An Algorithm to Avoid Power Command Jitter in Middleware-Based Distributed Embedded Systems

Bita Gorjiara, Pai Chou, Nader Bagherzadeh

Technical Report CECS-03-47
July 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425
(949) 824- 2481
{bgorjiar, chou, nader}@ece.uci.edu

An Algorithm to Avoid Power Command Jitter in Middleware-Based Distributed Embedded Systems

Bita Gorjiara, Pai Chou, Nader Bagherzadeh

Technical Report CECS-03-47
July 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425
(949) 824- 2481
{bgorjiar, chou, nader}@ece.uci.edu

Abstract

Middleware such as CORBA provides a software architecture that supports integration of legacy software components with new software in a way that is modular, scalable, and evolvable. However, these benefits come with high run-time overhead. In dynamic hard real-time distributed embedded systems, usually a central power manager calculates and issues all the power commands. The power manager must communicate with different modules to transparently perform mode transitions. Due to inherent communication overhead of middleware based embedded systems, issuing each power command will have considerable overhead for the power manager. This overhead limits the rate of issuing power commands and may cause a shift in their schedule. This paper has two contributions; first, it introduces the Power Command Jitter (PCJ) problem in middleware based embedded systems; second, it proposes an effective Power Command Adjustment algorithm (PCA) that re-orders and reschedules the power commands so that the correctness of the schedule is maintained while minimizing the energy loss. Our experimental results on a commercial software defined radio system (JTRS) shows PCJ can cause violation of real-time deadlines and unreliability of the system. Moreover, in presence of power command issue overhead, even as low as 3ms, the power manager can lose 25% of its energy savings.

Contents

List of Figures

An Algorithm to Avoid Power Command Jitter in Middleware-Based Distributed Embedded Systems

Bita Gorjiara, Pai Chou, Nader Bagherzadeh
University of California, Irvine
Irvine, CA 92697-3425
(949) 824- 2481
{bgorjiar, chou, nader}@ece.uci.edu

Abstract

Middleware such as CORBA provides a software architecture that supports integration of legacy software components with new software in a way that is modular, scalable, and evolvable. However, these benefits come with high run-time overhead. In dynamic hard real-time distributed embedded systems, usually a central power manager calculates and issues all the power commands. The power manager must communicate with different modules to transparently perform mode transitions. Due to inherent communication overhead of middleware based embedded systems, issuing each power command will have considerable overhead for the power manager. This overhead limits the rate of issuing power commands and may cause a shift in their schedule. This paper has two contributions; first, it introduces the Power Command Jitter (PCJ) problem in middleware based embedded systems; second, it proposes an effective Power Command Adjustment algorithm (PCA) that re-orders and reschedules the power commands so that the correctness of the schedule is maintained while minimizing the energy loss. Our experimental results on a commercial software defined radio system (JTRS) shows PCJ can cause violation of real-time deadlines and unreliability of the system. Moreover, in presence of power command issue overhead, even as low as 3ms, the power manager can lose 25% of its energy savings.

1 Introduction

Distributed embedded systems are becoming more complex due to the demands for more features and performance. Since these systems usually contain several processing elements, developing software for them is challenging and expensive. To develop reusable and platform independent software modules, some system designers have incorporated more software layers, or middleware, to these distributed embedded systems [1][2]. Currently new middleware technology is under development that can provide real-time services, software fault tolerance, software re-configurability and mobility [3][4][5][6]. There is a growing trend towards using middleware services in different application domains, such as telecom, aerospace, military testing and training ranges.

To manage power in hard real-time distributed embedded systems, usually a central power manager calculates and issues all the power commands. In order to make the mode transitions transparent to the running software modules, the power manager must also issue commands to save and restore configuration and data, before or after each power command. For example, Figure 1 shows the process of issuing a shutdown command in a middleware-based distributed system, proposed in [10]. First, a shutdown command is sent to the processor. Next, the processor saves its current status. The *mode transition overhead* includes the overhead of saving the status. Finally, the power manager communicates with device manager to turn off the power supply. As shown, it takes several communications to perform a successful resource shutdown. Note that each middleware-based communication has overhead on the processors at both ends [7][8][9], due to the extra data checking and processing. Therefore, it takes a considerable amount of time for the power manager to issue each command. We call this overhead Power Command Issue Overhead (PCIO). In Figure 1, PCIO includes the delay of (1), (4) and (5). If PCIO is not considered in the scheduling of the commands, the power manager will arbitrarily serialize and shift them, which can cause missing real-time deadlines. Even if no real-time deadline is missed, it can cause system failure or unreliable behavior, which is due to wrong assumptions about the current mode of the resources. In presence of PCIO, every two consecutive commands will have a minimum time separation and as a result, PCIO limits the rate of issuing power commands. To correctly schedule a power command, PCIO, mode transition overhead and network transmission overhead must be considered. We call the overall delay Power Command Delay (PCD).

