

INCORPORATING POWER AWARE AND FAULT TOLERANCE TECHNIQUES IN  
REAL TIME OS

A Thesis Presented

by

AKASH GOEL

Submitted to the Graduate School of the  
University of Massachusetts Amherst in partial fulfillment  
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

April 2005

Electrical and Computer Engineering

© Copyright by Akash Goel 2005

All Rights Reserved

INCORPORATING POWER AWARE AND FAULT TOLERANCE TECHNIQUES IN  
REAL TIME OS

A Thesis Presented

by

AKASH GOEL

Approved as to style and content by:

---

C. Mani Krishna, Chair

---

Israel Koren, Member

---

Csaba Andras Moritz, Member

---

Seshu B. Desu, Department Head  
Electrical and Computer Engineering

To my parents

## ACKNOWLEDGMENTS

I would like to express my deep appreciation to Prof. C. Mani Krishna and Prof. I. Koren for their constant encouragement, support and guidance during the course of my MS. I also wish to thank Prof. Krishna for his patience in answering my numerous question. I would also like to thank Prof. C. A. Moritz for reviewing my thesis and for sharing valuable insights. I would like to express my gratitude to NSF for the supporting this work. I am grateful to Vijay and Kamesh for helping me out at many occasions. I would also like to thank my friends at Umass for making my stay wonderful. Lastly, I would like to thank my parents and Debyani for their love and support throughout my stay here.

ABSTRACT

INCORPORATING POWER AWARE AND FAULT TOLERANCE TECHNIQUES IN  
REAL TIME OS

APRIL 2005

AKASH GOEL

INSTITUTE OF TECHNOLOGY, BANARAS HINDU UNIVERSITY, INDIA

M.S.E.C.E, UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor C. Mani Krishna

Embedded systems often have severe power and energy constraints. Dynamic voltage scaling is a mechanism by which energy consumption may be reduced. DVS is based on the fact that cpu's power is quadratically related to voltage, while its speed is linearly related. In this work we implement a dynamic voltage scaling based scheduler in eCos operating system running on Voltage Scalable Intel Xscale Board and show how energy usage can be reduced while still meeting hard real-time deadlines.

Embedded systems are often subjected to faults due to harsh environmental conditions. We describe a power aware fault tolerant scheduling algorithm for real time system and simulate it to demonstrate that reliability of a system can be traded for the power consumption.

Checkpointing is a mechanism for providing fault tolerance to the system. In this approach, at each checkpoint, the system state is saved and when a fault is detected, the system is restarted from the most recent checkpoint. We have implemented a checkpointing library for real-time applications in Rtlinux.

## TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS . . . . .	v
ABSTRACT . . . . .	vi
LIST OF TABLES . . . . .	ix
LIST OF FIGURES . . . . .	x
CHAPTER	
1. INTRODUCTION . . . . .	1
1.1 Overview . . . . .	1
1.2 Contribution . . . . .	3
1.3 Outline of Thesis . . . . .	3
2. TECHNICAL BACKGROUND . . . . .	5
2.1 Dynamic Voltage Scaling . . . . .	5
2.2 Fault Tolerance . . . . .	7
2.3 Literature Survey . . . . .	8
3. VOLTAGE SCALING ALGORITHM . . . . .	10
3.1 Models and Terminology . . . . .	10
3.1.1 Task Model . . . . .	10
3.1.2 Power Model . . . . .	11
3.2 Cyclic Scheduling . . . . .	11
3.3 Algorithm . . . . .	12
3.3.1 Overheads . . . . .	13
3.4 Implementation Details . . . . .	14
3.4.1 Hardware Platform . . . . .	14
3.4.2 Software Architecture . . . . .	15
3.4.3 Implementation of EDF in eCos . . . . .	19
3.4.4 Voltage Scaling algorithm in eCos . . . . .	19
3.4.5 Results . . . . .	20

4. FAULT-TOLERANT VOLTAGE SCHEDULING ALGORITHM . . .	23
4.1 Model and Terminology . . . . .	24
4.1.1 Fault and Recovery Model . . . . .	24
4.2 Algorithm . . . . .	25
4.3 Results . . . . .	25
5. IMPLEMENTATION OF CHECKPOINTING IN RTOS . . . . .	29
5.1 Introduction . . . . .	29
5.2 Usage . . . . .	31
5.3 Implementation details . . . . .	31
5.3.1 Program Counter Adjustment . . . . .	32
5.3.2 Stack Segment . . . . .	32
5.3.3 Registers . . . . .	33
5.4 Transfer of Checkpointed State . . . . .	34
5.5 Recovery process . . . . .	34
5.6 Experimental results and analysis . . . . .	35
5.7 Limitations . . . . .	36
6. CONCLUSIONS AND FUTURE DIRECTIONS . . . . .	37
REFERENCES . . . . .	39
A. AN IMPLEMENTATION OF FOUR TASKS' TASKSET WITH DYNAMIC VOLT- AGE SCALING THREAD IN eCOS . . . . .	42
B. IMPLEMENTATION OF VOLTAGE-CLOCK SCALING SOFTWARE . . . . .	53

## LIST OF TABLES

Table		Page
3.1	Voltage-Speed Settings . . . . .	15
4.1	Voltage Levels . . . . .	26
4.2	System parameters . . . . .	26

## LIST OF FIGURES

Figure		Page
3.1	Circuit Diagram. . . . .	14
3.2	Impact of $a$ on the CPU energy consumption. . . . .	21
3.3	Impact of the number of tasks on the CPU energy consumption. . . . .	22
3.4	Impact of number of tasks on the CPU energy consumption. . . . .	22
4.1	Impact of inflation factor on failure probability and CPU's energy consumption for worst case target utilization of 0.6 . . . . .	27
4.2	Impact of inflation factor on failure probability and CPU's energy consumption for worst case target utilization of 0.7 . . . . .	27
4.3	Impact of checkpointing overheads on failure probability . . . . .	28
4.4	Impact of failure probability on CPU energy consumption. . . . .	28

## C H A P T E R 1

### INTRODUCTION

#### 1.1 Overview

An embedded system can be defined as a system that is embedded within a given application and works entirely in the context of that application. Virtually all appliances that have a digital interface – watches, microwaves, VCRs, cars – use embedded systems. Some embedded systems include an operating system, but some are so small and specialized that the entire logic can be implemented as a single program.

Embedded systems are often battery-driven systems that have stringent power and/or energy constraints. Examples include a network of microsensors deployed in a battlefield for surveillance or a space system relying on solar and battery power. Since the application will fail if their embedded system fails, fault tolerance is often important. In addition to all this, embedded systems often run real-time applications that require meeting critical task deadlines. So these systems generally have the following key design constraints:

1. Performance
2. Power consumption
3. Fault tolerance

An embedded system that has high performance (and/or fault-tolerance) requirements will obviously has higher power consumption and vice versa. In recent years the tradeoff between power consumption and performance has gained much

attention and many sophisticated power management schemes have been proposed, aiming to reduce overall energy consumption, while still achieving good performance. Power can be managed by using various hardware, firmware, and system software facilities in an energy-efficient manner while continuing to meet the computational needs of the user. Power consumption can be reduced by shutting down idle units or by putting the processor into idle/sleep mode when there is no work. In principle, this is very simple to implement but do not account for the fact that any device which is currently idle may be required at any instant in the future. Due to the often substantial overheads associated in bringing the idle units back online, these schemes not always appropriate for real-time systems. Another alternative is to run a system under variable speed and voltage during its operation. This scheme, because of its lower overheads, gives much better performance. The main focus of this thesis is to implement and study such a system.

