ICS

TECHNICAL REPORT

A Software Architecture for Building Power Aware Real Time Operating Systems

Cristiano Pereira†, Vijay Raghunathan‡, Shalabh Gupta‡,
Rajesh Gupta† and Mani Srivastava‡
† Department of Information and Computer Science, University of California, Irvine
‡ Department of Electrical Engineering, University of California, Los Angeles

E-mail: {cpereira,rgupta}@ics.uci.edu {vijay,shalabh,mani}@ee.ucla.edu

Technical Report #02-07
March 14, 2002

Information and Computer Science
University of California, Irvine

A Software Architecture for Building Power Aware Real Time Operating Systems

Cristiano Pereira†, Vijay Raghunathan‡, Shalabh Gupta‡,
Rajesh Gupta† and Mani Srivastava‡
† Department of Information and Computer Science, University of California, Irvine
‡ Department of Electrical Engineering, University of California, Los Angeles

E-mail: {cpereira,rgupta}@ics.uci.edu {vijay,shalabh,mani}@ee.ucla.edu

Technical Report #02-07
March 14, 2002

As computing moves to battery operated portable systems, the functionality is increasingly implemented in software with an embedded/real-time operating system (RTOS). For such systems, there is a need for power-aware applications and system software. In this paper, we present a layered software architecture that enables the application and OS programmers to design energy-efficient applications and RTOS services. The software architecture consists of a power-aware RTOS kernel and a set of standard software interfaces that enable easy exchange of timing and power information among the underlying hardware platform, the RTOS, and the applications. To demonstrate the utility of our approach we focus on making the task scheduling process in an RTOS power-aware, and incorporate an OS-directed dynamic power management technique that enables adaptive power-fidelity tradeoffs during task scheduling. We have implemented it using the RedHat *eCos* operating system running on a complete variable voltage system based on the Intel XScale micro-architecture. We ran four different algorithms, from a simple shutdown based scheme to a dynamic predictive and adaptive DVS algorithm. The results show an energy gain of up to 66% when comparing to the execution without any power management incorporated and 17% comparing to the simple shutdown scheme.

Contents

List of Figures

List of Tables

1 Introduction

Fueled by rapid advances in system integration and wireless communication technology, embedded systems are increasingly becoming networked. These systems often involve integration of high-performance computing and wireless communication capabilities, many of which operate under real-time constraints. The complexity of these systems together with the need for high-flexibility, and aggressive time-to-market schedules have resulted in the wide use of programmable system solutions. Software support for these systems usually takes the form of a real-time operating system (RTOS), device drivers, and runtime libraries.

Wireless embedded systems are often battery driven and, therefore, have to be designed and operated in a highly energy-efficient manner to maximize the battery lifetime. The pressing need for reduced-energy solutions has spurred the research and development of several low-power circuit design methodologies [1, 25]. While using low-power hardware circuits is necessary, it alone is not sufficient, especially since the increasing levels of system integration and clock frequencies continuously worsen the energy problem. System lifetime can only be maximized by managing the various system resources in a power-aware manner, thus empowering the system with the ability to dynamically adjust its operating point in the performance-energy-fidelity tradeoff space. To address this issue, Dynamic Power Management (DPM) techniques are being investigated (see for example, [8]). A commonly used DPM scheme is to put the idle system components in a shutdown or into a low-power state. An alternative – and more efficient when applicable – technique is Dynamic Voltage Scaling (DVS), where the voltage and operating frequency of the processor are changed dynamically during runtime to just meet the instantaneous performance requirement. Realizing the potential benefits of DVS, several commercially available mobile embedded processors (Intel StrongARM [2], Intel XScale [22], Transmeta Crusoe [3], AMD Athlon [4], *etc.*) now support dynamic clock and voltage scaling.

In this context, an RTOS provides a number of *services* to an embedded system application. It manages the creation, destruction, and scheduling of tasks, as well as the communication between tasks. It is responsible for all resource allocation and management decisions, and serves as an interface between an application and the underlying hardware platform. The RTOS has global information about the performance requirements of all the applications, and can directly control the underlying hardware, tuning it to meet specific system requirements. These characteristics make the RTOS an ideal place to implement system-level power management policies.

1.1 Paper Overview and Contributions

This paper presents an structured and layered software architecture for power-aware wireless and portable embedded systems. The architecture consists of a power-aware RTOS kernel, and a set of standard software interfaces that enable easy exchange of timing and power information between the underlying hardware platform, the RTOS, and the application. It provides a programming interface (named PASA for Power-Aware Software Architecture) that can be used to efficiently incorporate system power management policies into the RTOS. To demonstrate the impact of PASA, we focus our attention on the task scheduling process in an RTOS. We use PASA to incorporate (and compare) several DPM techniques (from a simple shutdown based scheme to an advanced DVS scheme that enables adaptive power-fidelity tradeoffs) into the RedHat *eCos* [20] operating system running on a complete variable-voltage system based on the Intel XScale micro-architecture.

1.2 Related Work

The importance of a well-structured software architecture for energy-efficiency in mobile systems is discussed in [5]. As a first step to analyze and improve the energy impact of various OS decisions, researchers have attempted to characterize the power consumption of embedded RTOSs [6, 7]. There exists a multitude of work on OS-directed dynamic power management. Shutdown based power management schemes [8] attempt to optimize the system's transition policy between several states, each of which is characterized by a performance and power consumption level.

An early work on software architecture to enable power management at OS level include BIOS-based Advanced Power Management (APM [24]). The principal limitation of APM is that the OS has no knowledge of the APM actions. Its follow on, the Advanced Configuration and Power Management (ACPI [23]) allows OS-directed power management by defining hardware registers and BIOS interfaces (table, control methods). ACPI primarily enables use of Intel-specific hardware mechanisms for power reduction. It does not provide any support for real time contraints. More importantly, it does not provide any mechanisms for the application software that can allow it to use the operating system power reduction services.

Variable voltage schemes for energy-efficient task scheduling have mostly targetted either workstation-like environments where latency is not an issue, and average throughput is the metric of performance [9, 10], or hard real-time systems, where a single timing violation may be catastrophic to system functionality [11, 12, 13, 14]. It is only recently that a new class of power-aware scheduling schemes has emerged that targets *soft-real time systems*, and permits a few deadlines to be missed [15, 16], thereby adding another degree of freedom, namely *system fidelity* or *quality of computation*, to the design and operation of these systems. This is particularly important for wireless systems where missed deadlines and packet loss over the wireless link can be treated in a similar fashion providing a level of flexibility to enable aggressive power management.

