Integrating Power Management into Distributed Real-time Systems at Very Low Implementation Cost

Bita Gorjiara, Nader Bagherzadeh, Pai Chou

Department of Electrical Engineering and Computer Science
University of California, Irvine
{bgorjiar, nader, chou} @ece.uci.edu

Abstract

The development cost of low-power embedded systems can be significantly reduced by reusing legacy designs and applying proper modifications to meet the new power constraints. The proposed power management techniques in the literature for implementing distributed power managers in multi-processor systems are very costly in terms of hardware and software modifications. For example, extra software code for power management must be integrated into each component of the system. Furthermore, in order to turn on a component at a specific time/event, extra hardware timers and interrupt controllers must be added to each component along with proper software/device driver modifications.

In this paper, we propose a new centralized power management technique that reduces the power consumption of distributed real-time systems at very low implementation cost. Our power manager does not need software and hardware modifications of each individual component. Instead, it uses the model of the system/application to compute the schedule of turning on/off commands by dynamically simulating the system for a given application scenario. The dynamic simulation can be conservative to reflect the jitter in arrival time of events and/or variation in execution delay of tasks. We applied our power management technique to a distributed software-defined radio system and achieved significant energy savings (60% to 87%) at the cost of 1% energy consumed by the power manager itself, as verified by actual hardware measurements. Furthermore, our power manager reacts to the changes in *application scenario* (referred to as *mission*) within milliseconds.

1 Introduction

Reusing legacy embedded systems in new application domains can significantly reduce the design cost, especially if the volume of the product is low. However, if the new domain limits the amount of power consumption, then the system must be modified to meet the power constraints. For example, it may be very cost effective to reuse a software-defined radio system designed for large airplanes in small Unmanned Aerial Vehicles (UAVs). However, unlike large airplanes, UAVs have limited power budgets due to low fuel capacity. If the product volume is not high enough to justify the cost of redesigning a low-power system, then it is desirable to modify the legacy system for better power efficiency.

Many algorithms have been proposed for dynamic power management of systems. Time-out and predictive techniques are two popular approaches. In timeout, a resource goes to standby mode after staying idle for a specified duration. Whenever a new request arrives, the resource must be turned on, and the request must wait for completion of mode transition. Because of the transition overhead, the overall performance of the system may be affected. To improve performance and also to avoid energy waste caused by unnecessary timeouts, predictive algorithms have been proposed. These algorithms usually use the history of the idle durations to predict future idle lengths [2][4]; or learn the distribution of idle lengths and periodically update the threshold of mode transitions [6][8]. A survey of power management techniques can be found in [1]. Note that most of these power management techniques are applied to individual components.

For legacy systems, implementing distributed power managers based on the policies mentioned above is costly, because it requires redesigning the system hardware and software. Hardware timers and interrupt controllers must be added to each component to turn it on at a specific time or at arrival of a specific event. Additionally, software code of power manager must run on each component. Also, the application code must be modified to avoid system failure when a component tries to communicate with a standby component. On the other hand, today's systems may have high-power analog devices that need power management but cannot run any power manager code locally. Therefore, their power mode must be controlled by another component. In legacy systems, the cost of changing hardware and software, and then verifying the correctness and reliability of the system is very high.

Compared to distributed power management techniques, centralized power managers are less costly to implement. However they can waste significant energy and bus bandwidth for communication between resources and the power manager. To reduce such communications, Li et al [10] suggest considering mode dependency between components. The power management turns on and off the dependent components together, and as a result eliminates extra communications between some components and the power manager. However, defining fixed mode dependencies, as proposed in their approach, is not always possible because the dependencies tend to vary in time based on the running application. Therefore, their technique is only applicable to simple systems with fixed dependencies. Also, the authors do not propose any general approach to extract the dependencies from the application or system architecture.