The majority of modern-day microprocessors use digital CMOS circuits, and their energy consumption is proportional to the square of the core supply voltage while the delay is linearly proportional to the supply voltage. In other words, if the core voltage is reduced to half, the energy consumed by the processor will reduce to one fourth of its original value but the cpu speed (or delay) will be reduced to only half. The process of actively adjusting the cpu's core voltage and speed in response to the fluctuations in its utilization is known as Dynamic Voltage Scaling (DVS).

Dynamic Voltage Scaling has emerged as a powerful mechanism for reducing the energy consumption during system operation [1]. Many current embedded processors such as the Intel Xscale [15], and Transmeta Crusoe [16] allow their operating voltage to be changed dynamically.

Fault tolerance can be added to a system in different ways. The traditional approach is:- to run the same application on another set of hardware. This is very simple approach but requires double resources and obviously is twice as power

consuming as running only one copy. Another approach is the checkpointing-based approach. In this approach, at each checkpoint, the system state is saved and when a fault is detected, the system is restarted from the most recent checkpoint. There is a cost associated with checkpointing, which includes saving the process state to a secure device. Without checkpointing, a fault will require restarting an affected task from the beginning. Checkpointing therefore has the potential to reduce execution time. In this thesis, we look at the implementation of checkpointing in detail.

## 1.2 Contribution

This thesis describes the implementation details of a DVS based power aware Real-time operating systems (RTOS). This thesis also presents the implementation of checkpointing based fault tolerance in RTOS. The fault tolerance provided in a system can also be traded for power consumption and we look at this tradeoff in detail.

The main contributions made in this thesis can be summarized as follows:

1. Implementation of a low power real-time operating system. This includes:
  - Implementation of a dynamic voltage scaling based scheduler in an RTOS
  - A proof-of-concept hardware prototype using COTS components to demonstrate the feasibility of these techniques.
2. Implementation of checkpointing libraries in an RTOS
3. Simulate a low power fault tolerant scheduling algorithm with dynamic resource reclamation using DVS.

## 1.3 Outline of Thesis

The rest of this thesis is organized in the following manner. Section 2 provides technical background on power aware computing, dynamic voltage scaling and fault

tolerance along with a literature survey. Section 3 describes a voltage scaling algorithm, its implementation in an RTOS and results. Section 3 also provides a brief description of the hardware prototype that we have used for our energy-aware kernel related experiments. Section 4 describes a power aware fault tolerant scheduling algorithm and the simulation results of this algorithm. Section 5 provides the details of an implementation of checkpointing in an RTOS. This work concludes with a brief discussion in Section 6.

## C H A P T E R 2

### TECHNICAL BACKGROUND

Operating systems (OS) manage all the processes in a computer system and can thus play an important role in reducing the processor's energy consumption. It is the responsibility of the scheduler present inside the OS to schedule different tasks in such a way that all the real-time constraints are satisfied. Since tasks are associated with resources having specific energy models, the scheduler can exploit this information to reduce run-time energy consumption. For example, real-time tasks normally do not run to their worst-case execution time, and this time can be reclaimed during the execution. The main aim of this thesis is to implement a scheduler that can exploit run-time information to reduce the overall power consumption of a system.

The rest of this section is organized as follows. In the next subsection, we present the concept behind the *Dynamic Voltage Scaling* (DVS). We also present a subsection on fault tolerance followed by a relevant literature survey.

#### 2.1 Dynamic Voltage Scaling

Most modern day microprocessors use digital CMOS circuits, and their power consumption can be modeled accurately with simple equations. CMOS circuits have both dynamic and static power consumption. Dynamic power is consumed due to the switching of gates and is still responsible for a large percentage of the total power dissipated in CMOS circuits, although static power consumption, which is due to the leakage current between the power supply and ground, is expected to increase in the future. The dynamic power consumption of a CMOS circuit can be represented as [1]:

$$P_{cmos} = C_L N_{SW} V_{DD}^2 f \quad (2.1)$$

where

$C_L$  is the circuit output load capacitance.

$N_{SW}$  is the number of switches per clock cycle.

$f$  is the clock frequency.

The circuit delay, denoted by  $\delta$ , obeys the following equation [16]

$$\delta = \frac{C_L V_{DD}}{K(V_{DD} - V_T)^\alpha} \quad (2.2)$$

where

$V_{DD}$  is the CPU core supply voltage

$K$  is a constant depending on the process and gate size

$V_T$  is the threshold voltage

$\alpha$  varies between 1 and 2;  $\alpha = 2$  for long-channel devices which have no velocity saturation.

Equation 2.1 shows that clock frequency reduction linearly decreases the power consumption and voltage reduction reduces it quadratically. Equation 2.2 shows that  $\delta$  is inversely proportional to  $V_{DD}$  (for devices with  $\alpha=2$ ) so decreasing the supply voltage increases the circuit delay. Due to the increased delay, the circuits could not be driven at the same frequency  $f$  when the supply voltage  $V_{DD}$  is decreased. Clearly there is a fundamental tradeoff between operating speed (or performance) and supply voltage (or power consumption).

For hard real-time systems, where meeting deadlines is very critical, this tradeoff is particularly useful. Real-time tasks are normally characterized by their worst case execution time which can be bounded with reasonable certainty with the help of statistical analysis and profiling. These tasks are normally periodic with their

period being known in advance. Such a priori knowledge about the workload makes power management techniques even more significant for these systems.

Modern day processors such as the Intel Xscale, Transmeta Crusoe allow their operating speed to be adjusted on the fly. These processors do not come equipped with voltage scaling capabilities and must be augmented with hardware blocks to dynamically change the supply voltage, see Figure 1. These hardware blocks should also have software interfaces that can be provided to the OS to facilitate energy aware resource management. In the next chapter, we demonstrate our implementation of DVS in an off-the-shelf processor.

## 2.2 Fault Tolerance

Two techniques for providing fault tolerance in real time systems are based on checkpointing [15] and primary and ghost copies [13].

In the checkpointing-based approach, at each checkpoint, the system state is saved and when a fault is detected, the system is restarted from the most recent checkpoint. There is a cost associated with checkpointing, which includes saving the process state to a secure place. Without checkpointing, even a brief transient fault may require us to restart the task from the beginning. Checkpointing therefore has the potential to reduce execution time.

In the primary and ghost copies based approach, ghost copies are embedded in the schedule along with the primary copies. A ghost is a backup copy, which lies dormant until it is activated to take the place of a corresponding primary whose processor has failed. In case of failure, the whole application is started from the beginning.

Both the above-mentioned techniques for providing fault tolerance require some slack to be reserved in the schedule to meet deadlines in the event of faults striking the system. However, in the common case of no faults hitting the system the scheduler can reclaim this slack by running the CPU at a lower speed.