2 Software Architecture Requirements

In seeking to develop an architecture for the system and application software we briefly examine the requirements that such an architecture must satisfy. There are, of course, requirements related to the real-time nature of many embedded applications, many of which are satisfied by a range of available RTOSs. For instance, the OS should be able to monitor the real time parameters of task instances (e.g., deadlines). Further, the OS must also be able to monitor the system workload and to predict future execution times based on previous ones. The OS should also be able to manage the available hardware knobs for speed-power trade-off, adjusting them according to the system workload. This includes setting processor frequency and voltage, and set the processor into low power states by means of a simple and structured interface.

For efficient system-level power management, it is important that an application is able to monitor and control power related hardware "knobs" (such as processor voltage and frequency) as well as control and take advantadge of power aware operating system services (such as task scheduling). There is a need for mechanisms in the system software that allow efficient communication of energy, performance and accuracy tradeoffs for a given application. The electrical "knobs" must be made available to the operating system to enable the OS system writer to introduce power and energy awareness into traditional OS services.

Making an OS or runtime system aware of the system power and energy constraints is not sufficient in providing guarantees on how an application will perform in a power/energy constrained environment. The operating system services must also be available to the application programmer so that application can make use of this information in determination of power/energy dependent functionality and performance characteristics. Specifically, facilities are needed for an application to create and instantiate a task, taking into consideration the task timing parameters (period, deadline and execution time). The application should also be able to inform the operating system about the start and end of its computation and also about the expected remaining time of a given task instance (helping the OS to get a picture of the system workload).

While many dynamic power management strategies are specific to the underlying hardware and software, the application requirements for functionality and performance delivered under energy constraints can be specified independent of the platform being used. Given the diversity of hardware and software platforms used in portable, embedded and/or real-time systems, it is critical that power, energy and timing information is communicated through well-defined interfaces to ensure application portability across platforms. More concretely, to make the API level operating system independent, all system calls to the native operating system should be done by means of a POSIX interface (or any other portable operating system standard interface).

3 Software Architecture Description

We view the notion of power awareness in the application and OS as a capability that enables a continuous dialogue between the application, the OS, and the underlying hardware. This dialogue establishes the functionality and performance expectations (or even contracts, as in real-time sense) within the available energy constraints. We describe here our implementation of a specific service, namely the task scheduler, in PASA that makes it power aware. PASA is composed of two software layers and the RTOS kernel. One layer interfaces applications with operating system and the other layer makes power related hardware "knobs" available to the operating system. Both layers are connected by means of corresponding power aware operating system services as shown in Figure 1. At the topmost level, embedded applications call the API level interface functions to make use of a range of services that ultimately makes the application energy efficient in the context of its specific functionality. The API level is separated into two sub-layers. PA-API layer provides all the functions available to the applications, while the other layer provides access to operating system services and power aware modified operating system services (PA OS Services). Active entities that are not implemented within the RTOS kernel should also be implemented at this level (threads created with the sole purpose of assisting the power management of an operating system service, such as a thread responsible for killing threads whose deadlines were missed, is one example). We call this layer the power aware operating system layer (PA-OSL).

To interface the modified operating system level and the underlying hardware level, we define a power aware hardware abstraction layer (PA-HAL). The PA-HAL gives the access to the power related hardware "knobs" in a way that makes it independent of the hardware.

Table 1 lists the functions relevant to the implementation of power aware scheduling techniques. At the PA-API layer there are functions to create types (informing the real time related parameters) and instances of tasks, to notify start and end of tasks (needed by the OS in order to detect whether the task execution is over and the deadline of a task has been met), and to either inform the application

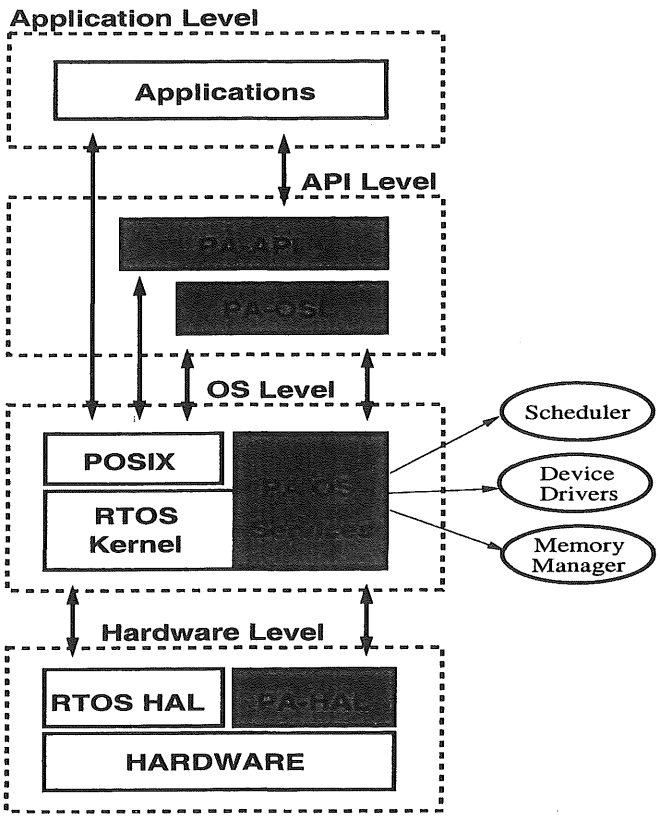


Figure 1. Power Aware Software Architecture

about the execution time predicted by the OS or tell the OS about the execution time prediction estimated by the application (which can be based on application specific parameters). At the PA-OSL layer there are functions to manipulate information related to the power aware scheduling schemes that are maintained within the kernel (such as the type table in the case of the scheduler), the thread responsible for killing threads whose deadlines were missed (assuming, of course, that the threads whose deadlines were missed are no longer useful). The alarm handler notifies the killer thread, which in turn kills the thread and re-creates it. At the PA-HAL layer functions to manipulate processor frequency and voltage levels and low power states are present. These are called by the RTOS scheduler when slowing down the processor or shutting it down. For processor frequency and voltage scaling, different platforms have different precautions that have to be taken care of before doing the scaling. These precautions might have to be done before the scaling, after it or both before and after. For these the functions `pahal_pre_set_frequency_and_voltage` and `pahal_post_set_frequency_and_voltage` are provided and must be filled by the OS programmer according to the platform.

Figure 2 shows an example on how the PA-API functions are used in the application source code in creating threads using PA-API functions. A thread is created specifying that the deadline and period are 100 and the worst case execution time is 31. The thread is instantiated and access to the power-aware functionality contracts is enabled and terminated by the functions `paapi_app_started()`

```
void main()
{
  thread_type1 =
    paapi_create_thread_type(100,31,100);

  paapi_create_thread_instance(thread_type1,
                               thread);
}
...
void thread()
{
  for (;;) {
    paapi_app_started();
    /* start of the original code */
    ...
    /* do some processing */
    ...
    /* end of the original code */
    paapi_app_done();
  }
}
```

Figure 2. Power aware source code

and `paapi_app_done()` respectively.