In this paper, we propose a new cost-effective centralized power management technique for distributed real-time systems. To avoid unnecessary communication between devices and the power manager, we dynamically simulate the system schedule based on high-level application and *mission-level* information as well as system architecture. The dynamic simulation can be conservative to reflect the jitter in arrival time of events or variations in execution delay of tasks due to data dependency. In real-time systems, usually the input events are periodic; however, the rate and type of the events may change dynamically based on the user demand or environmental changes. Our power manager can dynamically adjust the power commands for the new changes. Contributions of this paper include modeling of real-time systems for centralized dynamic power management and developing the power manager kernel that runs a fast high-level simulator to predict busy-time and idle-time of the resources at runtime. We applied our power management technique to a legacy software-defined radio system and achieved significant energy savings (60% to 87%). The energy savings computed using simulation was verified by actual measurement on the hardware implementation.

In this paper, Section 2 presents an overview of our approach. Section 3 discusses our model of computation and the amount of offline information needed for the online algorithm. Section 4 explains our online power management algorithm and Section 5 discusses our experimental results on the software-defined radio system, and the runtime overhead of the power manager.

2 Overview of our approach

Conventional power management techniques try to predict the length of idle durations, in order to avoid unnecessary mode transitions. However, in our approach, which is targeted for domain-specific real-time systems, we extract the busy-times and idle-times of resources using high-level application and communication knowledge. The behavior of embedded systems can be captured using Communicating Sequential Processes (CSP) [4]. In CSP, sequential processes communicate with each other for synchronization and/or data exchange. In system implementation, one or more processes may be mapped to a resource. Resources may have local operating systems and hence, local scheduling algorithms that determine the execution order of the mapped processes. In CSP, the complete functionality of the system is described usually in a high-level language such as SpecC [3] or SystemC [16]. However, for power management, we are interested only in timing and dependencies of tasks as opposed to their functionalities. In our model, we abstract away the functionality and capture timing and dependencies using a *task graph* model. Furthermore, embedded systems may provide more than one type of service. Therefore, in our model, each service type is captured using a separate task graph. Based on the user's decisions and/or surrounding situation, a subset of the services may simultaneously become activated on a system.

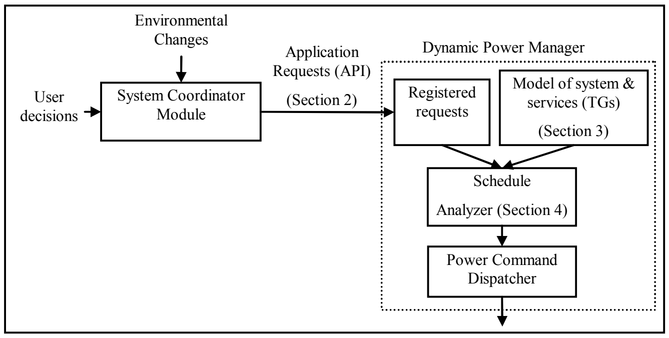

Figure 1. Block diagram of the proposed power manager

Our power manager uses the knowledge of system structure, running processes and task graphs to dynamically simulate and predict the schedule of tasks on the system, and extract the idle times of the components. Note that the real schedule of tasks on a system may *deviate* from the anticipated schedule due to existence of jitter in arrival time of the external events or variation in execution delay of tasks. This deviation can cause the power manager to sometimes turn off a resource when running a task. To address this issue, we add a *safety margin* to the computated of schedule in order to lengthen the ON duration of the resources. In Section 5, we discuss the amount of energy penalty caused by adding the safety margins.

Figure 1 shows the block diagram of our proposed power manager. The power manager captures the model of the system and the task graph of services at design time. At runtime, the user makes high-level decisions and a *System Coordinator Module* translates the decisions to application *requests*. Each request is represented by its service type (i.e. a task graph), period and deadline. Using a simple API, the coordinator module may *register* a new request or *cancel* a previously registered one. Then, the power manager is called to analyze the schedule of the system for the currently registered requests. The power manager generates power commands that are periodically applied to the system as long as the system maintains its status. In order to avoid any incorrect mode change, some of the resources may be kept on while power manager is analyzing the new schedule.

The following example helps to understand the role of the System Coordinator and the API: suppose that a software-defined radio system provides different waveforms of wireless links to many digital devices including two cameras. The user chooses to enable either one or both of the cameras and stream the video on one or more waveforms. Also, the user may select the picture refreshing rate, the resolution, and level of encryption. The System Coordinator Module uses the above information to determine the type of service (i.e. task graph) and the rate of the requests that the system must process in future. Then, it runs the power manager to generate the power commands. After a few hours, the user may stop streaming the video or change the settings. In that case, the coordinator module cancels the previous requests and registers the new ones. Then it runs the power manager again to re-generate the power commands.