### 2.3 Literature Survey

DVS has recently attracted a large body of research [1-4]. The main motivation behind this increasing attention is the limitation on battery life in embedded systems. Because of the increasing popularity of DVS, many embedded processors such as the Intel Xscale [15], and Transmeta Crusoe [16] are now equipped with the ability to dynamically scale their operating voltage. A study of periodic real-time tasks, which can consume energy at possibly varying rates, is presented in [21]. A benchmark suite and simulation environment for voltage scaling is presented in [22]. Dynamic voltage-scheduling algorithms for fixed-priority task systems were presented in [30]. Compiler-assisted DVS is studied in [44]. A number of low-power system-level design techniques have been developed. Idle functions units can be powered down [24] and predictive-shutdown-based power optimization techniques are presented in [25]. These methods are designed for systems with a fixed voltage supply. An alternative approach to save energy is based on dynamic power management (DPM), in which the operating system (OS) is responsible for reducing system power consumption. DPM was made possible largely due to the Advanced Configuration and Power Interface (ACPI) [26]. ACPI allows hardware power states to be controlled by the OS through system calls, effectively transferring the power reduction responsibility from the hardware (BIOS) to the software (OS). Most of the DVS research is based on simulations and not much implementation work has been reported. More recently, a speculative scheme has been proposed in [27]. This scheme has been implemented in the eCos operating system [28] running on a Intel Xscale-based board. This scheme is speculative; it allows processor speed to be reduced in anticipation of reduced execution time, and does not guarantee that deadlines will be satisfied. In [30], Earliest Deadline First (EDF) and Rate Monotonic (RM) based DVS algorithms have been developed and have been implemented

on the Linux operating system running on an AMD Athlon based notebook. These techniques are implemented as Linux kernel modules, and do not guarantee real-time behavior. A DVS algorithm based on EDF algorithm is also implemented in RTlinux [42]. This work assumes that all tasks have the same period, are non-preemptible, and all tasks run to their worst case execution time.

There has been a lot of research reported on checkpointing. Checkpointing libraries such as Libckpt, Condor, and checkpointing implemented as a kernel module such as epckpt and crak [9, 10, 11, 12, 13] run on several versions of Unix. Many researchers [5,6,7,8] also address checkpointing in a real time system. The actual implementation of checkpointing in real time operating systems has not yet been reported and no publicly available real time operating system supports any checkpointing mechanism, to our knowledge. Our main aim was to develop a checkpointing scheme in a widely accepted and freely available real time operating system.

There is also very little work reported on the power and energy aspects of fault tolerance. In [18], Unsal et al., proposed an energy-aware application-level fault tolerance scheme, where, secondary copies are used for fault tolerance in distributed real time systems. In another attempt, an energy-aware checkpointing scheme for embedded systems is presented [19]. This work is based on the assumption that at most  $k$  faults occur in the system; it doesn't consider probabilistic failure models. The affects of dynamic resource reclamation and slack reclamation on reliability and energy have also not been addressed; these can lead to significant power savings. In our work we address all these issues simultaneously.

## C H A P T E R 3

### VOLTAGE SCALING ALGORITHM

Real-time tasks normally do not run to their worst case execution times and thus leave some slack in their execution during run time. Running the CPU at a lower speed can reclaim this slack. In this chapter, we describe the implementation of an algorithm [1], which allows dynamic resource reclamation while guaranteeing that no deadlines will be missed.

#### 3.1 Models and Terminology

##### 3.1.1 Task Model

A typical real time task model consists of periodic real time tasks where each task is modeled by:

$T_i$  : Period of a task  $i$ .

$D_i$  : Relative deadline of a task  $i$  ( $=T_i$ ).

$WCET_i$  : Worst case execution time of a task  $i$  under fault free conditions.

$AET_i$  : Actual execution time of a task  $i$  under fault free conditions.

All the tasks are independent i.e. no task depends on the output of any other tasks. Also, tasks must run to completion in order to get any benefit from them i.e. they are not Increased Reward for Increased Service (IRIS) tasks [14].

$AET_i$  is assumed to vary uniformly in the range  $[a \times WCET_i, WCET_i]$ ,  $0 < a \leq 1$ . As  $a$  increases, the task's execution time becomes more deterministic.

### 3.1.2 Power Model

The processor can operate at two voltage levels:  $v_H$  and  $v_L$  ( $v_H > v_L$ ).  $v_H$  and  $v_L$  are voltages at which the processor can do useful computation. The slowdown factor by which the processor is slower at voltage  $v$  relative to when at voltage  $v_H$  is:

$$\text{slowdown}(v) = \frac{v(v_H - v_T)^\alpha}{v_H} \quad (3.1)$$

where  $v_T$  is the threshold voltage and  $\alpha$  varies between 1 and 2. We assumed  $\alpha = 1.3$  for all our experiments.

We define one unit of execution as the computation performed by the processor at  $v_H$  in unit time. Hence, one unit of execution will take  $\text{slowdown}(v)$  units of time at voltage  $v$ . The ratio of energy consumed per cycle by a processor at voltage  $v$  relative to that at voltage  $v_H$  is

$$\text{Energy\_ratio\_per\_cycle}(v) = \left(\frac{v}{v_H}\right)^\alpha \quad (3.2)$$

We have taken into account the cost associated with voltage scaling in our implementation. We find an upper bound on these overheads in section 3.3 and accounted for this cost by merging it into the worst-case profile information of tasks.

## 3.2 Cyclic Scheduling

For each periodic frame, which is the Least Common Multiple (LCM) of the periods of all the tasks in the system, all tasks are released at the beginning of the frame and must finish by the end of the current frame. We assume that the tasks have a predefined execution order. Key to the algorithm is a function,  $\Psi(t)$ , which has the property that if the amount of unfinished work at time  $t$  is not greater than  $\Psi(t)$ , there is enough processing capacity to meet all deadlines under the scheduling algorithm is being used. In other words,  $\Psi(t)$  is a sufficient condition for schedulability.

### 3.3 Algorithm

The algorithm consists of two phases. In the offline phase, which is executed before the system is actually used, we obtain the function  $\Psi(t)$  over the duration of the frame. In the online phase, when the system is operating, we set the voltage level to guarantee that the unfinished work will never exceed  $\Psi(t)$  at time  $t$ . This algorithm always sets the processor voltage to  $v_L$  when the unfinished work is guaranteed to be below  $\Psi(t)$ , and  $v_H$  otherwise. The offline phase is carried out in the following way. Let  $u$  be the worst-case utilization of the processor at high voltage (i.e., the utilization if every task ran to its WCET and was run at high voltage). Construct, for each task  $T_i$  a corresponding inflated task  $T'_i$  with the same period as  $T_i$  but whose execution time is deterministic and is equal to  $\frac{1}{u}$  times the WCET of  $T_i$ . The set of tasks,  $T' = \{ T'_1, T'_2, T'_3, \dots, T'_n \}$ , has utilization 1 by construction. Now, use the EDF scheduling algorithm to schedule it.  $\Psi(t)$  is the unfinished work, according to this schedule, of tasks currently awaiting service. Define by the `offline_task(t)` the task (if any) running at time  $t$  under the offline schedule.