4 Implementation

The software architecture presented above has been incorporated in the *eCos* operating system [1]. We ported *eCos* to an Intel XScale processor based platform called 80200 Intel Evaluation Board (80200 Board for short) [21].

The XScale platform supports nine frequency levels ranging from 200MHZ to 733Mhz, even though only seven of them are used in the 80200 board due to its own limitations. The processor can also be put on three different low-power modes: IDLE, DROWSY and SLEEP. The SLEEP state is the most power saving one but requires a processor reset in order to return it to active mode. The idle state, on the other hand, is the least power saving but requires a simple external interrupt to wake the processor up.

As is the case with most RTOSs, *eCos* requires a periodic interrupt to keep track of the internal operating system tick, responsible for the timing notion within the system. In the 80200 board the only source of such interrupt is the internal XScale performance counter interrupt. In our case, this turned out to be a problem because the interrupt is internal to the processor. Therefore it cannot wake it up from one of the low power modes. Instead, we use a source of external interrupts to awaken the processor.

[1] The source code for PASA implementation can be downloaded from http://www.ics.uci.edu/~cpereira/pads

The external interrupt is connected into the interrupt pin of the processor and the interrupts are generated from an external microcontroller board.

Further, the only counter available in the 80200 board is the internal XScale performance counter, which increments one unit every processor clock cycle. Unfortunately, this counter is no longer valid when we put the processor in a low power mode because it stops. In addition, each time the clock frequency is changed, the counter has to be adjusted leading to loss of precision in mainting time. Our remedy was to associate one of the pins of the 80200 board external bus header, which enables it to generate a signal telling the external microcontroller board to get a timestamp and store its value on the host-PC. In this manner we have an accurate and precise notion of time external to the 80200 board. This is needed to keep track of time during execution of experiments. The relationship among the 80200 board, the microcontroller board and the host-PC is depicted in Figure 3 (See a picture of the hardware setup in Section E). The Maxim board shown in the lower left part is a wire-wrapped board that is directly responsible for the dynamic voltage scaling of the XScale platform.

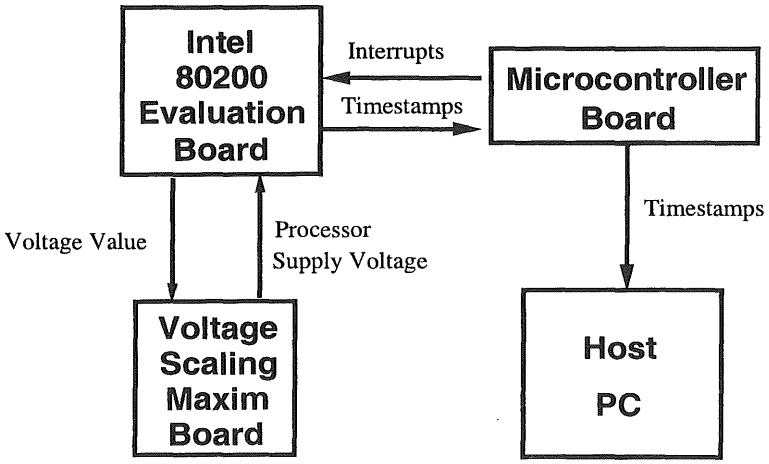


Figure 3. Hardware Architecture

4.1 Voltage Scaling

The hardware platform for implementing dynamic voltage scaling (voltage scaling at runtime) consists of a Maxim 1855 evaluation board, and an interface circuitry made using a PLD. Whenever a voltage change of the processor is required, the processor sends a byte through the peripheral bus of the 80200 board to interface circuitry which basically acts as an addressable latch. The outputs of this latch are connected to the digital inputs of the Maxim variable supply board. These inputs select the output supply voltage of the Maxim board and the processor analog and core supplies are fed from this supply voltage. For the experiments, the system was configured to run at supply voltages from 1.0V to 1.5V and frequencies from 333 MHz to 733 MHz. The frequency is changed using XScale internal registers. The supply voltage from the Maxim board can vary at steps of 0.05V.

In order to prove the utility of our software architecture, we implemented different power management algorithms using PASA. In the sequel we describe opportunities during task scheduling to both slowdown and shutdown the processor.

4.2 Power Aware Scheduling Algorithms

It has been observed in many systems that, during runtime, the processor utilization factor is often far lower than 100%, resulting in long idle intervals. This slack that inherently exists in the system can be exploited for DPM by either shutting down the processor, or through the use of processor slowdown and DVS. Note that the extent of slowdown is limited by the schedulability of the task set at the reduced speed, since excessive slowdown may cause deadline violations. A second opportunity for DPM arises due to the time-varying nature of the system workload. Performance analysis studies have shown that for typical embedded system applications (*e.g.*, audio and video encoding/decoding, encryption/decryption *etc.*), the instance to instance task execution time varies significantly and is often far lower than the worst case execution time (WCET) [18], which results in additional slack being created in the task schedule. Since task instance execution times are not known at design time, to exploit this observation, the power management policy has to be dynamic in nature.

System shutdown. The most obvious way of exploiting idle intervals in the schedule is to shutdown the processor. Modern embedded processors offer multiple low-power states, corresponding to varying degrees of system shutdown. Most shutdown based DPM policies are either predictive [19], or stochastic [8] in nature. Several techniques for DPM policy optimization have also been proposed [8].

Static voltage and frequency scheduling. Quite often, in order to reduce the complexity of implementation, power management policies are static in nature. Note that *static* in the context of such a DVS policy refers to the fact that the voltage and frequency settings of the processor are determined offline at design time. The settings can vary from task to task, which will require the processor voltage and frequency to be changed dynamically during runtime. The main advantage of a static power management policy is that it is simpler than a dynamic policy. However, the downside is that such a policy cannot exploit DPM opportunities that arise dynamically, during system operation. We have implemented a static DVS technique described in [16]. The objective of the algorithm is to determine a slowdown factor for every task such that energy consumption is minimized while still guaranteeing the schedulability of the task set.

The **response time** of a task instance is defined as the amount of time needed for the task instance to finish execution, from the instant at which it arrived in the system. The worst case response time (WCRT), as the name indicates, is the maximum possible response time that a task instance can have. For a conventional fixed speed system, the WCRT of a task under the RM scheduling scheme is given by smallest solution of the equation [26]:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \times C_j \tag{1}$$

where $hp(i)$ denotes the set of tasks with priority greater than that of task i, and $\lceil\ \rceil$ denotes the ceiling operator. The summation term on the right-hand-side of the equation represents the total interference that an instance of task i sees from higher priority tasks during its execution. If the WCRT of the task is less than its deadline D_i, the task is guaranteed to be schedulable.