This paper introduces the Power Command Jitter (PCJ) problem in middleware-based hard real-time distributed embedded systems and proposes a Power Command Adjustment (PCA) algorithm to fix this problem. The algorithm reschedules or removes the proper power commands in order to keep the resources ON during their scheduled busy intervals. It also minimizes the energy loss by prioritizing the power commands. Our experimental results on a commercial software defined radio system shows that in presence of power command issue overhead (PCIO), even as low as 30μs, the power manager can loose 25% of its energy saving.

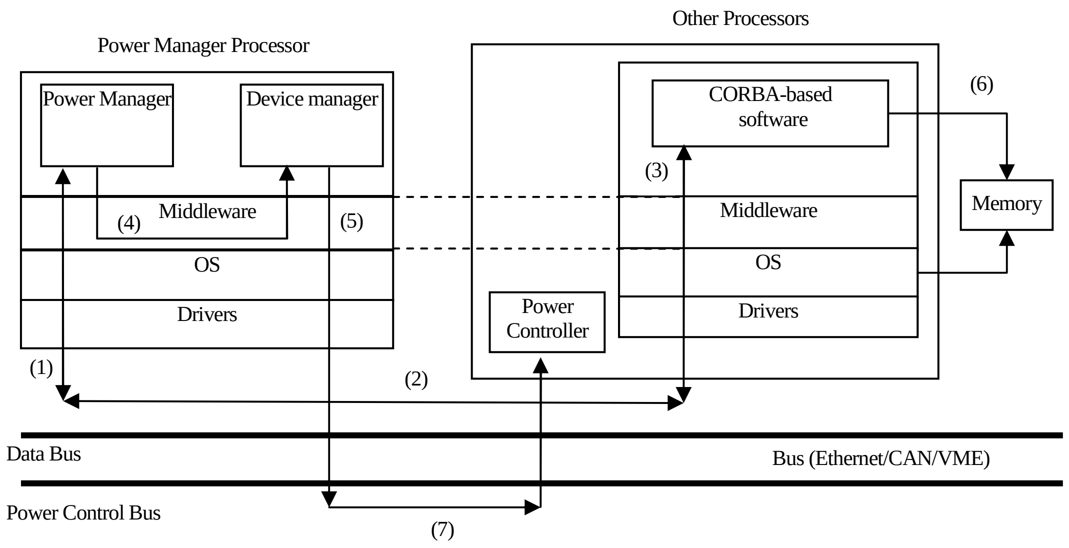


Figure 1. System architecture and the process of issuing a power command

The rest of the paper is organized as follows. Section 2 presents an introduction to middleware based embedded systems as well as previous work on system level power management. Section 3 introduces power command jitter problem using an example. Section 4 formulates this problem followed by the PCA algorithm described in Section 5. Section 6 shows the experimental results and Section 7 concludes the paper.

2 Background

2.1 Middleware-Based Embedded Systems

Middleware refers to a layer of software above the operating system and below the application. Its purpose is to provide a software architecture for integration of software components. Originally developed for general-purpose distributed computing, in recent years, middleware such as CORBA has been adopted in implementation of embedded systems including network routers, laser printers, software-defined radios, and other complex systems. The main benefits of middleware are modularity, evolvability, and scalability. Modularity comes from the inter-component communication services provided by the middleware, which decouples the object's calling interface from its implementation. Evolvability means parts of the software, including the power manager itself, may be upgraded smoothly without requiring a reboot. As software complexity increases, the ability to integrate legacy software quickly with an upgrade path has also motivated the adaptation of middleware in these systems.

A main challenge with such middleware has always been the high run-time overhead. This is because the middleware must perform many tasks in order to support the high-level abstraction. Communication between objects can be expensive because it involves a lookup for the object's location and packaging the parameters in a message. It is a challenge to make real-time guarantees, because the middleware operations can dominate the processor utilization, leaving much less time for application execution. Although advances in microprocessors have made it less of a problem, incorporating middleware still means higher power than without middleware. As a result, power management becomes particularly important for middleware-based embedded systems.

2.2 Power Management Techniques

Dynamic Power Management (DPM) and Dynamic Voltage Scaling (DVS) for parallel and distributed real-time systems have been studied by different researchers [11][12][13][14]. DPM usually means turning off (shutdown) or putting devices in non-operational low power modes (standby) when they are not in use; DVS means lowering the voltage (and frequency) of a component for higher energy efficiency when it does not need to run at its full speed. Stochastic DPM approaches are also applicable for non-real-time systems. However, they cannot be used in hard real-time systems due to their unreliable nature. Many approaches consider mode transition overhead and Power Command Delay (PCD) in scheduling of power commands. However, all of them assume that there is no PCIO, and the central power manager can issue as many commands as needed. Hong et al [15] takes into account the inherent limitation on the rates at which the voltage and clock frequency of a component can be changed by the power supply controllers and clock generators. However, they do not consider the system level constraint on power command rate caused by middleware overhead. The difference between these two constraints is that the first only limits the number of commands issued for one controllable resource and the second limits the total number of commands that can be issued for the entire system. As a result the second constraint (discussed in this paper) is sensitive to the size of the system. To the best of our knowledge, none of the previous researchers have considered this practical issue.