The online phase is equally easy to describe. EDF is used here as well. When a task starts executing (or restarts following a preemption), the worst case unfinished work is checked and compared with  $\Psi(t)$ . If at  $\Psi(t)$ , the task is run at  $v_H$  until it is completed or preempted by another task. If below  $\Psi(t)$ , we determine the next time  $\tau$  when the worst-case unfinished work will equal  $\tau(t)$ . The CPU is run at  $v_L$  until either  $\tau$  is reached or the task has completed or been preempted. When a task is completed or preempted, we recalculate the worst-case unfinished work in the system. If the task is preempted, then calculating the unfinished work is straightforward. The amount of work done since the last time the task was resumed or executed is subtracted from the unfinished work in the system to get the current unfinished work in the system. But if the task is over, the worst-case unfinished

work is calculated depending upon the following conditions: If the deadline of the next task is greater than the deadline of this task (just finished) then just subtract the actual execution time (remaining) of the task from the unfinished work to get the current unfinished work in the system. If the deadline of the next task is less than the deadline of the task just finished, then generate another pseudo task in the system with the same period as that of the task just finished (say  $T_i$ ), execution time =  $WCET_i - ACET_i$ , and the same deadline as that of  $T_i$ . This pseudo task is like any other task, except that the cpu is idle while executing it. Let  $\text{online\_task}(t)$  denote the task that is running under the online schedule at time  $t$ . At any time,  $t$ , the processor will be running at low voltage except when any of the following conditions is satisfied:

1.  $\text{offline\_task}(t) = \text{online\_task}(t)$ .
  2. The worst-case unfinished work under the online algorithm is equal to  $\Psi(t)$ .
- If any of these conditions are satisfied, the processor runs at high voltage. The above-mentioned algorithm guarantees that no task will miss its deadline.

### 3.3.1 Overheads

Voltage-clock scaling decision is taken at the following points:

1. When a task is preempted or completed.
2. When the offline curve intersects the online curve.

The overhead associated with this algorithm can be bounded by the fact that we know the task arrival epochs. Allowing for the fact that a scaling action may have to be taken at the very beginning of the execution or at the intersection point, this bound is given by

$$\Omega = n(\omega_{DVS} + \omega_{intersect})$$

where the following notation is used:

$n$	Number of intersection points and task completions plus 1
$\omega_{DVS}$	Overhead of DVS action
$\omega_{intersect}$	Overhead of calculating intersection point

### 3.4 Implementation Details

#### 3.4.1 Hardware Platform

We implemented our prototype system on ADI Engineering’s Intel Xscale BRH board. The Xscale platform supports nine frequency levels ranging from 200 Mhz to 733 Mhz [17]. CLK, the input reference clock for Intel Xscale processor, accepts an input clock frequency of 33 to 66 Mhz. Processor uses an internal PLL to lock to the input clock and multiplies the frequency by a variable multiplier (changable through software) to produce a high-speed core clock (CCLK). This gives software the ability to change the frequency without having to reset the core. Intel Xscale also supports low voltage operation with a core supply as low as 0.95 V. At each voltage level, there is a maximum clock frequency (CCLK), though core can operate at lower frequencies if desired. Intel Xscale CPU’s core on this board is powered from 1.5 V, created by a switching regulator (LT1767). However, this can be changed by varying a single resistance in a range from 1.2V to 1.8V. By using ADI engineering’s BRH board we were able to get frequency-voltage scalable platform with minimal changes in the circuit.

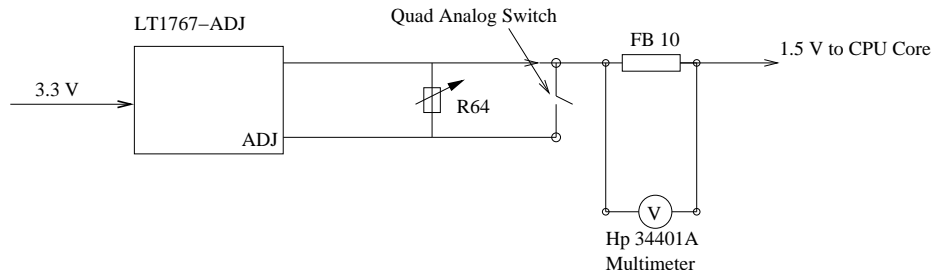


Figure 3.1. Circuit Diagram.

Voltage	CPU freq (Mhz)
1.2	400
1.5	733

Table 3.1. Voltage-Speed Settings

If feedback resistance (R64) is short circuited, 1.5 V drops to 1.2 V. So in principle, if we place a switch across this resistance, upon switching it ON we will get a supply voltage of 1.2 V and switching it OFF will produce 1.5 V. As shown in figure 3.1, the power regulator circuit of the board is modified by placing an MM74HC4066 (quad analog switch) across the feedback resistance of power regulator. This switch is controlled through the COM2 port, by connecting the RTS line of the serial port(COM2) to the control pin (pin 13) of this switch. The RTS line is either at +5 V or at -5 V. We write a value to RTS line such that the switch is in the ON state when 1.2 Volts is required, and OFF otherwise. We use a diode and 10 Kohm resistor to clip the voltage to 0 V if the input goes to -5 V. Using this circuit, a total of 2 voltage settings 1.2 V and 1.5 V are made available to the kernel.

Voltage-clock scaling overheads include the phase-locked loop (PLL) synchronization time of 2000 cpu cycles and, based on [33], approximately 40  $\mu$ seconds of voltage stabilization time. 40  $\mu$ seconds is the time taken by voltage regulator to produce a stable voltage and during this time period the processor continues to run, albeit at a lower speed.

The speed voltage combinations that we have used for our experiments are summarized in Table 3.1.

### 3.4.2 Software Architecture

We implemented our voltage scaling algorithm in the Embedded Configurable Operating system (eCos). eCos is a highly configurable real time operating system

[20]. Every embedded system has needs of its own. Most embedded operating systems today provide more functionality than the actual requirements of individual applications. This extra functionality introduces unneeded complexity into the software. eCos provides the control to the developer to remove such functionality as is not needed. It has more than 200 configuration points (which can be chosen using a configuration tool) and can be used for fine-grained scalability. eCos can provide the basic functionality of an RTOS with memory footprint in tens to hundreds of KBytes. The core functionalities provided by eCos are:

- Hardware Abstraction Layer (HAL)- It provides a software layer that gives general access to hardware.
- Kernel- It includes interrupt and exception handling, thread and synchronization support, a choice of schedulers, timers, counters and alarms.
- ISO C and math libraries
- Device drivers- It includes standard serial, ethernet, Flash ROM and others.
- GNU debugger (GDB) support V It provides target software for communicating with a GDB host.

Both eCos and the application run in supervisor mode. There is no division between user and kernel modes in eCos. It is the responsibility of the application developer to take proper care while writing an embedded application, as a single improper pointer access can wipe out the whole system. In many operating systems, interrupts are disabled during the execution of schedulers but this is not the case with eCos. This keeps interrupt latency low. The eCos kernel implements two static priority based preemptive schedulers:

1. Multilevel Queue Scheduler : This scheduler allows the execution of multiple threads at each of its priority levels. Priority levels can be configured from

1 to 32, corresponding to priority numbers 0 (highest priority) to 31 (lowest priority). This scheduler allows the preemption of a lower-priority thread by a higher-priority thread and it also supports time slicing within a priority level.

2. Bitmap Scheduler : This scheduler allows the execution of threads at multiple priority levels but no two threads can have the same priority level. It also implements preemptive scheduling, though time slicing is irrelevant in this scheduler.