Proactive DVS scheme for power-fidelity tradeoff. In [16] the authors observe that even though task instance execution times vary significantly, they depend on data values that are obtained from physical real-world signals (*e.g.*, audio or video streams), and hence are likely to have some temporal correlation between them. This temporal correlation can be exploited to proactively manage processing resources by predicting the execution times of individual task instances based on the past history of execution

times, and setting the processor speed and voltage accordingly. Several prediction models can be used, such as simple average, exponential average, least mean square, *etc.* While the use of more sophisticated models yields better prediction accuracy, it also places a higher computational burden on the scheduler, which is undesirable. As in [16], we use a simple average model that is computationally light-weight and reasonably effective.

Adaptive speed setting policy. As in any predictive policy, mispredictions do occur, which lead to task deadline misses. Wireless embedded systems invariably have some communication noise (in the form of data losses and packet errors), and are designed to be tolerant to these channel impairments. The result of a few task deadline misses is no different from noise, and only leads to a slight drop in *system fidelity*. To keep the system fidelity under specified limits, they introduce an adaptive feedback mechanism into the prediction process. The recent deadline miss history is monitored, and if the number of deadline misses is found to be increasing, the prediction is made more conservative, reducing the probability of further deadline misses. Similarly, a decreasing deadline miss history results in more aggressive prediction to reduce energy consumption. The prediction scheme thus becomes adaptive to a recent history of missed deadlines, resulting in an adaptive power-fidelity tradeoff which can be fine tuned to suit application needs.

5 Experiments and Results

We executed four different DVS algorithms using the software and hardware platforms previously described. The first is a simple "shutdown when idle" scheme. The second applies static slowdown factors and shutdown. The third applies shutdown, static and dynamic slowdow factors, the latter being based on predictions. Finally the fourth scheme combines shutdown, static and dynamic slowdown factors, and in addition a deadline miss driven adaptive factor, which keeps slowing down the processor by a factor, S, when no deadlines are being missed in the last W threads executions. This process is repeated until the total slow down factor reaches a minimum M. When THRESHOLD deadlines are missed in the previous window W of tasks executions, the processor is speeded up by factor I.

For our prototype system, we designed synthetic tasksets in which each task executes a busy loop whose execution time varies according to the best case (BCET) and worst case (WCET) execution times (see example in Section A) of the task in the same manner as described in [15]. The actual execution time of each task instance is computed from a random gaussian distribution with Mean $= (BCET + WCET)/2$ and Standard Deviation $= (WCET - BCET)/6$. The BCET is expressed as a percentage of the WCET. Four different tasksets were used for the experiments. The gaussian distribution function was extract from the Gnu Scientific library (GSL) [27] code.

All algorithms used in the experiments shutdown the processor as soon as it becomes idle. The processor is awakened when the next external interrupt arrives and then continues executing. The period of the fastest task is the same as the period of the external interrupt to ensure that at awake time, the processor is not idle. A limitation of the data collection strategy for our current testbed requires us to make all tasks with periods multiple of the fastest task. However, this is not a fundamental limitation of the software architecture or the DPM algorithms.

The shutdown mechanism is implemented using the operating system idle thread. Whenever the system becomes idle, the idle thread is scheduled. We take advantadge of this and run the code to switch the processor to IDLE state as the first action of this task. When the processor wakes up this task is

resumed and the highest rate task starts executing.

With reference to the algorithms described earlier, the slowdown factors are maintained internally to *eCos* in the form of tables. Each task type is associated with a static factor, which is computed during system initialization. The dynamic and adaptive factors are maintained in a table of task instances. The task type table also maintains a specified number (10 in our experiments) of execution times of previous tasks executions. *eCos* kernel has full access to these tables. PA-API layer accesses these tables by means of functions provided by the PA-OSL layer.

With all these information available, whenever a context switch occurs, the information about the preempted and scheduled tasks are updated and the voltage and frequency of the scheduled task are adjusted according to the static, dynamic and/or the adaptive factors based on to the algorithm running at the moment.

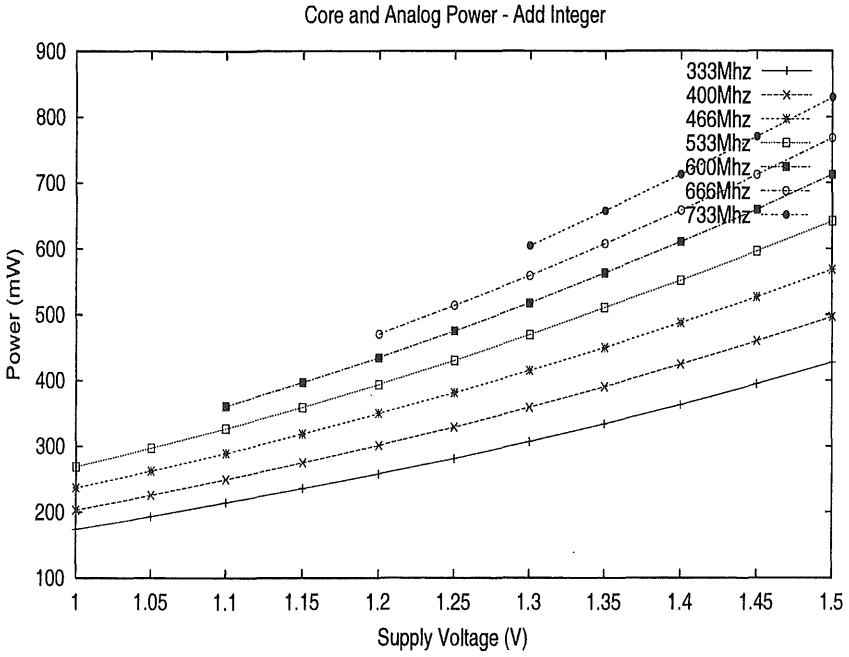


Figure 4. Core and Analog power for XScale Add-Integer instructions at different frequencies and voltages

The tasksets used in the experiments are shown in Table 2. Each task set was executed using different algorithms and ratios $BCET/WCET$, namely $0.1, 0.3, 0.5, 0.7, 0.9$. The first column shows the taskset name, the second their characteristics (Period, Worst Case Execution Time, Deadline) and the third the static slow down factor according to the Rate Monotonic Analysis criteria. This slowdown factor is multiplied into the maximum processor frequency value resulting in the frequency the task is executed. On the top of this the other factors are applied, depending on the algorithm being used. The results are shown in Figure 5 for different ratios $BCET/WCET$. These show the percentage of total energy consumption of the tasksets running four different algorithms over the power consumption of running the same tasksets without any power management strategy. A smaller ratio indicates a better power management strategy. All of the power management strategies were implemented using the PASA described earlier. We see improvements of up to 66% of energy consumption comparing with no power

management and 17% comparing to the simple shutdown scheme.