3 Motivating Example

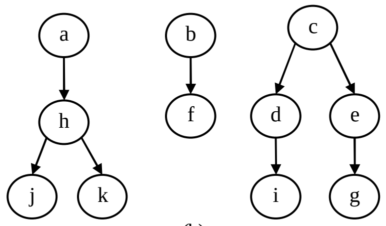
In this example we show that how PCIO can invalidate a schedule or limit the amount of energy saving achievable by DVS and DPM. We also show the importance of choosing and issuing the right set of commands.

3.1 *Example System under Power Management*

Assume we have a system composed of four processors for running tasks, and a separate control unit for changing the power state of processors at run time. The system supports both DVS and DPM and it has three DVS modes and one shutdown mode without any transition overhead. Figure 2(a) shows the power level of each mode. The application running on the system is represented as a set of periodic task graphs shown in Figure 2(b). The task graphs capture data dependencies between the tasks.

Modes	Power (mW)
DVS_m1	400
DVS_m2	200
DVS_m3	100
Shutdown	0

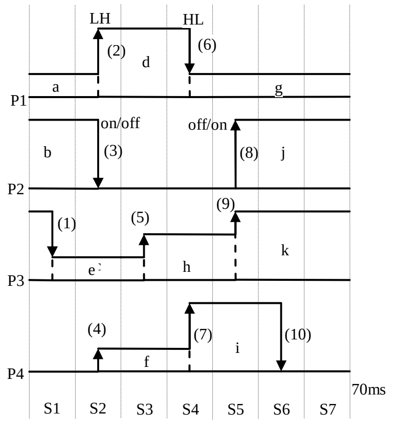
(a)



(b)

Figure 2. (a) Processor modes (b) Application Task Graphs

Figure 3 shows the schedule of tasks on four processors, where the height represents the power consumption of each processor. The period is 70ms and the same schedule is repeated in each period. In this case, 10 commands are needed to do both DVS and DPM, which is shown by up and down arrows. However, given a PCIO of 10ms, the power manager cannot issue the power commands at their scheduled time, because every two commands are separated at least by 10ms. Figure 4 shows the schedule of Arbitrarily Serialized Power Commands (ASPC) where, some commands are shifted to the next period and as a result the real-time deadline is missed. We refer to this schedule shift as Power Command Jitter. To avoid this problem the power manager must be limited to issuing up to seven power commands in each period. We define a Command Slot (CS) as the time interval in which only one command may be issued. In the example of Figure 3, the period (70ms) is divided into seven command slots (10ms each). Without power management, the system consumes 112000mJ (i.e. 4×70×400) for an entire iteration of the schedule. If all power commands could be issued at their scheduled time, then the amount of energy saving would be 49% compared to no power management ($100 - 100*(\Sigma^{k}_{i=a}\ \text{Area}_{i})\div 112000$).



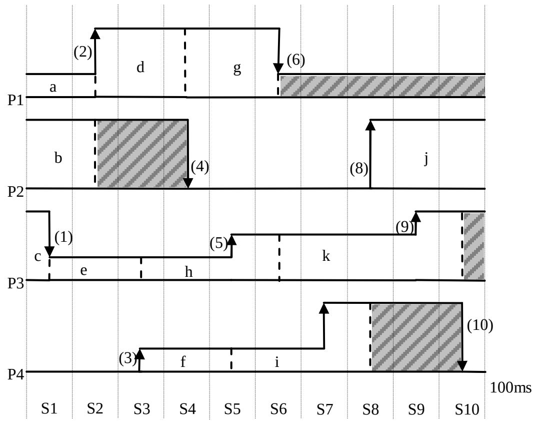


Figure 3. Task and power schedule

Figure 4. Arbitrarily serialized power commands

Since the amount of power and energy savings achieved by various power commands are different, an optimization algorithm is needed to select and keep the most beneficial commands. Moreover, these commands cannot just be dropped randomly, because a task cannot execute if we drop a command to wake up its associated resource. In order to maintain a valid schedule of tasks, removing one command may require adjustments to or removal of several other commands. Table 1 shows an example of required adjustments to commands when one of them is removed. It also shows the corresponding reduction in energy saving. For example removing command (8) disables execution of task *j*, because the processor is OFF. To fix this problem, we have to also remove command (3). As a result of removing both (3) and (8), the processor *p*2 will be ON during the entire execution time. In that case, the achieved energy saving is reduced by 10.7% (i.e. (5+20+5)×400÷112000). Removing command (4) is rather more complex because, it is an ON command coupled with the OFF command (10). To prevent elimination of task *f*, the command (10) should change from a shut-down command to a slow down command. Note that both the start and the end of a period should be in the same power mode, in order to make the schedule repeatable. Figure 5 shows the schedule after eliminating commands (3), (4), (8). Table 2 shows all the solutions and the amount of saving achieved by each. As one can see, a PCIO of 10ms can decrease the amount of energy saving from 49% down to 8.8% (solution 11) in this example.