3 Modeling system and services

This section presents our model of computation and the amount of offline information needed for online power manager. The model commonly used for capturing embedded systems is Communicating Sequential Processes (CSP) [4] model. We use the CSP as the base of our model and simplify it for dynamic power manager by replacing the functionality with tasks. The task graph model presented here can be extracted from CSP descriptions in SpecC and SystemC using a profiler.

A heterogeneous system is usually composed of a set of analog and digital resources. We model a system S with $S(R, P, B, G)$, where R is the set of resources, P is the set of processes, B is the set of buffers, and G is the set of services (task graphs). For each resource $r \in R$ we define an ON mode, denoted by $om(r)$, a set of idle modes denoted by $IM(r)$ and a local scheduling algorithm denoted by $SA(r)$. The scheduling algorithm can be First-In First-Out (FIFO), Rate Monotonic, Earliest Deadline First (EDF), etc. Each $im \in IM(r)$ is represented by its power consumption, shown by $pwr(im)$, and timing overhead of mode transition from om to im and from im to om, shown by $ovT1(im)$ and $ovT2(im)$, respectively. The energy overhead of the transitions are denoted by $ovE1(im)$ and $ovE2(im)$. Also, each process is mapped to a resource:

$$\forall \pi \in P,\ res(\pi) = r \text{ where, } r \in R$$

Additionally, each process has an input buffer:

$$\forall \pi \in P,\ buff(\pi) = b \text{ where, } b \in B$$

Usually buffers are implemented by memory units for processors and by digital buffers for hardware accelerators and buses. Analog devices and ASICs have a single default process with a limited buffer size.

Each service $TG \in G$ is modeled by a task graph $TG(T, D, E)$, where, T is the set of tasks, D is the set of edges of the graph, and E is the set of events that triggers the tasks. For each task $\tau \in T$, $process(\tau) \in P$ is the process that runs τ, and $\delta(\tau)$ represents its execution delay. The edges of the graph capture the dependencies and execution order of the tasks:

$$D = \{(\tau 1, \tau 2) \mid \tau 1, \tau 2 \in T, \tau 2 \text{ is dependent on } \tau 1\}$$

Each event $e \in E$ can trigger only one task denoted by $\theta(e) \in T$. However, in general, a task may need to receive more than one event before it can be executed. The set of input events of a task τ is denoted by $\omega(\tau) \subset E$. After execution of a task τ, a set of events must be dispatched to trigger its successor tasks. The output events of τ are represented by $\chi \subset E$ and are formally defined as follows:

$$\forall \tau \in T,\ \chi(\tau) = \{e \mid (\tau, \theta(e)) \in D\}$$

We also define *root event* of a graph as follows:

$$rootEv(TG) = \omega(\tau_r) \text{ where, } \forall \tau \in T,\ (\tau, \tau_r) \notin D$$

In task graph model, serialization, concurrency and synchronization of tasks can be captured using the events.

3.1 System modeling example

Suppose that we have a system consisting of a General-Purpose Processor (GPP) and two ASICs. The system processes a periodic request of type *request*1 with period of 1000ms and deadline of 600ms.

Also, suppose that whenever a new request arrives, the system behaves as follows (shown in Figure 2): the GPP does some pre-processing (denoted by task *n*1) and initiates another task (*n*2) on ASIC1. Then, GPP performs additional processing (*n*3) and waits for the result of *n*2. As soon as the execution of the task *n*2 finishes, the processor handles the results (*n*4) and initiates two other tasks (*n*5 and *n*6) on ASIC1 and ASIC2. Finally, the processor receives the output of *n*5 and *n*6 and finishes the processing (*n*7). This kind of interaction between resources is very common in heterogeneous systems composed of both hardware and software components. It is worth noting that although the request is periodic, GPP and ASIC1 do not process regular periodic tasks.