We collected traces for the various executions of each taskset and based on the traces we calculate the total power consumption. For the traces, the application is loaded into the 80200 board through the serial interface. Once its running, for each change on the processor state we send a timestamp signal to the microcontroller board, which in turn stores it in the host PC. By change in the processor state we refer to voltage/frequency change, shutdown to idle and wake up from idle. For each time we request a timestamp, we also store the ID of the event that just happened in the 80200 Board internal memory. After finishing the execution of a taskset, the entire sequence of IDs is sent into the PC host through the serial port. Having both traces available on the PC host we calculate the energy consumption, which is the power consumption of XScale Add-Integer instructions at different frequency levels (that were measured as shown in Figure 4), multiplied by the execution time at a specific frequency level. For the IDLE state the power consumption was 1 mW as per the datasheet.

Table 3 shows the results of deadlines missed for Tasksets A, B, C and D. We used a window $W = 5$ and for each taskset different values of M, S and I ($M = 0.85, S = 0.05, I = 0.1$ for taskset A; $M = 0.90$, $S = 0.05$, $I = 0.1$ for tasksets B and D; $M = 0.85$, $S = 0.01$, $I = 0.2$ for taskset C) with which we traded off deadline misses and energy savings (note that the factors used were not the ideal ones since the optimality of this factor is not on the scope of this paper). THRESHOLD value is 2. We can see a low rate of deadlines missed except for the lower priority tasks of each taskset when executing shutdown/static/dynamic/adaptive with $BCET/WCET$ ratio 0.9 for A, B, C and D, and 0.7 for D.

6 Conclusions

In this paper, we have presented a sofware architecture, PASA, that enables communication of energy related data among applications, RTOS and hardware. In order to show its utility, PASA was incorporated into *eCos* operating system running on a complete variable voltage system based on a XScale processor. We also implemented four different DVS algorithms, from a simple shutdown based scheme to a dynamic predictive and adaptive DVS algorithm and collected data to show the energy savings on a real system implemented using PASA. The results show a gain of up to 66% when compared to the execution without any power management incorporated and 17% compared to the simple shutdown scheme. For future we will add more power aware services to the RTOS and augment PASA with functions to access these services. We will also implement real power measurements with real applications running on the system.

A Synthetic Task Set Example

In this section an example of one taskset is presented. The function `cyg_user_start` creates the task types and instances. Each task executes a variable delay loop depending on the result of the gaussian distribution function. When the execution is done the functions call the eCos function to get suspended. The implementation below is not completely following the POSIX standard.

```
/*--------------------------------------------------------------------
Copyright (c) 2001-2002 The Regents of the University of California.
All Rights Reserved. Permission to use, copy, modify, and distribute
this software and its documentation for educational, research and
non-profit purposes, without fee, and without a written agreement is
hereby granted, provided that the above copyright notice, this
paragraph and the following two paragraphs appear in all copies.
This software program and documentation are copyrighted by The
Regents of the University of California ("The University of California")

THE SOFTWARE PROGRAM AND DOCUMENTATION ARE SUPPLIED "AS IS," WITHOUT
ANY ACCOMPANYING SERVICES FROM THE UNIVERSITY OF CALFORNIA.
FURTHERMORE, THE UNIVERSITY OF CALIFORNIA DOES NOT WARRANT THAT THE
OPERATION OF THE PROGRAM WILL BE UNINTERRUPTED OR ERROR-FREE. THE
END-USER UNDERSTANDS THAT THE PROGRAM WAS DEVELOPED FOR RESEARCH
PURPOSES AND IS ADVISED NOT TO RELY EXCLUSIVELY ON THE PROGRAM FOR
ANY REASON.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF
THE POSSIBILITY OF SUCH DAMAGES. THE UNIVERSITY OF CALIFORNIA
SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND
THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE,
SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
--------------------------------------------------------------------*/
/*--------------------------------------------------------------------

  Author(s): Cristiano Pereira
  Date: 1/10/2002
  Purpose: Made up task set, used to test the predictive algorithm
  Description: This is a made up task set. Each task executes a loop,
               which takes random execution time simulating in this way
               variable task execution time in the system.
```

```
-------------------------------------------------------------------------*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <cyg/kernel/kapi.h>
#include <cyg/hal/hal_io.h>

#include "procpower.h"
#include "schedpower.h"
#include "gsl_randist.h"

#include "schedpower_debug.h"

#define FRACTION 0.9

#define NUM_TASKS 3
#define STACK_SIZE 4096

cyg_thread thread_s[NUM_TASKS];
char stack[NUM_TASKS][STACK_SIZE];
cyg_thread_entry_t simple_thread;

/* killer thread handle, thread region and stack area */
cyg_thread kt_thread_s;
cyg_thread test_thread_s;
char kt_stack[STACK_SIZE];
char test_stack[STACK_SIZE];
cyg_handle_t killer_handle;
cyg_handle_t test_handle;

struct thread_type_handle_t type_handle[NUM_TASKS];
struct thread_instance_handle_t instance_handle[NUM_TASKS];

void simple_thread(cyg_addrword_t data);
void test_thread(cyg_addrword_t data);

#include "sched_var.macro"

void cyg_user_start(void)
{
  int i, j;
  struct thread_type_info_t th_type_info;
  struct thread_instance_info_t th_instance_info;
```

```
init_log();
initialize_processor_power_management();
log_event(0x9);

cyg_thread_create(0, killer_thread, (cyg_addrword_t) 0, "Killer",
                  (void *) kt_stack, 4096, &killer_handle, &kt_thread_s);

cyg_thread_resume(killer_handle);

init_power_scheduler();

/* thead type info */
th_type_info.wcet = 50;
th_type_info.deadline = 160;
th_type_info.period = 160;
th_type_info.hard = 1;
th_type_info.entry_point = simple_thread;

create_task_type(th_type_info, &type_handle[0]);

/* thead type info */
th_type_info.wcet = 100;
th_type_info.deadline = 320;
th_type_info.period = 320;
th_type_info.hard = 1;
th_type_info.entry_point = simple_thread;

create_task_type(th_type_info, &type_handle[1]);

/* thead type info */
th_type_info.wcet = 150;
th_type_info.deadline = 640;
th_type_info.period = 640;
th_type_info.hard = 1;
th_type_info.entry_point = simple_thread;

create_task_type(th_type_info, &type_handle[2]);

for ( i = 0; i < NUM_TASKS; i++ ) {
  create_task_instance(type_handle[i],
                       &instance_handle[i],
                       (cyg_addrword_t)1,
                       "Thread",
                       (void *) stack[i],
      STACK_SIZE,
```

```
                        &thread_s[i]);
    }
}

void simple_thread(cyg_addrword_t data)
{
    cyg_uint32 i = (cyg_addrword_t)data, j, k;
    cyg_uint16 instance_index = get_instance_table_index(cyg_thread_self());
    cyg_tick_count_t exec_time, current_time, start_tick;
    unsigned usec1, usec2;
    unsigned msec1, msec2;
    cyg_uint32 delay;
    gsl_rng *rng = (gsl_rng *)gsl_rng_alloc(gsl_rng_default);

    while (1) {
      // Useful thread work is within app_started and app_done function calls
      app_started(instance_index);
      start_tick = current_time = cyg_current_time();

#if defined(XSCALE_PROCESSOR) || defined(LINUX_SYNTHETIC_PROCESSOR)
      // Get execution time according to a Gaussian distribution
       delay = (unsigned)ceil(((1.0 + FRACTION)*
                               (instance_table[instance_index]->type_info->wcet))/
                               (double)2.0 + gsl_ran_gaussian(rng,
                              ((1.0 - FRACTION)*
                              (instance_table[instance_index]->type_info->wcet))/
                              (double)6.0));
       // Wait for hundreds of microseconds
       DELAY_100USEC( delay - 1 );
#endif
      app_done(instance_index);
      current_time = cyg_current_time();

      cyg_thread_suspend(cyg_thread_self());
      /* sleep until the beginning of the next thread period */
    }
}
```