Removed commands	required command changes	required command elimination	added energy overhead %
(2) or (6)		(6) or (2)	13.4
(3) or (8)		(8) or (3)	10.7
(5)	(1)		1.8
(9)	(5)		3.57
(1)	(5)		5.36
(1),(5)		(9)	8.93
(1),(9)		(5)	8.93
(9),(5)		(1)	8.93
(4)	(10)		2.68
(7)	(4)		5.36
(7),(4)		(10)	16.1

Table 1. Energy overhead of removing each command and the required changes

No	Removed Commands	Total amount of saving %
1	1, 4, 5, 7, 9, 10	24.9
2	1, 4, 5, 7, 10	21.32
3	1, 4, 5, 9, 10	15.96
4	1, 4, 5, 10	22.21
5	1, 5, 7, 9, 10	18.64
6	1, 2, 5, 7, 10	22.21
7	1, 2, 5, 6, 9	18.64
8	1, 2, 5, 6	22.21
9	1, 5, 7, 9, 10	22.21
10	1, 3, 5, 7, 8, 10	29.36
11	1, 5, 9	8.821

Table 2. All possible solutions and the amount of energy saving

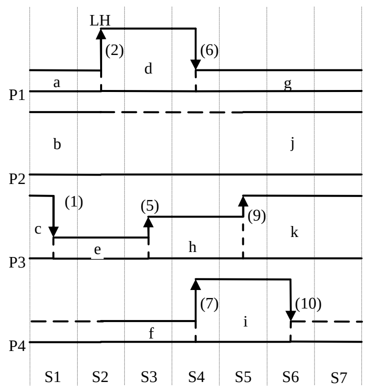


Figure 5. Schedule after removing (3), (4), (8)

Instead of eliminating conflicting commands, we may be able to schedule them in their neighboring empty slots. For sake of simplicity, in this paper, we consider only standby and shutdown commands. A similar approach is applicable in presence of DVS commands.

4 Problem Formulation

In this section, we formulate the problem of scheduling power commands in presence of PCIO. Assume that we have n idle intervals in the period P and the power manager can issue N commands (one in each slot). Each idle interval ω_i is associated with a processing element $proc(\omega_i)$ and is represented by (s_i, e_i) pairs, where,

- $s_i \in [0,P)$: start time of the idle interval, and
- $e_i \in [0,P)$: end time of the idle duration

where, $s_i < e_i$. The power command issue overhead is denoted by *PCIO*. Also, the command slot, in which an event η is scheduled, is denoted by $CS(\eta)\in[1, N]$. Moreover, the status of each command slot k is captured in $SL[k]\in\{0,1\}$, where 0 means the slot is available and 1 means it is already taken by another event.

Corresponding to each idle interval ω_i, we define a standby interval μ_i during which the proc(ω_i) may be shutdown. In addition to shutdown, other standby modes can also be used depending on the size of the μ_i. The standby interval μ_i is represented by:

- $v_i\in \{0,1\}$: to indicate whether the associated power commands are scheduled (1) or not (0). If not, the resource stays ON during the interval.
- $s'_i\in [0,P)$: start of standby interval μ_i (OFF/Standby command)
- $e'_i\in [1,P]$: end of standby duration μ_i (ON command)

To ensure the correctness of the scheduled tasks, the idle intervals must envelope the standby intervals. That is,

$$s_i \le s'_i < e'_i \le e_i \qquad (2)$$

Also, none of the scheduled commands share the same command slot. That is,

$$CS(s'_i) \ne CS(s'_j),\ CS(e'_i) \ne CS(e'_j)\ \forall i \ne j,\ v_i = v_j = 1 \qquad (3)$$

The objective is to schedule the standby intervals so that the total consumed energy is minimized, while none of the power commands is conflicting with others. The total energy is the sum of energy consumed during task execution, idle time and standby time.

$$E = E_{exec}+E_{idle}+E_{stby} \qquad (4)$$

Or

$$E = E_{exec} + \sum_{i=1}^{n} (1 - v_i) \cdot E_{idle}(e_i - s_i) + \sum_{i=1}^{n} v_i \cdot \begin{pmatrix} E_{idle}(s_i' - s_i) \\ + E_{stby}(e_i', s_i') \\ + E_{idle}(e_i - e_i') \end{pmatrix} \quad (5)$$

E_{idle} and E_{stby} are two functions that calculate the amount of consumed energy for a specified interval, assuming the corresponding resource is in idle and standby modes, respectively. E_{stby} also includes the transition energy of ON to the selected standby mode as well as of standby to ON mode. The standby mode transition cannot take place for small idle intervals, because the amount of energy saving will be less than transition energy. The smallest possible standby interval for mode change is denoted by *minST*.