The currently available eCos schedulers don't have the notion of periods and deadlines, and all of them are fixed-priority-based schedulers. We have implemented a dynamic-priority-based scheduler in eCos. This scheduler is based on the Earliest Deadline First (EDF) policy [21]. Priorities are therefore assigned at run time based on the tasks' deadlines.

The bitmap scheduler is a static-priority-based preemptive scheduler. It supports up to 32 tasks (or threads) in the system. The application writer at the beginning assigns a priority to a thread which is retained, unchanged, for the thread's lifetime. No two threads can have the same priority. The periodic nature of real time tasks is captured by a delay function which puts the task in a `sleep` state for a time equal to its period.

The various steps involved in the eCos kernel from thread creation to its execution are:

- `cyg_thread_create()`: This function is used in eCos to create a task. In this we provide the priority at which the task will be run, pointer to the code that will be executed and pointer to stack and name of the thread. As a result of this function a thread is created and is registered in the system with its priority. Initially, thread is created in the sleep state, and is started by:

- `cyg_thread_resume()`: This function resumes a sleeping thread. Upon calling this function the newly created thread starts running.
- `cyg_thread_wait()`: This function takes a “delay” value as its argument, and is provided to put a thread in to sleep mode for a specified amount of time. This function is also used in eCos to implement periodic real time tasks. A periodic task calls this function with delay equal to its period. In this function a timer interrupt is set for a time equal to the current time plus the period of the task and then the scheduler is called to schedule another highest-priority runnable task in the system.

Two most important data structures used in the Bitmap scheduler are:

1. `runqueue`
2. `thread_table`

`runqueue` is a 32-bit value which stores the status (active or sleeping) of each thread in the system, arranged based on their priority level. So `runqueue` will indicate, by the value stored at a particular bit position, whether a task, with priority corresponding to that bit position, is active or not in the system. For example an active task with priority 7 results in 1 at bit position 7 in `runqueue`. The scheduler first scans through this `runqueue` from 0 (i.e. highest priority) to find the highest priority active task in the system and then uses this number as an index to the `thread_table` data structure to determine the corresponding thread pointer. `thread_table` is an array of thread pointers with 32 entries. Once the scheduler discovers the highest priority active task in the system, it uses that priority value to index into the `thread_table` to get the thread pointer and then schedules it.

The bitmap scheduler supports pre-emption. If the currently running task is of a lower priority than the task that just arrived in the system, the scheduler preempts it and schedules this newly-arrived task. This is already supported in eCos.

### 3.4.3 Implementation of EDF in eCos

The task structure in eCos kernel is modified to have the additional parameters:

- WCET
- Period
- Deadline

New APIs are being implemented in EDF to create and run a task using the EDF policy.

- `cyg_edf_thread_create()` : This function is responsible for creating the thread's data structure inside the kernel and is different from the already existing `cyg_thread_create` in a way that it takes Period, WCET as its input parameter.
- `cyg_edf_thread_wait()` : This function puts the current task into sleep mode for a duration equal to its period. In this function we update the priority of all active tasks based on their respective deadlines and then call the scheduler.

We have implemented EDF scheduler on top of Bitmap scheduler. When a task completes execution, call to the native Bitmap scheduler are captured by our EDF scheduler, which first modifies the priorities of individual tasks based on their respective deadlines and then call the Bitmap scheduler.

### 3.4.4 Voltage Scaling algorithm in eCos

We first run the EDF algorithm to generate the offline curve based on the inflated WCET of different tasks. The EDF scheduler in eCos has been modified in a way that it first calculates the online load based on WCET of different tasks and then updates it accordingly whenever a task is done or pre-empted. We have also implemented a DVS thread that compares offline and online load to find the

appropriate CPU speed settings. If the online curve is below the offline curve it determines the intersection point of the offline and online curves and changes the setting as required and then sleeps till that intersection point is reached. Every time a task is done or pre-empted, the scheduler recalculates online and offline loads and wakes up the DVS thread. Tasks are implemented as for loops in the same way as in [2]. Appendix A has the code of our implementation. Appendix B describes our implementation of APIs which interfaces with voltage-clock scaling hardware.

### 3.4.5 Results

The experimental setup for these results is as follows. The execution times of all the tasks are specified in terms of the high-voltage setting. The task periods are chosen randomly to be integers between 200 and 1200 ms. Voltage-speed settings for our experiments are summarized in Table 3.1.

The parameters of importance are the worst-case processor utilization,  $U_w$ , the number of tasks, and the value of  $a$  (the actual execution is uniformly distributed in the range  $[a \times WCET, WCET]$ ). Energy consumption is expressed as a percentage of the consumption if the processor was run at high voltage during execution of the tasks.

The impact of  $a$  on the cpu’s energy consumption for different worst-case utilizations is shown in Figure 3.2. As  $a$  decreases, the variance in the execution time of tasks increases. This means that the a priori information we have about the tasks’ execution times decreases. For a worst case utilization of 0.5 and less, the entire workload can usually be run at low voltage, and so the energy consumption is 40% of the high-voltage consumption. For a worst case utilization of 0.6 and more, the value of  $a$  begins to make an impact on energy consumption. For a small value of  $a$ , execution times of tasks are more variable, which means that (for a given worst case utilization) the system has more run-time slack and must run at a lower voltage

for longer. As  $a$  increases and the execution times do not vary as much any more, the system has less run-time slack and run at high voltage for longer, and thus the energy consumption increases. When the worst case utilization is 0.8 or 0.9, the fraction of time the system has to run at high voltage increases and the energy consumption goes up.

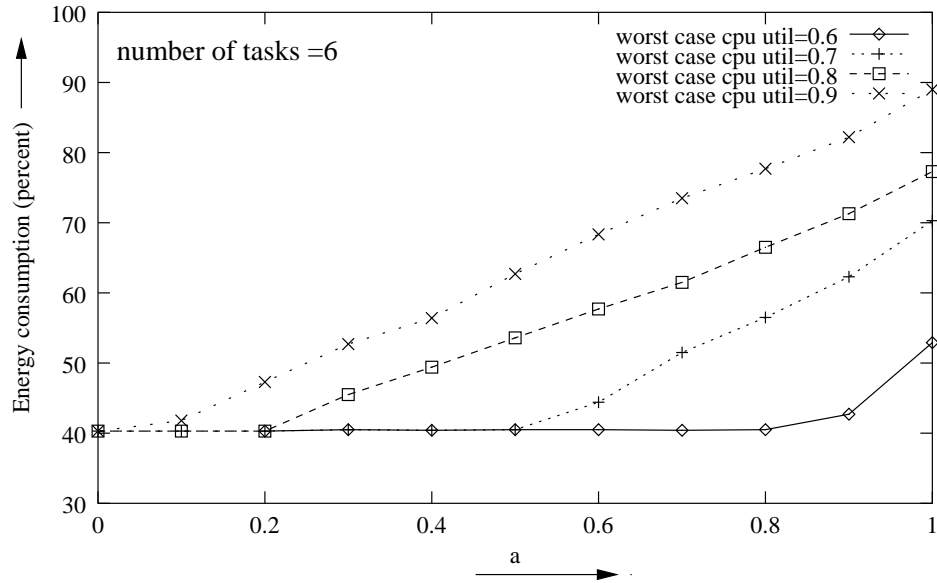


Figure 3.2. Impact of  $a$  on the CPU energy consumption.