Suppose that in this example, the GPP communicates to ASIC1 via a shared memory while it communicates to ASIC2 through a bus. Also, suppose that there are two software processes running on the GPP: $process_1$ and $process_2$. Also, a fixed priority scheduling algorithm is used that gives higher priority to $process_1$. Figure 3 shows the system model using the elements presented in this section. For each process, a buffer exists that stores incoming events. Note that buffers *B*3 and *B*1 may actually be implemented using a single shared memory unit. However, they are shown as separate entities in this model. In this figure, buffers *B*5 and *B*6 are bus interface buffers.

Figure 4 shows the task graphs of the service TG_1. Figure 5 shows the execution delay of the tasks in Figure 4, and their corresponding processes and resources. In TG_1, tasks *c*1 and *c*2 are added to represent the bus delay. Note that in general, several services can be described for a system using task graphs. However, the user may request a subset of the services at a given time. For example, *request*1 activates service TG_1 with period of 1000ms and deadline of 600ms. Using the above model, the dynamic power manager analyzes the schedule of the tasks on the resources and produces a timing diagram similar to that of Figure 2. Then it generates the power commands based on the length of the idle intervals and the mode-transition overheads.

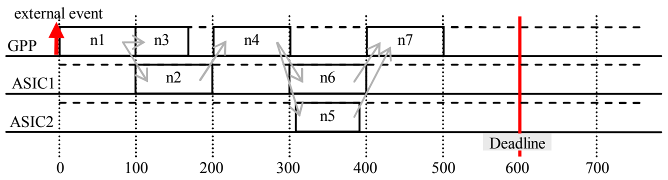

Figure 2. Schedule of tasks on a system servicing *request*1

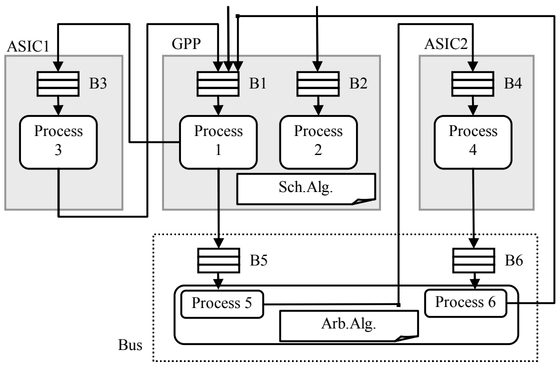

Figure 3. System modeling example

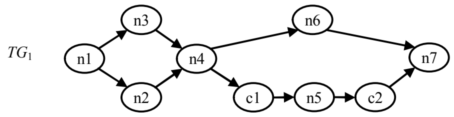

Figure 4. Services supported by the system of Figure 3

Tasks	Exec. Delay	Process	Resource
n1	100	Process1	GPP
n2	100	Process3	ASIC1
n3	30	Process1	GPP
n4	100	Process1	GPP
n5	80	Process4	ASIC2
n6	100	Process3	ASIC1
n7	100	Process1	GPP
c1	10	Process5	Bus
c2	10	Process6	Bus

Figure 5. Tasks of the task graphs

4 Our dynamic power-management algorithm

Our dynamic power management algorithm requires a discrete event simulation engine for high-level simulation of systems. Although discrete event simulations can be very slow for low-level system models, they tend to run very fast for high-level models due to relatively low numbers of components and events. The inputs to our algorithm are the system model and the application requests. The output of the algorithm is a sequence of power commands usually generated for the hyper-period duration of registered requests. The hyper-period duration of a set of requests is the Least Common Multiple (LCM) of their periods. Figure 6 shows the pseudo code of the power management algorithm. The code consists of modules and procedures. Modules are parallel entities that may wait for certain events before continuing their executions. In module Module_DPM_Alg, after calculation of the hyper-period, all task graphs that must be processed during the period are generated. Then, the root events of the task graphs are extracted (line 4). At the arrival time of each request, its *rootEv* is added to the buffer of the process that must run θ(*rootEv*). After dispatching all root events, the power manager algorithm waits until the end of simulation (line 8) and then collects and sorts the power commands generated during the simulation. The rest of the procedures and modules explain how power commands are actually generated.