5 Power Command Adjustment Algorithm

Figure 6 shows the pseudo code of the proposed Power Command Adjustment (PCA) algorithm for finding an optimized solution for the problem. The algorithm inputs a list of idle intervals *IIL_List* and then creates and initializes a list of standby intervals *SI_List*. Next, it pushes them to a priority queue *pQueue* that sorts the standby intervals based on their energy saving. Here, the intuition is to schedule the most energy saving standby intervals first and then schedule the rest if possible. The algorithm checks for schedulability of each standby interval. Whenever the command slot is not available to schedule s' and e', then the algorithm tries to resize the interval by checking the slots between the two (see RESIZE()). As the standby interval μ is resized, it should be re-prioritized and re-queued, and this can be repeated until an available command slot is found for both s' and e'. If no available slot is found or the standby interval is very short after resize, it will not be scheduled.

```
PowerCommandAdjustmentAlg(IIL_List, N, PCIO)
  for each i in 1 to N
      SL.Add(0)                          //all slots are initially available
  Create SI_List
  for each i in 1 to n
      SI_List[i].s′ = IIL_List[i].s      // initialize the start and end of standby interval
      SI_List[i].e′ = IIL_List[i].e
      SI_List[i].v = 0                   // initially no standby mode transition is done
  Create priority queue pQueue           // energy saving is the key value
  for each i in 1 to n
      key = CalculateEnergySaving(SI_List[i])
      pQueue.Push(SI_List[i],  key)      // sorts based on energy saving
  while (pQueue.Size() > 0)
      $\mu$ = pQueue.Pop()                    // selects the most beneficiary interval
      if (Schedulable(SL, $\mu$))
          Schedule(SL, $\mu$)
          $\mu$.v = 1                         // the power commands are scheduled for interval $\mu$
      else if (Resize(SL, $\mu$))
          key = CalculateEnergySaving($\mu$)           // recalculate the key after resize
          pQueue.Push($\mu$, key)
      else
          $\mu$.v = 0                         // no power command is scheduled for $\mu$
  totalEnergy = Calculate total energy using Equation (5)
  return totalEnergy, SI_List

Schedulable(SL, $\mu$)                        // checks if both start and end of interval $\mu$ can be scheduled
  if(CS($\mu$.s′) = 0 and CS($\mu$.e′) = 0)
      return true
  return false

Resize(SL, $\mu$)                             // relocates start and end event to available slots
  if (CS($\mu$.s′) = CS($\mu$.e′) )                // if no further resize is possible, returns false
      return false
  while(CS($\mu$.s′) != CS($\mu$.e′) and SL[CS($\mu$.s′)] =1)          // relocates start
      $\mu$.s′ = $\mu$.s′ + PCIO
  while(CS($\mu$.s′) != CS($\mu$.e′) and SL[CS($\mu$.e′)] =1)          // relocates end
      $\mu$.e′ = $\mu$.e′ − PCIO
  if($\mu$.e′ − $\mu$.s′ < minST)                // checks the minimum size condition
      return false
  return true

Schedule(SL, $\mu$)                           // allocate two slots corresponding to the start and end of $\mu$
  SL[CS($\mu$.s′)] = 1
  SL[CS($\mu$.e′)] = 1
```

Figure 6. Pseudo code of Power Command Adjustment Algorithm

6 Experimental Setup

To investigate the effect of Power Command Issue Overhead (PCIO) on different systems and applications, we have developed a high-level system simulator that simulates application execution and identifies the idle intervals of all the resources. Since we just need to extract resource utilization information, there is no need for a low-level simulator. As a result, the application model can be simplified to a set of dependent tasks with a fixed mapping to the resources. We also consider the communication delay in order to get a more precise timing. This section first explains our approach to modeling, simulation, and profiling of different system elements, and then presents experimental results.

6.1 Modeling

The system is modeled as a set of resources, communication channels and application. The resources include general-purpose processors, DSP processors, FPGAs and ASICs. Each resource model consists of a set of standby (power) modes as well as the specification of a local scheduling algorithm. The standby modes are represented by their power consumption and the delay of entering and exiting from them. The local scheduling algorithm of a resource defines the order of task execution whenever multiple tasks are simultaneously ready to be executed on that resource. Simple resources can use a First-In-First-Out (FIFO) algorithm, while complex resources can use Earliest Deadline First (EDF), Rate Monotonic, or other algorithms. Communication channels include both point-to-point connection and shared buses and they are represented by their bandwidth, and their energy per bit characteristics. The application is represented as a set of periodic Task Graphs (TG). Task graphs represent the order and the duration of resource contribution to accomplish a system-level task. Each node of the graph contains the execution time, and a reference to the mapped resource. The edges show the amount of the data that must be transferred between the tasks.

6.2 Simulation and Profiling

The simulator starts by loading system components and application information. It calculates the Least Common Multiple (LCM) of all the task graphs' periods and generates a set of TG objects that should be executed during the simulation. When a TG reaches its arrival time, the simulator initiates its execution by putting its first task in the ready list of the mapped resource. Resources choose the tasks from their ready list based on their local scheduling policy. After finishing execution of a task, the resource initiates a communication request, which is followed by putting the dependent tasks in the ready list of their own corresponding resources. At the end of the simulation, a list of idle intervals is generated.