B Interface with the Microcontroller board

Below the interface with the microcontroller board is presented. Every time the `log_event` is called a time stamp is requested to the microtroller board, which sends it through the serial port. Also a event identifier is written into the XScale board memory.

```
unsigned char power_log[MAX_EVENTS_LOGGED];
unsigned short log_index = 0;
unsigned char *timestamp_mem = (unsigned char *)(0x022c0000);

void time_stamp()
{
  *timestamp_mem = 0xFF;
  *timestamp_mem = 0x10;
  *timestamp_mem = 0x10;
  *timestamp_mem = 0xFF;
}

void init_log()
{
  log_index = 0;
}

void log_event(unsigned char event_type)
{
  time_stamp();
  power_log[log_index++] = event_type;
}
```

C External and Internal Counter Interrupts Handling

```
// Interrupt service routine for the internal timer
static cyg_uint32 pmu_ccnt_ovfl_ISR(cyg_vector_t vector, cyg_addrword_t data)
{
  usec_elapsed_time++;
  hal_microsecond_timer_reset(0);
  hal_interrupt_acknowledge(CYGNUM_HAL_INTERRUPT_PMU_CCNT_OVFL);
  return CYG_ISR_HANDLED;
}

void hal_microsecond_timer_initialize(unsigned period)
{
    // event types both zero; clear all 3 interrupts;
    // disable all 3 counter interrupts;
    // CCNT counts every processor cycle; reset all counters;
    // enable PMU.
    register unsigned init = 0x00000401;
    register unsigned init1 = 0x00000000;
    asm volatile (
        "mcr      p14,0,%0,c0,c0,0;" // write into PMNC
        "mcr      p13,0,%1,c8,c0,0;"
        :
        : "r"(init),"r"(init1)
        /*:*/
        );
    if ( period != 0 ) {
      orig_usec_period = period;
      period = (~period) + 1;
      usec_period = period;
    }
    hal_microsecond_timer_reset(0);
    hal_microsecond_timer_reset_count();
}

void hal_microsecond_timer_reset(unsigned new_value)
{
  if ( !new_value ) {
    asm volatile (
        "mrc      p14,0,r0,c1,c0,0;" // read from CCNT - how long since OVFL
        "mcr      p14,0,%0,c1,c0,0;" // write into CCNT
        :
        : "r"(usec_period)
        : "r0"
        );
```

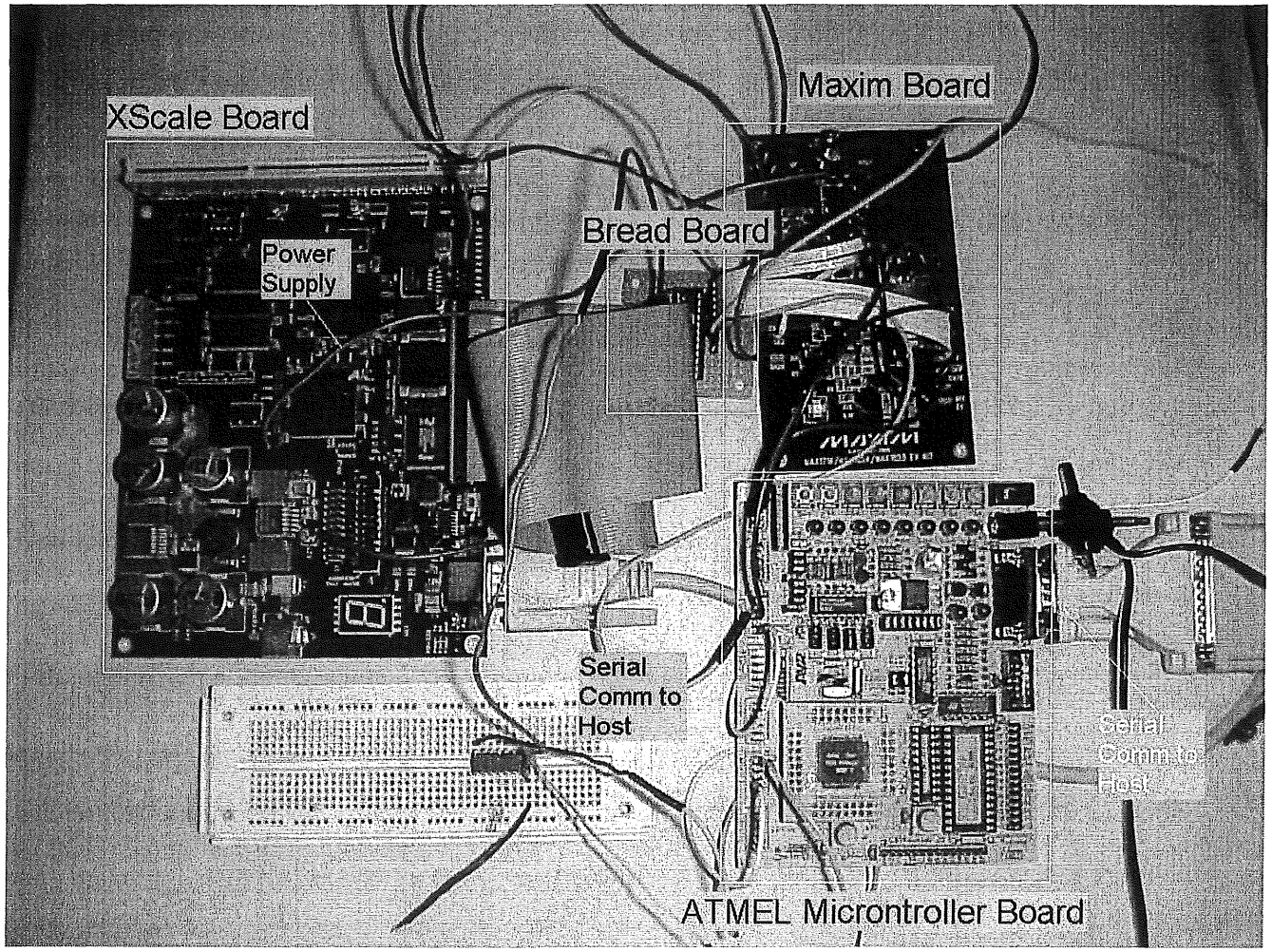
```
  }
  else {
    asm volatile (
        "mrc      p14,0,r0,c1,c0,0;" // read from CCNT - how long since OVFL
        "mcr      p14,0,%0,c1,c0,0;" // write into CCNT
        :
        : "r"(new_value)
        : "r0"
        );
  }
}

void hal_microsecond_timer_reset_count(void)
{
  usec_elapsed_time = 0;
}

static cyg_uint32 nirq_ISR(cyg_vector_t vector, cyg_addrword_t data)
{
  int isr_ret = 0;
  unsigned char old;
  unsigned char i;

  // Acknowledges the external interruption
  old = *int_ack_timestamp_mem;
  for ( i = 0; i < 40; i++ )
    *int_ack_timestamp_mem = old & 0x2F;
  *int_ack_timestamp_mem = old | 0x10;

  if ( !first_time ) {
    isr_ret = hal_call_isr (CYGNUM_HAL_INTERRUPT_TIMER);
    CYG_ASSERT (isr_ret & CYG_ISR_HANDLED, "Interrupt not handled");

    clk++;
  }
  else {
    first_time = 0;
  }
  hal_microsecond_timer_reset(0);      // reset the microsecond internal timer
  hal_microsecond_timer_reset_count(); // reset the microsecond counter

  #include "wakeup_threads.macro"
  return isr_ret;
}
```