The impact of the number of tasks on the system energy is due to the fact that as the number of tasks in the system increases for a given load, the system has more opportunities to more effectively exploit the slack. For example if the system has only 2 tasks, the system will have less opportunity to use for another task the slack released by an early completion. In general, the higher the value of number of tasks, the greater is the advantageous impact of small values of  $a$ . This is illustrated in Figure 3.3 where for a small value value of  $a$ , a 6-task system performs noticeably better then systems with a smaller number of tasks, but this gap narrows as we move to a higher value of  $a$ , as shown in Figure 3.4

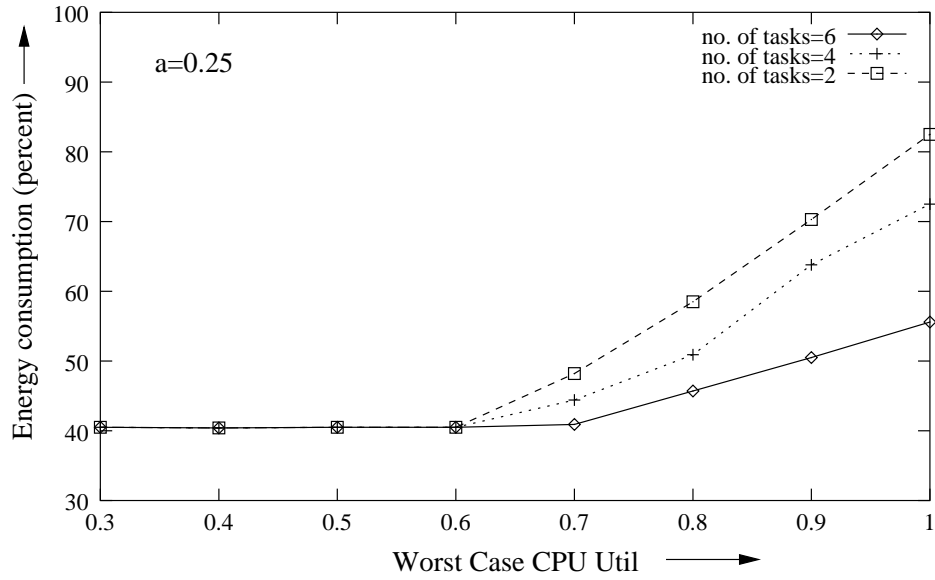


Figure 3.3. Impact of the number of tasks on the CPU energy consumption.

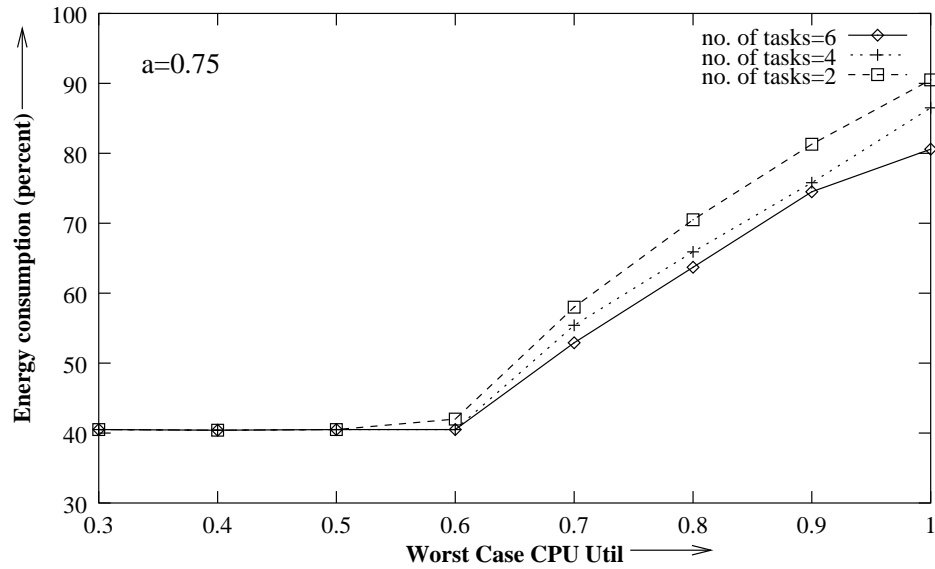


Figure 3.4. Impact of number of tasks on the CPU energy consumption.

## C H A P T E R 4

### FAULT-TOLERANT VOLTAGE SCHEDULING ALGORITHM

Embedded systems are often subjected to faults due to harsh environmental conditions. These systems are also constrained by severe power and / or energy limits. Examples include space systems relying on solar and battery power supply, and seaborne systems working off a limited battery supply. In this paper, we study a tradeoff between reliability and energy for an embedded system consisting of multiple real-time tasks.

Dynamic Voltage Scaling has emerged as a powerful mechanism for reducing the energy consumption during system operation. And many current embedded processors like Intel Xscale, Transmeta Crusoe allow their operating voltage to be changed dynamically. Since a reduction in voltage leads to a drop in processor speed and thus increase in the execution time, a number of techniques have been proposed to exploit this tradeoff.

Real time tasks are time constrained, i.e. the tasks have a fixed deadline by which they have to finish execution. At each checkpoint, the system state is saved and when a fault is detected, the system is restarted from the most recent available checkpoint. There is a cost associated with checkpointing, which includes saving the process state to a secure device. Without checkpointing, a fault can lead to restart of task from the beginning. Checkpointing reduces re-computation time due to faults.

Real time tasks normally do not run to their worst case execution time and this time can be reclaimed during the run time In this chapter we try to study the

effect of dynamic resource reclamation, dynamic power management using dynamic voltage scaling and fault tolerance using checkpointing on energy. Then we extend this to study the tradeoff between energy and the reliability arising because of slack reclamation. To the best of our knowledge, this is the first work that addresses all these issues simultaneously.

If we run a processor in low voltage for a longer time, we have less time to run the lost computation, but at the same time the energy savings are higher so there is a tradeoff. This chapter studies this tradeoff. As a first step we try to extend the dynamic resource reclamation to a fault tolerant system for energy reduction and then we do the slack reclamation to study this tradeoff. During normal operation, we do dynamic resource reclamation and we also exploit slack to conserve energy such that tasks can meet their deadline and during a failure, we run the system at maximum speed to execute the lost computation. We also discuss the implementation issues related to algorithm presented here.

#### 4.1 Model and Terminology

This algorithm is an extension to the voltage scaling algorithm described in the previous section. We use the task and power model described in chapter 3.

##### 4.1.1 Fault and Recovery Model

We assume a real time system that is subjected to transient faults with the following characteristics: Failures occur the system as a Poisson process with a fixed failure rate. The lifetime of a transient fault is exponentially distributed with a parameter  $\mu$ . We assume a real-time system that takes checkpoints and stores them in a secure place. The inter-checkpoint interval for all tasks is the same and we assume that checkpoints are not corrupted. The cost associated with the checkpointing is assumed same for all the checkpoints.

## 4.2 Algorithm

This algorithm always sets the processor voltage to  $v_L$  when the unfinished work is guaranteed to be below  $\psi(t)$  and no fault hits the system, and  $v_H$  otherwise.

Unfinished work in the system is calculated in the same way as described in Section 3.2. At any time,  $t$ , the processor will be running at low voltage except when any of the following conditions is satisfied:

- 1a.  $\text{offline\_task}(t) = \text{online\_task}(t)$
- 1b. the worst case unfinished work under the online algorithm is equal to  $\psi(t)$ .
2. A fault hits the system.