6.3 Experimental Results

In this section, we present two sets of experiments. The first experiment shows the impact of PCIO on energy consumption of a three-channel software defined radio system. The second experiment shows the effect of system scale on energy consumption in presence of PCIO. We use the software defined radio system for the second experiment too. However, we change the number of activated channels, to vary number of controllable devices.

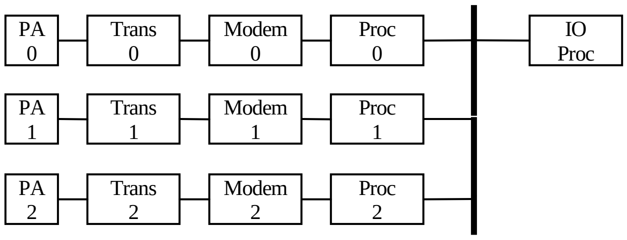


Figure 7. The block diagram of a software defined radio system

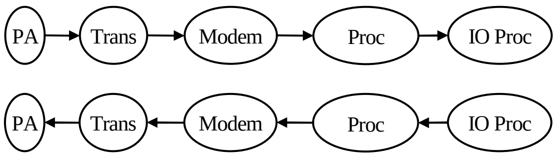


Figure 8. Send and receive task graphs for one channel

Figure 7 shows the block diagram of a CORBA-based software defined radio system [1]. The system has three communication channels for sending and receiving messages. It is also capable of generating different waveforms and can communicate with different stations simultaneously. Each channel consists of four elements (processor, modem, transceiver and Power Amplifier (PA)). The power modes of the resources are presented in Table 3. Note that the PA has different power values, which will be selected by the application based on the distance of the receiver station. Each resource has two standby modes (*stdby*1 and *stdby*2) that are controlled by power manager (Table 4). The PA has an extra standby mode (*stdby*0) that does not have any transition overhead and is controlled by the PA itself. During mode transitions, the resources are considered ON. Figure 8 shows the task graphs for sending a message to or receiving it from an external station using one of the channels. We have randomly generated 20 testbenches of 10s length. The distance between every two consecutive messages is bounded by *maxDistance* value that varies in range 200μs to 4000μs in different testbenches. In this section, the testbenches are referred using their *maxDistance* value (e.g. TB-*maxDistance*).

	Proc	Modem	Trans	PA	IO Proc
ON mode	6W	4W	25W	9, 40, 200 W	5W

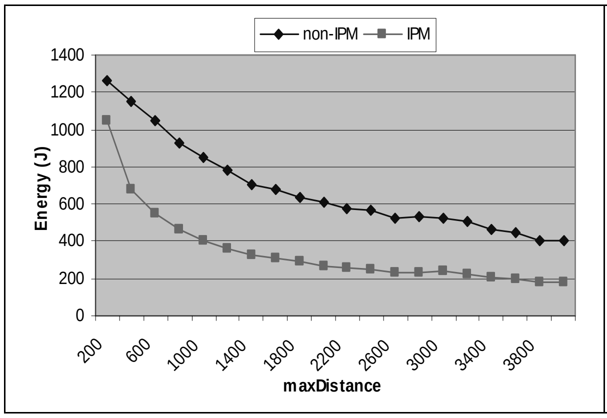
Table 3. Power consumption of different resources in their ON mode

Standby Modes	Power(W)	Entering Ov (μs)	Exiting Ov(μs)	*minST*(μs)
Stdby0 (PA only)	5	0	0	1
stdby1 (all)	1	45	45	100
stdby2 (all)	0.1	120	120	1000

Table 4. Standby modes of the resources

6.3.1 Impact of PCIO on energy consumption

In this experiment we show energy consumption of the testbenches for different values of PCIO. In general, the value of PCIO is dependent on the designed API for power commands, the number of CORBA objects that are involved in issuing power commands and the way they communicate with each other. It is also dependent on the speed of the machine that runs power manager as well as the efficiency of the utilized middleware. PCIO is usually proportional to collocation delay [7] and scheduler service overhead [8]. Collocation delay is mainly due to CORBA inter-object calls and is related to extra data processing required for CORBA compliant communications. The delay of scheduler service is due to the additional middleware code that should be executed on top of real-time scheduling of the OS and is OS-specific [5]. For a typical controlling API, like the one proposed in [10], and a Linux based CORBA platform, PCIO can be as high as a few milliseconds. In this experiment, PCIO is considered in the range of 30μs to 900μs. Figure 9 compares the energy consumption of all testbenches for a system with a non-ideal power manager (non-IPM) vs. one with an ideal power manager (IPM). The non-IPM has a PCIO of 50μs, while the IPM has a PCIO of 0. The horizontal axis shows different testbenches represented by their *maxDistance* value. In both curves, the energy gradually drops due to the increase in size of idle intervals, which provides more opportunity for power saving. However non-IPM cannot benefit from idle intervals as much as IPM does, because it can issue one power command in every 50μs, and as a result some commands will conflict with each other. The conflicting commands are removed or rescheduled using PCA algorithm.