In our algorithm, for each resource and process in the system, a module is created. In *Module_Resource*, a resource waits until it is activated by either a timing interrupt or by receiving an event (depending on its scheduling algorithm). Then, using the *selectNextActiveProcess*() the next active process is selected for the resource. This procedure uses the resource's local scheduling algorithm (SA) to select from the list of *ready* processes. A process is considered ready if it has one or more events in its input buffer. If no ready process exists, then the resource goes to an idle state (line 17). However, if a ready process exists, then depending on the previous state of the resource two situations may happen: (1) resource has been idle or (2) the resource was already running another process. In the first case, the resource leaves the idle state (line20), and in the second case, the current active process is preempted (line 22). At the end, the resource resumes the *nextActiveProcess*.

```
 1 Module_DPM_Alg (reqs, resources, commands, safetyMargin){
   // inputs: application requests (TG, period) ,
   //         resources (active mode, idle modes, functions)
   //         TGs (nodes, edges, resources, execDelays, processes)
   //         safetyMargin
   // output: commands (resource, mode, issue time)
 2     hp = calcHyperPeriod(reqs)  // for long hyper-periods, break it to smaller
       intervals
 3     TGs = generateAllTGs(reqs, hp)
 4     events = generateAllRootEvents(TGs);
 5     for-each rootEv in events
 6         waitTill(rootEv.arrivalTime)
 7         buff( process(θ(rootEv))).add(rootEv)  //dispatching the events
 8     waitTill(hp)
 9     for-each resource in resources
10         commands = Union(commands, getCommand(resource))
11     sort commands based on their issue time
12 }
   Module_Resource (resource res) {
13     while (1)
14         wait for activation
15         nextActiveProcess = selectNextActiveProcess(res, SA(res))
16         if(nextActiveProcess=null)
17             if(res.activeProcess != null) onIdleStarted(res, NOW)
18             res.activeProcess = null
19         else
20             if(res.activeProcess=null) onIdleFinished(res, NOW)
21             else if(res.activeProcess != nextActiveProcess)
22                 preempt(res.activeProcess)
23             res.activeProcess=nextActiveProcess
24             resume(res.activeProcess)
25 }
26 onIdleStarted(resource res, Time NOW){
27     res.idleStartTime = NOW
28 }
29 onIdleFinished(resource res, Time NOW) {
30     idleDur= NOW – res.idleStartTime – 2× safetyMargin  //calculate idle duration
31     Select $im \in IM(res)$ so that $idleDur > ovT1(im) + ovT2(im)$ and
       $idleDur \times pwr(om(res)) – (ovE1(im)+ovE2(im)+pwr(im) \times (idleDur–ovT1(im)–ovT2(im))$ is maximized
32     if (im != null)
33         t1 = res.idleStartTime + safetyMargin    //issue time of stdby command
34         t2 = NOW – ovT2(im) – safetyMargin       // issue time of ON command
35         c1 = new command(res, im, t1)            // stdby command
36         c2 = new command(res, om, t2)            // ON command
37     addCommands(res, c1)
38     addCommands(res, c2)
39 }
   Module_Process (process π) {
40   while (1)
41       if( buff( π ) is empty or preempted ) wait to be resumed
42       e = getNextEv( buff( π ) )
43       τ = θ (e)
44       τ.triggeredEvList.add(e)
45       if(τ.triggeredEvList = ω (τ))
46           startTime = NOW
47           waitTill(δ (τ) + NOW) or preemption    //interrupt the wait if preempted
48           if(preempted)
49               δ (τ) = δ (τ) - (NOW - startTime)
50           else    //not preempted
51               for-each event in χ (τ)
52                   buff(process(θ (event))).add(event)  //dispatching the events
53 }
```

Figure 6. Pseudo code of our power management algorithm

Whenever a resource leaves the idle state (*onIdleFinished* procedure), based on the length of the idle duration, the specified safety margin, and the available standby modes, the power manager decides about mode transition (line 31). If any suitable transition was possible, the power manager adds two commands to shutdown (line 35) and turn on the resource (line 36) at appropriate times.