D Modified eCos Source Tree

The modified eCos source file with the logging and power management alterations can be downloaded from `http://www.ics.uci.edu/~cpereira/pads/`.

E Picture of the Hardware Platform

Below follows a picture of the hardware setup described by Figure 3.



References

[1] A. P. Chandrakasan and R. W. Brodersen, *Low Power CMOS Digital Design.* Kluwer Academic Publishers, Norwell, MA, 1996.

[2] Intel Inc. (*http://developer.intel.com/design/strong/*)

[3] Transmeta Inc. (*http://www.transmeta.com*)

[4] AMD Inc. (*http://www.amd.com*)

[5] A. Vahdat, A. R. Lebeck, and C. S. Ellis, "Every Joule is precious: A case for revisiting Operating System design for energy efficiency" in *ACM SIGOPS European Worshop*, 2000.

[6] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Power analysis of embedded operating systems" in *Proc. ACM DAC*, June 2000.

[7] A. Acquaviva, L. Benini, and B. Ricco, "Energy characterization of embedded real-time operating systems", in *Proc. COLP*, 2001.

[8] L. Benini, A. Bogliolo, and G. De Micheli,"A survey of design techniques for system-level dynamic power management," in *IEEE Trans. on VLSI Systems*, vol. 8, iss. 3, pp. 299–316, June 2000.

[9] M. Weiser, B. Welch, A. Demers and S. Shenker, "Scheduling for reduced CPU energy" in *Proc. First Symp. on OSDI*, pp.13–23, Nov. 1994.

[10] K. Govil, E. Chan, and H. Wasserman "Comparing algorithms for dynamic speed-setting of a low-power CPU" in *Proc. MOBICOM*, Nov. 1995.

[11] T. Ishihara and H. Yasuura, "Voltage Scheduling Problem for Dynamically Variable Voltage Processors, in *Proc. ISLPED*, pp.197–202, Aug. 1998.

[12] I. Hong, M. Potkonjak and M. B. Srivastava, "On-line scheduling of hard real-time tasks on variable voltage processors" in *Proc. ICCAD*, 1998.

[13] Y. Shin, and K. Choi,"Power conscious fixed priority scheduling for hard real-time systems," in *Proc. ACM DAC*, pp. 134–139, June 1999.

[14] C. M. Krishna and Y. H. Lee, "Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems" in *Proc. RTAS*, 2000.

[15] P. Kumar and M. B. Srivastava,"Predictive strategies for low-power RTOS scheduling" in *Proc. IEEE ICCD*, September 2000.

[16] V. Raghunathan, P. Spanos, and M. B. Srivastava,"Adaptive power-fidelity in energy-aware wireless systems" in *Proc. IEEE RTSS*, December 2001.

[17] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment" in *Journal of ACM*, vol. 20, pp. 46-61, Jan. 1973.

[18] Y. -T. S. Li, and S. Malik "Performance analysis of embedded software using implicit path enumeration" in *IEEE Trans. on CAD*, vol. 16, iss. 12, pp. 1477-1487, Dec. 1997.

[19] M. B. Srivastava, A. P. Chandrakasan and R. W. Brodersen "Predictive system shutdown and other architectural techniques for energy efficient programmable computation" in *IEEE Trans. on VLSI Systems*, March 1996.

[20] RedHat. eCos: Embedded configurable operating system. http://www.redhat.com/products/ecos/, 1998.

[21] Intel. Intel 80200 processor evaluation platform board manual (80200EVB), 2001.

[22] Intel. Intel 80200 processor based on intel xscale microarchitecture developers manual, 2000.

[23] Intel, Microsoft, and Toshiba. Advanced configuration and power interface specification, 1996.

[24] Intel and Microsoft. Bios interface specification, 1996.

[25] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi. Battery-driven system design: A new frontier in low power design. In *Invited Embedded tutorial, Asia and South Pacific Design Automation Conference (ASP-DAC) / International Conference on VLSI Design*, January 2002.

[26] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment" in *Journal of ACM*, vol. 20, pp. 46-61, Jan. 1973.

[27] Redhat. GSL: The gnu scientific library, 2002.