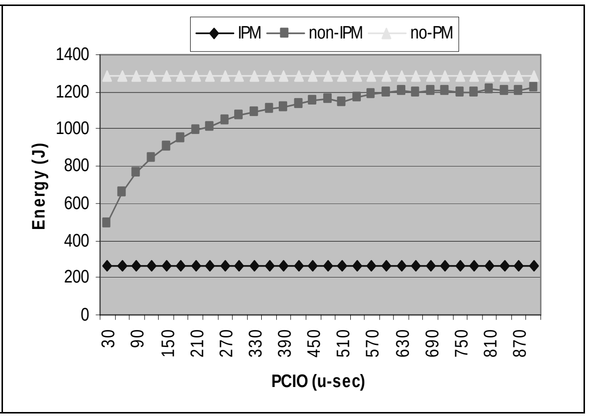


Figure 9. Comparing the energy consumption of the testbenches when PCIO=0 vs. PCIO=50μs

Figure 10. Comparing the energy consumption of TB-2000 for different values of PCIO

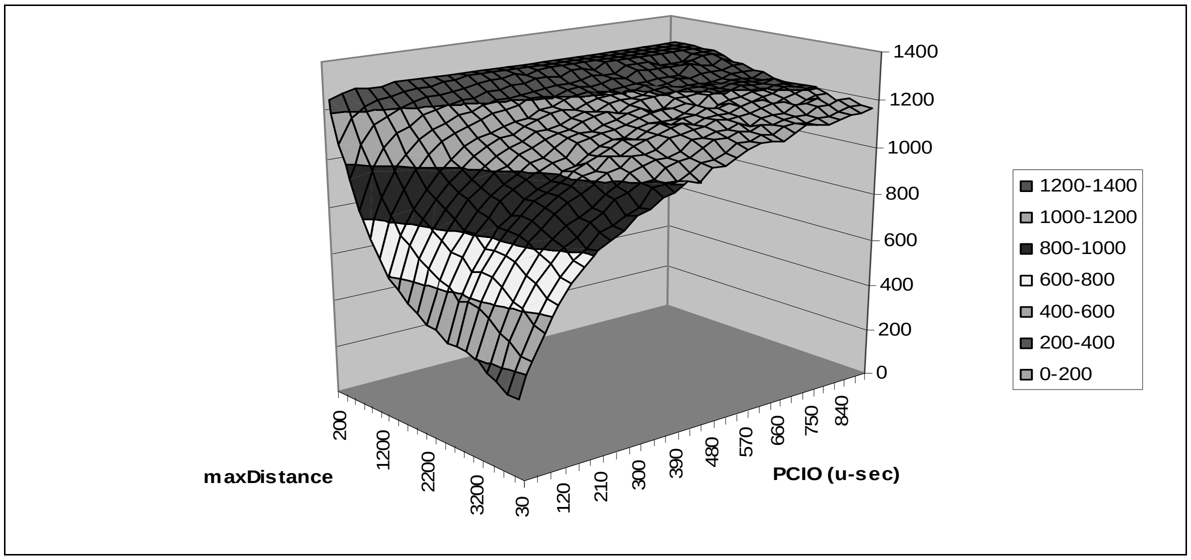


Figure 11. Comparing the energy consumption of the testbenches for different values of PCIO

Figure 10 shows the sensitivity of the result to the value of PCIO. In this case, testbench TB-2000 is simulated with different PCIOs. The amount of energy consumption rapidly goes up by increasing PCIO, because all idle intervals smaller than PCIO are discarded and the conflicting ones are resolved. Note that a PCIO as low as 30μs can increase the energy consumption by 25%. Figure 11 shows the energy consumption of all testbenches for different values of PCIO. Note that for testbenches with short idle intervals, the amount of energy consumption increases with a higher trend.

6.3.2 Impact of system scale on energy savings of non-ideal power managers

In this experiment we change the number of controllable resources in the software defined radio system by activating different numbers of channels. When only one channel is active, the power manager controls five resources. However, when all channels are active, the power manager must control 13 resources. Figure 12 shows the energy consumption of the activated channels. The results are shown for different cases: (1) ideal power management (IPM), (2) non-ideal power manager with a PCIO of 50µs (uses PCA algorithm), and (3) without any power management (no-PM). Testbench TB-2000 is selected for this experiment. Compared to ideal power manager, the system with non-ideal power manager consumes 22%, 24% and 27% more energy when one, two and three channels are activated respectively. Note when more resources are controlled the non-ideal power manager looses more energy.