In *Module_Process*, a process π waits until it is resumed. Then, it reads an event from its buffer and obtains the task τ triggered by the event (line 43). Task τ may need to receive several other events before it can be executed. To keep track of the events, we use *triggerdEvList* to store the events that have been dispatched for task τ. The list is complete when all the events in ω(τ) are received. If the list is not completed yet, the process skips this event; otherwise, it simulates processing of task τ by waiting for δ(τ) (line 47). If the process is preempted by the Module_Resource, then the wait is interrupted, the remaining execution delay of task τ is computed, and the process must wait to be resumed once again. If not preempted, the process finishes the execution of task τ and dispatches its output events from set χ(τ) (line 52).

In cases that calculated hyper-period is very long or the requests change very often, the computation can be performed for a smaller window of time instead of the entire hyper-period. In that case, always the schedule of the next window must be computed before the end of the current window.

5 Experimental results

In this section, the results of applying our power management technique to a multi-channel software-defined radio system [9] are presented. The radio was originally designed for large airplanes and was modified for small Unmanned Aerial Vehicles (UAVs). UAVs are small airplanes that are usually remotely piloted and can carry cameras, sensors and communications equipment. The radio sends and receives many real-time messages used to control and monitor the aircraft. The energy budget in this system is constrained due to the limitation on the amount of fuel that the small aircraft can carry.

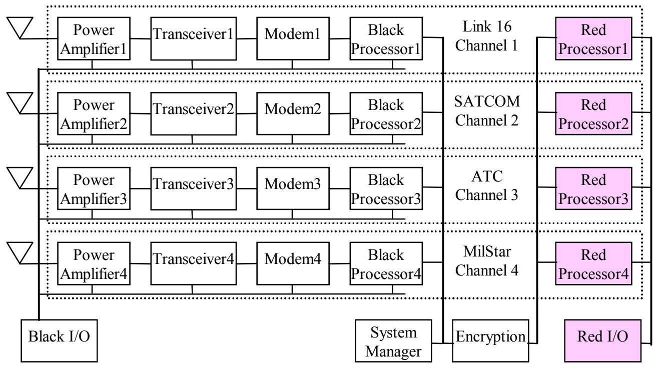

Figure 7. The software-defined radio system [9]

The system (Figure 7) has four channels that each processes a specific waveform (e.g. Link16, SATCOM, ATC and MilStar). Each channel has two general-purpose processors (called Black and Red processors), a modem, a transceiver, and a power amplifier (PA) to process network protocols, modulate/demodulate, convert to RF/baseband, and transmit/receive signals, respectively. Among these components, PA and transceiver are analog while the rest are digital processing elements. The Black and Red Processors run a real-time OS that gives a higher priority to received messages over the ones being sent. Some of the messages are critical and need to be encrypted before being sent or decrypted after being received. We call them classified messages. These messages must go through the shared Encryption unit, Red Processors and Red I/O. Non-classified messages go through Black I/O instead. The power manager and system coordinator modules reside on a separate processor, called System Manager, outside the channels. Table 1 shows the characteristics of each component in terms of its power modes and transition overhead based on the measurements on the modified radio system [9].

Note that mode dependency model proposed in [10] is not sufficient for this system because of the existence of shared resources in the paths. For example, no correct mode dependency can be defined between the Encryption unit and Black Processor1 (BP1) because they must be turned on together when Channel-1 is used and they should not be turned on together when other channels are used, or when the message is not classified.

To model the system for our power management, we need to capture the task graphs of the system corresponding to the services that it provides. Figure 8 shows the task graph of sending a non-classified message on Channel-1, and the corresponding timing diagram of the system. The message arrives at the Black I/O and, after some initial processing, is passed to the Black Processor1. The Black Processor1 handles the communication protocol and eventually streams the message to Modem1, Transceiver1 and PA1. The execution delay of each task is specified in the task graph model. In this system, there are four channels that each can send and receive classified and non-classified messages. Therefore, the total number of task graphs in the system is 16 (i.e. 4×2×2).