When the system is operating in normal mode (no faults in the system), the above-mentioned algorithm guarantees that no task will miss its deadline. In the event of failure, our algorithm adopts a pessimistic approach and tries to reduce the chances of missing the deadline by setting the voltage to  $v_H$  for rest of the frame. The chance of missing a deadline is governed by the timing of fault arrival, transient lifetime and tradeoff factor.

## 4.3 Results

The experimental setup for our simulations is as follows. All execution times are specified in terms of high voltage and are in milliseconds. The task periods are chosen randomly from this set of values 40, 50, 80, 100, 200. The reason behind this approach is to limit the Least Common Multiple (LCM) to a manageable value and thus to speed up the simulation runs. The processor has been modeled based on the technology used for the Intel Xscale processor. The parameters values used are:

The parameters of importance are the worst-case processor utilization,  $U_w$ , the average processor utilization,  $U_{av}$ , the number of tasks,  $n$ , the value of  $a$ , failure rate, recovery time, checkpointing cost, tradeoff factor. Energy consumption is expressed

Voltage	value
$v_H$	1.75
$v_L$	1.00
$v_T$	0.20

Table 4.1. Voltage Levels

Parameter	value
failure rate	$1 \times 10^{-5}/\text{msec}$
checkpointing cost	0.5 msec
no. of tasks	4

Table 4.2. System parameters

as a percentage of the consumption if the processor was run at high voltage when executing the workload.

First consider the effect of the the inflation factor on failure probability, shown in Figure 4.1. Table 4.2 summarizes the values of different parameters. As we increase the inflation factor, we consume more of the available slack in the schedule and thus increase the failure probability. But at the same time we save more energy. The greater the recovery time, the greater are the chances to miss a deadline during recovery and thus the greater the failure probability. Recovery times have no measurable impact on the energy curve, because failure is a very rare event and thus the energy numbers are dominated by the no-failure case. In Figure 4.2, all the parameters are kept the same and  $U_w$  is increased.

In Figure 4.3, we demonstrate the effect of checkpointing cost on failure probability when dynamic resources are reclaimed using DVS. Initially as we increase the checkpointing interval, the failure probability decreases because it reduces the number of checkpoints in the system and hence reduces the execution time and thus increases the slack to recover from failures. But if we keep increasing the checkpointing interval, the failure probability increases because more computation

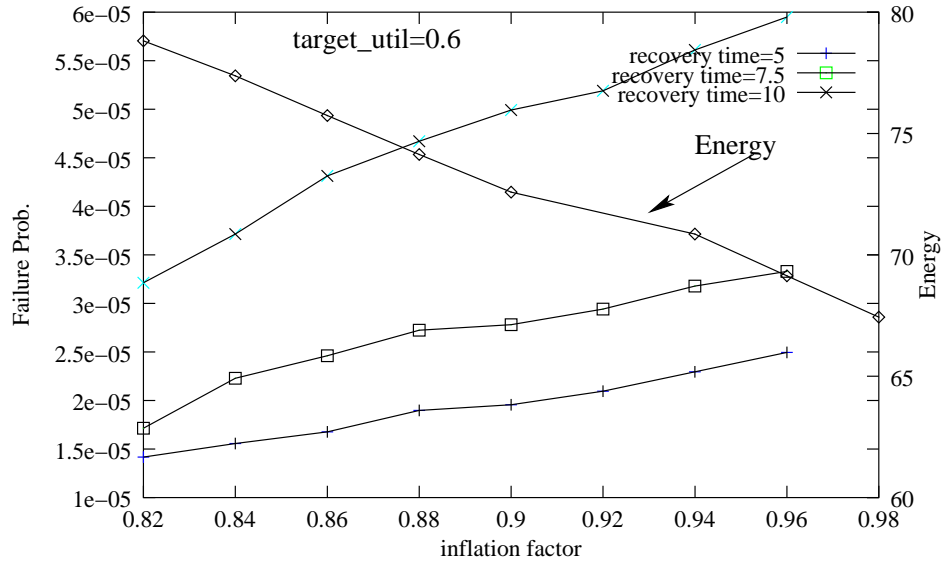


Figure 4.1. Impact of inflation factor on failure probability and CPU's energy consumption for worst case target utilization of 0.6

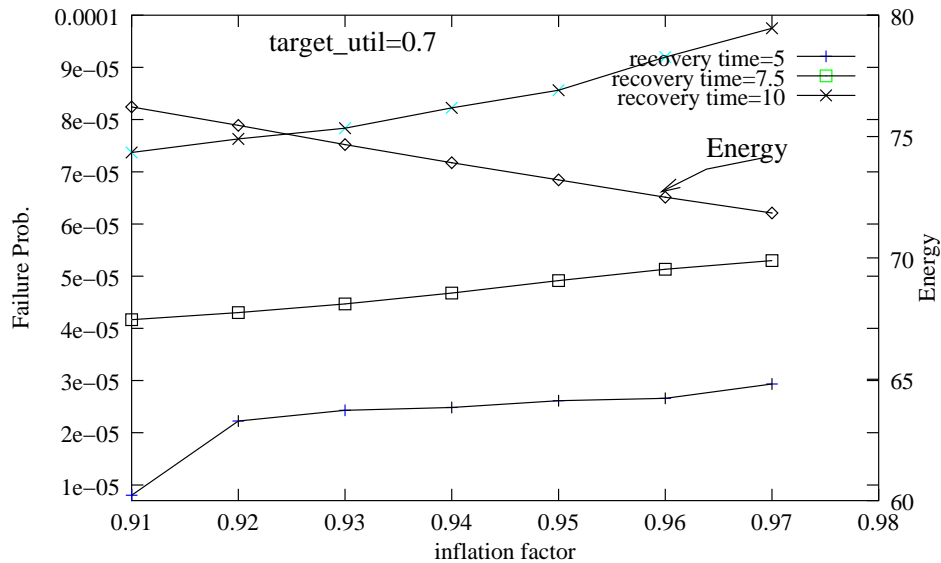


Figure 4.2. Impact of inflation factor on failure probability and CPU's energy consumption for worst case target utilization of 0.7

is lost upon failure.

