{2b} is time spent executing on the coprocessor and C_{2c} is time spent after the coprocessor job finishes. If these tasks are both released at time 0 and are executed in order of deadline (Scenario 1), then τ_1 finishes on time but τ_2 finishes late. However, both tasks could have finished on time if τ_2 was able to start its coprocessor job earlier. The solution used with the

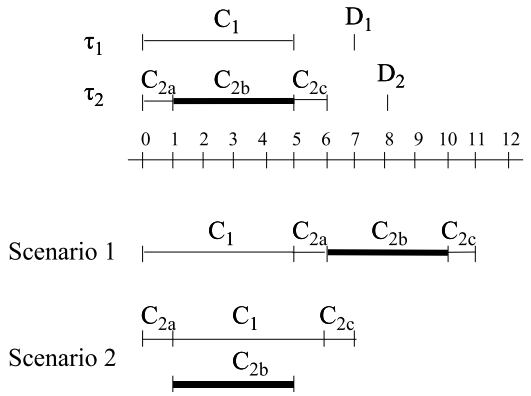

Fig. 6. Scheduling with Modified Deadlines

cs1 kernel is to assign a second deadline to tasks that invoke coprocessors:

1) $\forall \tau_i$ that use a coprocessor:
   initial deadline $d_i' = d_i - (C_{ib} + C_{ic})$
2) when τ_i invokes the coprocessor service method
   deadline = d_i

Applying this rule, the initial deadline for τ_2 becomes $d_2' = 8 - (4 + 1) = 3$. Scenario 2 shows the result. τ_2 executes before τ_1. τ_2 transfers execution to the coprocessor at $t = 1$, at which time it's deadline is restored to $d_2 = 8$ and τ_1 starts executing. At $t = 5$, the coprocessor is ready to be serviced but since $d_1 < d_2$, τ_1 continues executing.

Instead of employing one task with two deadlines, d_i' and d_i, as done above, one could employ a pair of tasks as shown in Figure 7. This raises 2 problems however. The first problem is that the slack must be divided statically between the task pair. (Slack of task i, S_i, is the maximum time that the execution of τ_i can be delayed without it missing its deadline.) The choice of appropriate slack division is complex and is task set dependent. The second problem arises when multiple task pairs share a coprocessor. The application would need to coordinate execution of the task pairs so that, after one task has dispatched a job to the coprocessor, the correct task is invoked to service the coprocessor when it completes. The two-deadline approach addresses both these problems as explained below.

1) There is no need to find an optimum division of slack because the slack is shared dynamically between the first and second parts of the task:

$$\begin{aligned} S_i' &= d_i - C_{ia} + C_{ib} + C_{ic} \\ S_i &= d_i' - f_i' \end{aligned}$$

   The slack of the first part of the task, S_i', is equal to the slack of the whole task. The second part of the task has slack, S_i, equal to any slack not used when the first part finishes at f_i'.
2) The task that sent the job to the coprocessor is the same task that services the coprocessor when it completes. This coordination of execution is enforced by the coprocessor methods. A task will only invoke the

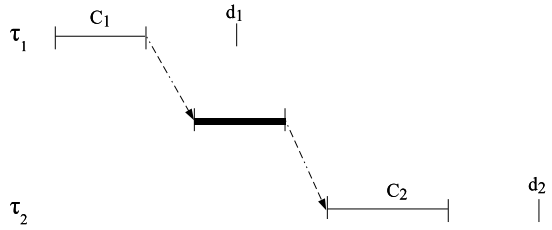

Fig. 7. Coprocessor Task Division

coprocessor service method after it has gained exclusive access through the coprocessor setup method.

An algorithm for the feasibility analysis of hybrid task sets scheduled under EDF is described by Stankovic *et. al.* in [11]. This algorithm is for uni-processors and is not compatible with the coprocessor problem presented above which is a combined job-shop and computer scheduling problem. Development of an algorithm to analyze feasibility under EDF is needed to make the coprocessor model viable for hard real-time problems.

Such a feasibility algorithm will also need to consider kernel overhead. For the idle engine application, the kernel is invoked with an average frequency 1390 Hz with an average execution time of $54.36\mu s$. This time includes context switches, message passing and task releasing/blocking, and scheduling. This represents $\sim 7.5\%$ of the processor time which could impact the feasibility of a task set.

It should be noted that the modified deadline is not required for aperiodic tasks triggered by auto-processors. They are not released until the auto-processor is ready for service and so the execution time of the auto-processor does not need to be accounted for in the associated software task.

The kernel does not directly support a wait of arbitrary duration (i. e. there is a real-time clock but no system timer). However, this can be implemented by using a coprocessor object to encapsulate a timer device. The worst-case execution time of the coprocessor is set to the duration of the longest wait required by the task. The deadline of the task is modified as above for a task using a coprocessor.