Table 1. Power modes of different components [9]

	ON mode	StdbyMode		
Components	**Power (W)**	**Power (W)**	**ON-Stdby (ms)**	**Stdby-ON (ms)**
Red/Black IO	5	1	50	1
Black Proc.	6	1	50	1
Modem	4	1	50	1
Transceiver	25	0.1	50	2
PA	10	1	50	2
Encryption	10	2	100	5
Red Processor	10	2	50	1
System Mangr.	16	1	50	1

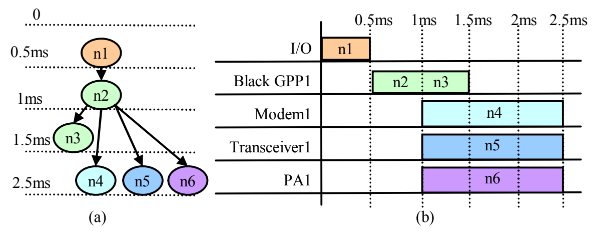

Figure 8. Sending a non-classified message on Channel 1: (a) task graph, (b) schedule of tasks on the components

To study different aspects of the system, we modeled it in SystemC and used state-based power estimation technique [12] to estimate the amount of energy consumption. However, we implemented our final power manager on the actual hardware as well and used measurements to confirm the energy savings.

We modeled this system in SystemC once without any power manager (No-PM) and once with our application-based power manager (APM). The inputs of the simulator are requests (messages) and its outputs are total energy consumption, and number of lost events. We assume that events are lost if they arrive when resources are in standby mode or in mode transition. For our application the tolerable event loss is 1% or less.

As our testbench, we used the actual communication profile of the radio system recorded during a ten-hour test mission. The profile contains more than 300000 messages. Different messages arrive at the system with different rates, and the rate and type of the messages change in different phases of a mission. For example, during take off and landing of the aircraft, the rates of certain messages increases, while during the actual flight, their rates drop. Without any power management (No-PM) and excluding the System Manager component, the system consumes 7.92MJ during the ten-hour mission. Using an *ideal* power manager, a power manager that exactly knows the arrival time of all the messages, the energy consumption is reduced to 0.95MJ achieving 88% energy saving.

Table 2. The event loss percentage and energy consumption of our approach (APM) for different jitter and safety margin values

Safety Margin (ms)	Jitter (ms)					Energy saving (%)
	10	50	100	200	300	
5	0.32	0.44	11.21	28.79	36.06	87.8
7	0.08	0.26	10.66	27.87	35.79	87.6
10	0	0.32	9.3	27.48	35.19	87.2
20	0	0.12	9.3	27.48	35.19	87.2
30	0	0.04	3.3	22.62	31.81	84.9
40	0	0.08	1.35	20.16	30.14	83.8
50	0	0	0.04	17.57	28.43	82.7
60	0	0	0.08	15.11	26.56	81.5
80	0	0	0	10.5	23.34	79.3
100	0	0	0	6.56	20.56	77.0
120	0	0	0	2.86	16.82	74.8
140	0	0	0	0.52	13.36	72.5
160	0	0	0	0.04	10.62	70.2
180	0	0	0	0	7.16	68.0
200	0	0	0	0	4.37	65.7
220	0	0	0	0	1.99	63.5
240	0	0	0	0	0.44	61.2
260	0	0	0	0	0.04	59.0
280	0	0	0	0	0.04	56.7
300	0	0	0	0	0	54.4
320	0	0	0	0	0	52.2

To evaluate our power management technique, we ran our experiments assuming variations in the execution delay of tasks and jitter in the arrival times of events. Since the variations of execution delay can also be modeled by event jitter, here we only focus on the event jitter. In our approach (APM), if the events do not have any jitter, then the ideal energy savings of 88% can be achieved with 0% event loss. However, if the events have jitter, then a safety margin is added to the computation of idle durations to keep the event loss low. The amount of safety margin is selected based on the amount of jitter. To find the appropriated safety margin, we ran the SystemC simulation model with different safety margins and jitter values. Table 2 shows the rate of event loss and the amount of energy savings computed for different jitter and safety margin values. The event loss drops fast by adding the safety margin. However, the energy savings also decrease linearly. Adding a safety margin equivalent to half of the jitter value can reduce the event loss to 10% or less. For our application the tolerable event loss is less than 1%. Therefore, for each jitter value, we select the safety margin that reduces the event loss to less than 1% and show the corresponding energy savings in Table 3. For a jitter of 10-50ms the energy savings is as high as 87.8% while it reduces to 82.7%, 72.5% and 61.2% for jitter values of 100ms, 200ms and 300ms respectively.