Layer	Functions/Threads
PA-API	paapi_create_thread_type()
	paapi_create_thread_instance()
	paapi_app_started()
	paapi_get_time_prediction()
	paapi_set_time_prediction()
	paapi_app_done()
PA-OSL	paosl_create_task_type_entry()
	paosl_create_task_instance_entry()
	paosl_killer_thread()
	paosl_killer_thread_alarm_handler()
PA-HAL	pahal_initialize_processor_pm()
	pahal_get_frequency_levels_info()
	pahal_get_current_frequency()
	pahal_set_frequency_and_voltage()
	pahal_pre_set_frequency_and_voltage()
	pahal_post_set_frequency_and_voltage()
	pahal_get_lowpower_states_info()
	pahal_set_lowpower_state()

Table 1. PASA relevant functions

A	T1 = (16000, 5000, 16000)	0.85
	T2 = (32000, 10000, 32000)	0.85
	T3 = (64000, 15000, 64000)	0.85
B	T1 = (10000,1500,10000)	0.95
	T2 = (10000,3000,10000)	0.95
	T3 = (20000,9700,20000)	0.95
	T4 = (40000, 700,40000)	0.95
C	T1 = (8000, 3100, 8000)	0.85
	T2 = (16000, 2500, 16000)	0.85
	T3 = (32000, 4400, 32000)	0.85
	T4 = (88000,10000, 88000)	0.85
	T5 = (176000,8000,176000)	0.85
D	T1 = (18000, 8800,18000)	0.88
	T2 = (36000, 3500,36000)	0.88
	T3 = (72000, 6000,72000)	0.88
	T4 = (90000,13000,90000)	0.88
	T5 = (108000, 2200,108000)	0.88
	T6 = (180000, 1000,180000)	0.75

Table 2. Tasksets used in the experiments

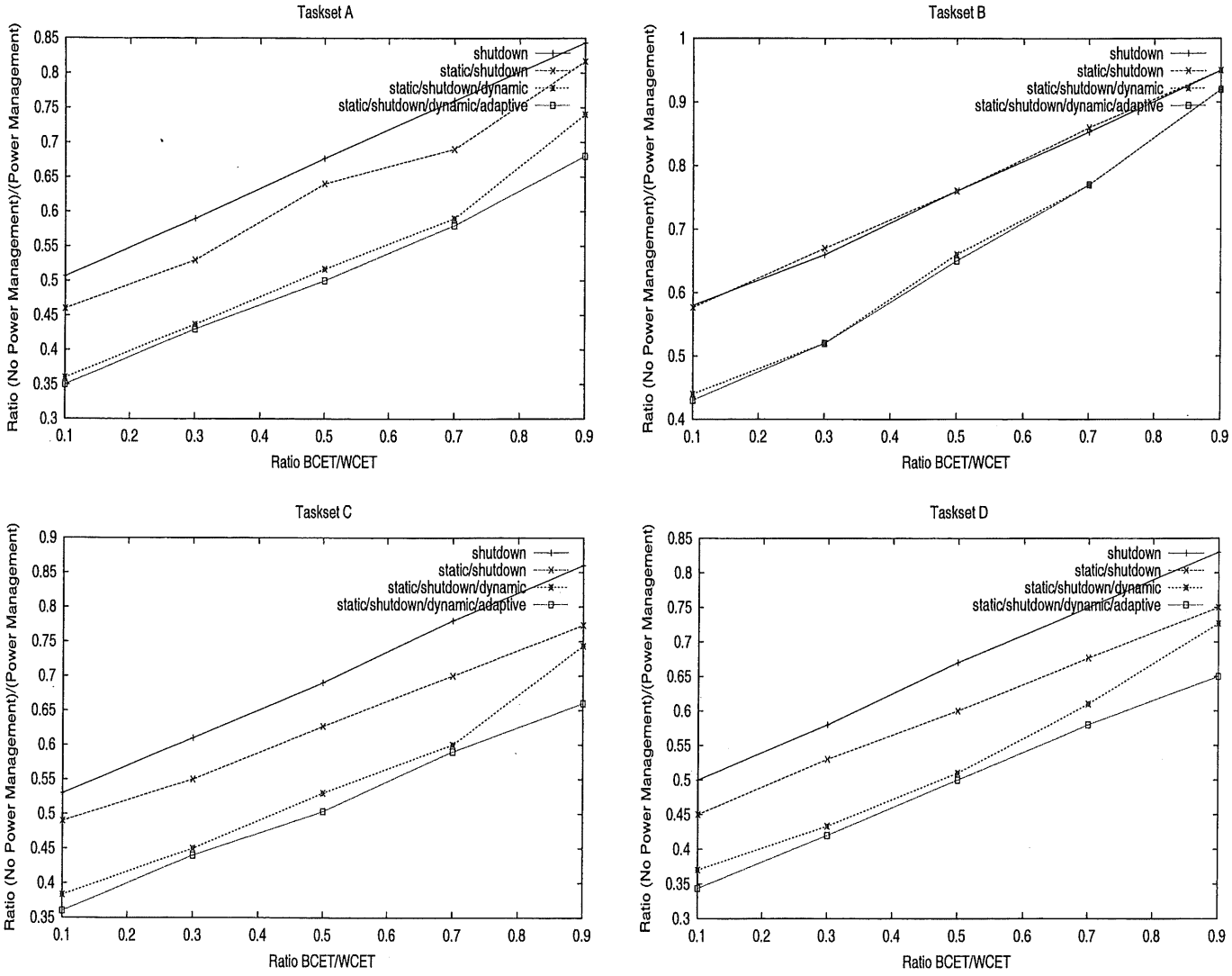


Figure 5. Ratio between (No power management)/(Power management) for each task set executing different algorithms at different ratios BCET/WCET

A	0.1	0.3	0.5	0.7	0.9	B	0.1	0.3	0.5	0.7	0.9
T1	0	1	1	1	2	T1	0	0	0	0	0
T2	0	0	0	0	2	T2	0	0	0	0	1
T3	0	0	0	3	25	T3	0	2	2	2	2
						T4	0	2	0	0	0
C	0.1	0.3	0.5	0.7	0.9	D	0.1	0.3	0.5	0.7	0.9
T1	1	2	1	1	1	T1	2	2	0	0	0
T2	0	0	0	0	0	T2	2	2	0	0	0
T3	0	0	0	0	4	T3	0	0	0	0	0
T4	0	0	0	0	10	T4	0	0	0	2	5
T5	0	0	0	0	25	T5	0	0	0	2	21
						T6	0	0	0	0	20

Table 3. Percentage of deadlines missed per algorithm for tasksets A, B, C and D using shutdown/static/dynamic/adaptive algorihm