A. Shared Coprocessors

A coprocessor may be shared by multiple tasks. For example, in the idle engine test-case the cordic algorithm can be extended to calculate not only cosine but other elementary functions such as $\cosh$, $\exp$, and square-root [12]. Task FSM+DES calls the C sqrt() library function. It can be replaced with an invocation of the cordic coprocessor. The worst-case execution times are shown in Table III. As a result, the minimum simulation time slice becomes $0.00384s$, which is slightly larger than without the cordic squareroot ($0.00383s$). Factors contributing to this unexpected result are list below.

1) Worst-case execution time (on the ISP) for Task FSM+DES decreases by $0.000657 - 0.000533 = 0.000124s$, not the $0.000346s$ of the sqrt() function. This is because there is overhead for setting up and servicing the coprocessor.
2) There is added kernel overhead of 2 extra context switches when the task sleeps and when it is re-enabled.
3) Overall time (including coprocessor time) for Task FSM+DES becomes $0.000533 + 0.000167 = 0.000700s$, which is greater than the time of the software-only solution ($0.000657s$).
4) There may be contention for the coprocessor since both Task FSM+DES and Task Environment share it.

TABLE III
SQUAREROOT WORST-CASE EXECUTION TIMES

sqrt() func		0.000346 s
cordic sqrt		0.000167 s
FSM+DES	with sqrt()	0.000657 s
	with cordic	0.000533 s

In this instance, the addition of a coprocessor did not result in speed-up, even though the worst-case execution time (on the ISP) for the task decreased.

In the cases where it is beneficial for multiple tasks to share a coprocessor, the coprocessor must be scheduled in conjunction with the ISP. This is an interesting problem for 2 reasons. First, the ISP is preemptable, while the coprocessor is not. Second, it is the execution order of software tasks that determines the order in which jobs are released for the coprocessor. The coprocessor jobs are therefore released in the EDF order of their associated software tasks. However, once released the coprocessor jobs may be scheduled in one of several ways: for example FIFO order, fixed priority or EDF. In essence, it combines job-shop scheduling (the job starts on the ISP, continues on the coprocessor, and returns to the ISP for completion) with computer scheduling (EDF scheduling of ISP). It is a subject for further investigation.

V. CONCLUSION

A real-time kernel has been presented with integrated support for hardware processors. The idle engine application demonstrates the use of kernel coprocessor objects. It demonstrates a novel approach to EDF scheduling of tasks using coprocessors by using modified deadlines.

Schedule analysis is not performed by the kernel. It merely schedules tasks by their deadline. This limits the run-time overhead, making the kernel more suitable to real-time systems. The application of this kernel to real-time applications will require the development of an algorithm for feasibility analysis under EDF scheduling. This analysis will have to analyze scheduling of periodic and sporadic tasks on a preemptable ISP in conjunction with scheduling of non preemptable coprocessors.

The kernel auto-processor and coprocessor objects have been designed to facilitate hardware-software partitioning of an embedded system. That is, kernel support for hardware processors is intended to aid in the integration of software and hardware components, by reducing the amount of software

synthesis needed for communication between software and hardware. However a partitioner that employs this kernel will require the schedule analysis proposed above, not only to verify a solution, but also to guide the partitioner in choosing components to migrate between software and hardware.

REFERENCES

[1] P. Chou, E. A. Walkup, and G. Borriello, "Scheduling for reactive real-time systems," *IEEE Micro*, vol. 14, no. 4, pp. 37–47, August 1994.

[2] F. Thoen, M. Cornero, G. Goossens, and H. D. Man, "Real-time multi-tasking in software synthesis for information processing systems," in *Eighth International Symposium on System Synthesis*. California: IEEE Computer Society Press, 1995, pp. 48–53.

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

[4] F. Balarin, M. Chiodo, A. Jurecska, L. Lavagno, B. Tabbara, and A. Sangiovanni-Vincentelli, "Automatic generation of a real-time operating system for embedded systems: Extended abstract," in *International Workshop on Hardware/Software Co-Design (CODES/CACHE)*, Braunschweig, Germany, 1997.

[5] J. R. Jackson, "Scheduling a production line to minimize maximum tardiness," Management Science Research Project, University of California, Los Angeles, Research Report 43, 1955.

[6] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli, "Scheduling for embedded real-time systems," *IEEE Design & Test of Computers*, pp. 71–82, 1998.

[7] A. Balluchi, L. Benvenuti, M. D. D. Benedetto, T. Villa, H. Wong-Toi, and A. L. Sangiovanni-Vincentelli, "Hybrid controller synthesis for idle speed management of an automotive engine," in *Proceedings of the American Control Conference*, Chicago, Illinois, 2000, pp. 1181–1185.

[8] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of IEEE Real-Time Systems Symposium*, 1990.

[9] J. Labetoulle, "Some theorems on real-time scheduling," in *Computer Architectures and Networks*, E. Gelembe and R. Mahl, Eds. North Holland Publishing Company, 1974.

[10] J. Volder, "The cordic trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, pp. 330–334, 1959.

[11] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[12] J. Walther, "A unified algorithm for elementary functions," in *Spring Joint Computer Conference*, 1971, pp. 379–385.