We ran the power manager with a conservative safety margin of 140ms on the actual hardware and could achieve 68% energy savings according to the actual measurement. This shows less than 5% error in our power model in SystemC.

5.1 Runtime overhead of our power manager

To evaluate the cost of the power manager, we ran it on System Manager processor (PowerPC 500MHz, 256MB RAM, 16W) for the entire ten-hour mission. The total execution time of our power manager is nine minutes. Therefore, on average, for every 80 seconds of the mission, one second of computation is performed by the power manager. This computation accounts for 8.6KJ energy consumption, which is less than 1% of the total energy consumption of the system with APM. This clearly shows that low-cost centralized power managers can be implemented on legacy systems and achieve significant energy savings with reasonable overhead.

To further reduce the overhead of power manager, we suggest designing custom hardware on an FPGA that implements the power manager. In that case, the high-level model of the system can be emulated using hardware.

Table 3. The result of APM approach for different jitter values

Jitter (ms)	Min. safety margin (ms) (less than 1% event loss)	Energy saving (%)
10	5	87.8
50	5	87.8
100	40	82.7
200	140	72.5
300	240	61.2

6 Conclusion

In this paper, we present a new cost-effective centralized dynamic power management technique for legacy real-time systems. In our approach, we employ the high-level application and communication knowledge as well as future workload information to anticipate the idle intervals of the components using a low-cost scheduling analysis technique. Using this technique, the required communications between resources and the central power manager is reduced significantly. In real-time systems, usually the input events are periodic, however, the rate and type of the events may change based on the user demand or environmental changes. Our power manager can dynamically adjust the power commands for the new changes. Also, our power manager takes into account the possibility of jitter in the arrival time of external events as well as the variation in execution time of tasks. Our experimental results on a software-defined radio system show that our technique can achieve 60% to 87% energy savings while spending 1% energy overhead for running the power manager.

References

[1] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management". *IEEE Transactions on VLSI Systems*, VOL.8, NO.3,2000

[2] E. Chung, L. Benini, and G. De Micheli. "Dynamic power management using adaptive learning tree". *ICCAD,* 1999.

[3] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC: Specification Language and Methodology", Kluwer Academic Publishers, Boston, MA, ISBN 0-7923-7822-9, 2000.

[4] C. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.

[5] C. Hwang and A. Wu, "A Predictive System Shutdown Method for Energy Saving of Event-Driven Computation", *ICCAD* 1997.

[6] S. Irani, S. Shukla, and R. Gupta. "Online strategies for dynamic power management in systems with multiple power-saving states". *ACM Transactions on Embedded Computing Systems*, 2003.

[7] M. Jersak, R. Ernst, "Enabling Scheduling Analysis of Heterogeneous Systems with Multi-Rate Data Dependencies and Rate Intervals". In *Proc. DAC*, 2003.

[8] P. Kachroo, S. Shukla, T. Erbes, and H. Patel. "Stochastic learning feedback hybrid automata for power management in embedded systems". *In IEEE International Workshop on Soft Computing in Industrial Applications*, 2003.

[9] S. Koenck, B. Getz, "JTRS Resources and Relational Behavior Definition", *DARPA Power Aware Computing and Communications*, Rockwell Collins Inc., F33615-02-C-4000, Nov. 2002.

[10] D. Li, Q. Xie, P. Chou, "Scalable Modeling and Optimization of Mode Transitions Based on Decoupled Power Management Architecture", Design Automation Conference (DAC'03), 2003.

[11] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes," *DAC*, 1999.

[12] R. Bergamaschi, Y. Jiang, "State-based power analysis for systems-on-chip", In *Proc. of IEEE Design Automation Conference* (DAC)*,* 2003.

[13] T. Simunic, L. Benini, G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems", in *Proc. Design Automation Conference* (DAC) 1999.

[14] M. Srivastava, A. Chandrakasan. R. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation", IEEE Transactions on VLSI Systems, Vol. 4, No. 1 (1996), pp. 42-55.

[15] Q. Wu, Q. Qiu and M. Pedram, "Dynamic power management of complex systems using generalized stochastic Petri nets". In Proc. IEEE Design Automation Conference (DAC'00), 2000, pp. 352-356.

[16] www.systemc.org.

[17] www.specC.org