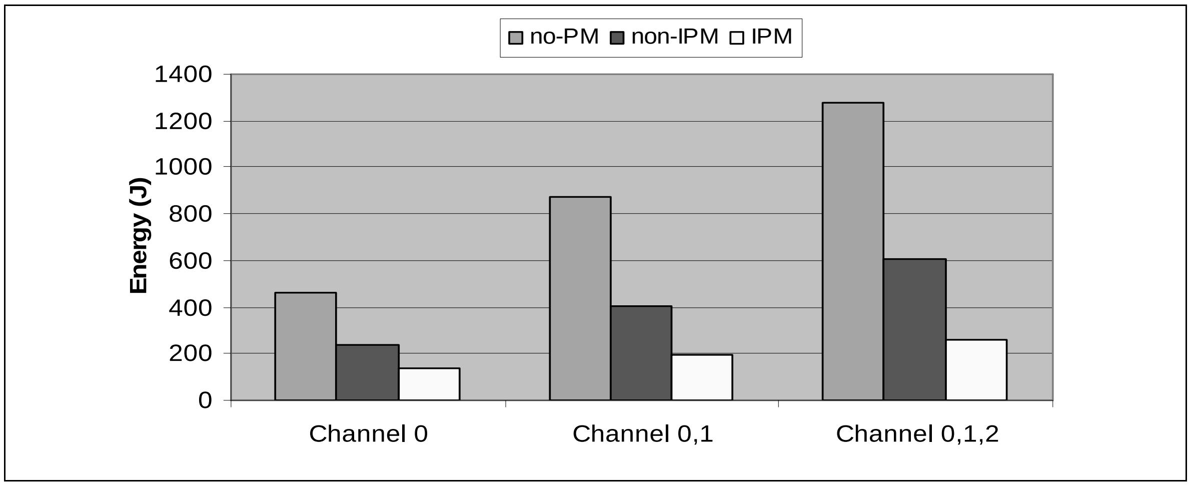


Figure 12. energy consumption of the system for different number of channels

noPM: no power manager is used, PCA: non-ideal power manager, IPM: ideal power manager

7 Conclusion

This paper introduced the Power Command Jitter problem in middleware based distributed real-time embedded systems. In such systems, the Power Command Issue Overhead (PCIO), caused by inherent middleware overhead, limits the rate of issuing power commands and therefore changes their schedule. We proposed an effective Power Command Adjustment (PCA) algorithm that re-orders, reschedules or removes the power commands so that the correctness of the schedule is maintained while minimizing the corresponding energy loss. Power command overhead can significantly affect the amount of energy saving achievable by power manager. Our experimental results on a commercial software defined radio system shows that in presence of power command issue overhead, even as low as 30µs, the power manager can loose 25% of its energy saving. Therefore, it is important to estimate the amount of PCIO and consider it in the scheduling of power commands. Also, designing and developing techniques to minimize PCIO is necessary for reducing the energy loss.

8 References

[1] Joint tactical radio system. http://jtrs.army.mil.

[2] Real-time and embedded CORBA forum. http://www.realtime- corba.com/.

[3] D. Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *IEEE Proceedings of the Third International Symposium on Distributed-Objects and Applications (DOA'01)*, 2001.

[4] C. Gill and R. Cytron. Extending real-time CORBA for next generation distributed real-time mission-critical systems. In *OMG Second Workshop on Real-Time and Embedded Systems*, 2001.

[5] D. Levine, S. Flores-Gaitan, and D. Schmidt. An empirical evaluation of OS support for real-time CORBA object request brokers. In *Real-Time Technology and Applications Symposium (RTAS '99)*, 1999.

[6] R. Noseworthy. IKE 2-implementing the stateful distributed object paradigm. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.

[7] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, V. Kachroo, and A. Gokhale. Applying optimization principle patterns to real-time ORBs. *IEEE Concurrency Magazine*, January-March 2000.

[8] D. Schmidt, D. Levine, and S. Mungee. The design and performance of real-time object request brokers. *Computer Communications*, pages 294–324, April 1998.

[9] CORBA performance and overheads. http://www.ois.com/resources/ corb-10.asp.

[10] D. Haverkamp, D. Jensen, and S. Koenck. CORBA interfaces for power management / CORBA for low power embedded devices. In *OMG Workshop on Distributed Object Computing for Real-time and Embedded Systems*, 2003.

[11] Neal K. Bambha, Shuvra S. Bhattacharyya, Juergen Teich, and Eckart Zitzler. Hybrid global/local search strategies for dynamic voltage scaling in embedded multiprocessors. In *Proc. International Workshop on Hardware/Software Codesign (CODES/CACHE)*, page 243, 2001.

[12] Y. Zang, X. Hu, and D. Chen. Task scheduling and voltage selection for energy minimization. In *Proc. DAC*, 2002.

[13] Marcus Schmitz, Bashir Al-Hashimi, and Petru Eles. Energy efficient mapping and scheduling for DVS enabled distributed embedded systems. In *Proc. Design, Automation and Test in Europe - DATE*, 2002.

[14] J. Luo and N. K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed realtime embedded systems. In *Proc. Int. Conf. Computer-Aided Design*, pages 357–364, 2000.

[15] Inki Hong, Gang Qu, Miodrag Potkonjak, and Mani B. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processor. In *IEEE Real-Time Systems Symposium (RTSS '98)*, 1998.