In Figure 4.4, failure probability is plotted against energy, for different values of the checkpointing interval. System has 6 tasks. We obtain the failure probability and energy consumed by varying the checkpointing interval. Different curves are

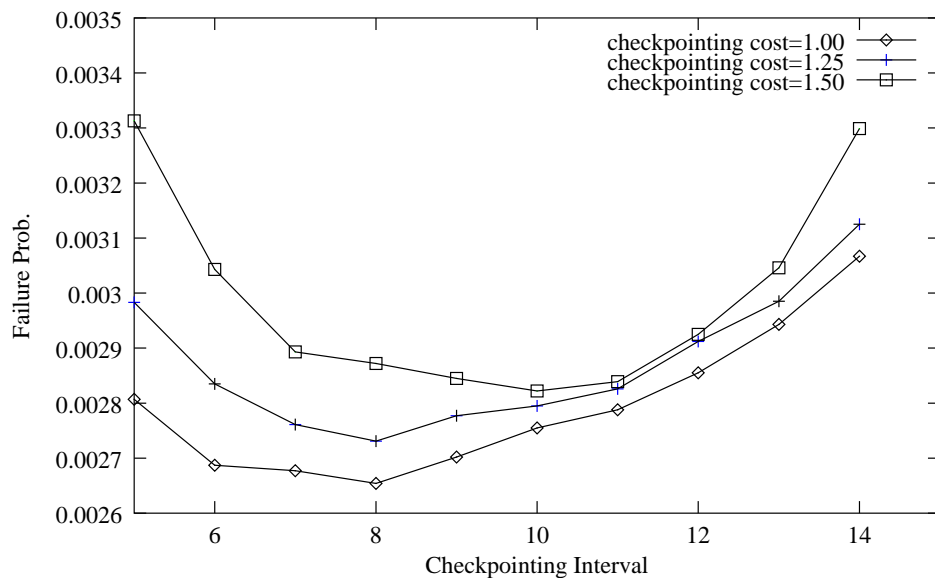


Figure 4.3. Impact of checkpointing overheads on failure probability

plotted for different inflation. This curve tell us the number of checkpoints and amount of inflation needed to provide the optimal reliability for a given energy budget.

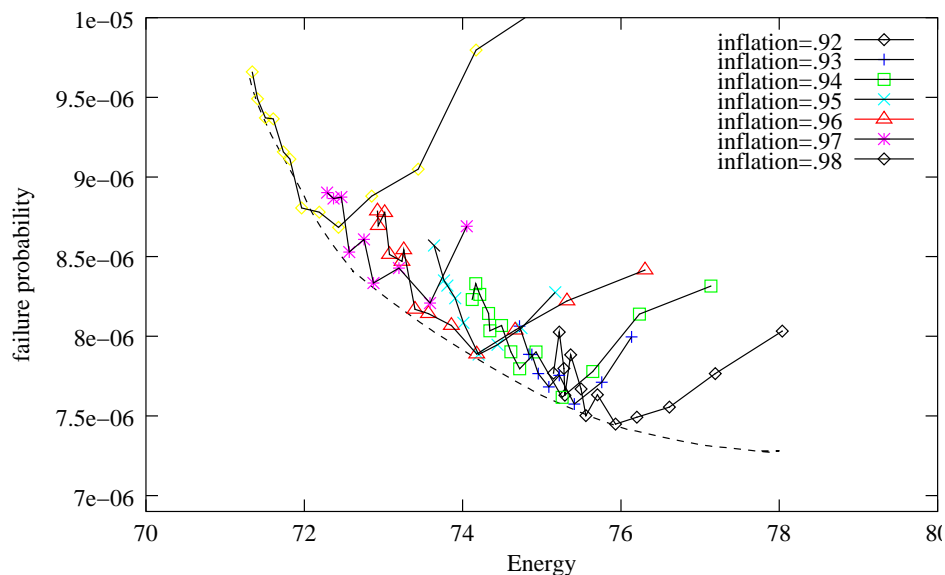


Figure 4.4. Impact of failure probability on CPU energy consumption.

## C H A P T E R 5

### IMPLEMENTATION OF CHECKPOINTING IN RTOS

#### 5.1 Introduction

Checkpointing involves storing the state of a process at any point of time and restarting from that point in the event of a fault. For long-running applications, checkpointing allows users to limit the amount of lost computation upon a failure without having to rerun the entire application. In a hard-real-time system where applications have a fixed deadline and missing those deadlines can be catastrophic, checkpointing can be effectively used.

The main goal of this work is to implement checkpointing in a freely available and widely used RTOS. Checkpointing is already available for Unix based OS (like Linux) [9, 13]. Linux is free, embeddable and is able to perform efficiently with small RAM based computer systems. However, Linux is not a hard real-time OS. Linux uses disabling of interrupts for providing synchronization inside the kernel while executing critical section, which compromises the systems' ability to promptly respond to external events and brings unpredictability in interrupt dispatch latency. Other shortcomings in Linux from the hard real-time perspective include time-sharing (normally 10 msec) scheduling, virtual memory system timing unpredictability, and lack of high-granularity timers. We have implemented checkpointing in real-time Linux (Rtlinux) [14]. Rtlinux deals with all the shortcomings of Linux and presents Linux as a hard real-time OS. In Rtlinux, to remove the unpredictability in interrupt dispatch latency, a layer of emulation software is placed between the Linux kernel and the interrupt controller hardware. In this way all the hardware interrupts are

caught by the emulator. This emulator checks whether Linux interrupts are enabled or disabled. If enabled, the Linux interrupt handler is invoked immediately. If Linux interrupts are disabled, the handler is not invoked. Instead, a bit is set in the variable that holds the information about all pending interrupts and later when Linux re-enables interrupts, the handlers of all the pending interrupts are executed. These simulated interrupts are also known as soft interrupts. In this way, Rtlinux runs Linux as one of its threads. When there is no real-time thread to schedule, the Rtlinux scheduler schedules Linux. Since Linux has no direct control over the interrupt controller, it does not influence processing of real-time interrupts that do not pass through the emulator. Rtlinux also uses disabling on interrupts inside its critical section, but Rtlinux interrupts are fast and processing is kept minimal inside Rtlinux interrupt service routines. Due to this, the interrupt dispatch latency can be bounded with reasonable certainty. In Rtlinux, because the timer chip is put in the interrupt-on-terminal-count mode, an interrupt can be scheduled with approximately 1 microsecond precision. In this way the need for a periodic timer interrupt is removed and the CPU is interrupted only when needed. Periodic interrupts are simulated for Linux using soft interrupts. Rtlinux runs all RT-tasks in the kernel address space. By running all the tasks in a single address space overheads related to virtual memory are reduced. Because the kernel address space is used, overheads associated with protection level changes are also eliminated. Task switching is also easier if all tasks run in one address space because a context switch requires pushing all the registers on the stack and changing the stack pointer to point to the new stack. Rtlinux thus fits best for hard real-time systems where predictability is of most importance.

To use checkpointing in Rtlinux, only minor changes are needed in the RT-tasks. Upon processor failure, the application can be restarted with the command line flag on the processor on which its state is saved, and the program will be restarted from

the most recently checkpointed state. Currently it has been implemented on Intel architecture, though extending it to other architectures should be straightforward.

## 5.2 Usage

To add checkpointing support in Rtlinux applications, the programmer has to add one line to include the checkpoint header file in the application source file, for example:

```
#include ‘‘rtlckpt.h’’
```

Then the user has to determine a suitable place where he/she wants to take the checkpoint and also has to allocate some shared memory. Though the user doesn't have to modify the application code heavily, he/she has to just place a function `rtl_check_for_recovery()` in the beginning of the `start_routine()` i.e. in the beginning of the application code and has to place a function `rtl_ckpt_here()` at the point where he/she wants to take the checkpoint in the application. In RTlinux an application is an object file and is loaded into the kernel address space in the following way: `insmod rtl_application.o` In order to run the Rtlinux application during recovery the user has to give a command line argument with the module loader: `insmod rtl_application.o 1`.

## 5.3 Implementation details

Rtlinux applications' state includes:

1. Code Segment
2. Stack Segment
3. Registers

When a Rtlinux application is loaded using a kernel module loader, its code gets attached to the kernel code segment right below it. The code segment of same application on different machine can have different (virtual) addresses. For this reason, the program counter on one machine need not point to the same instruction

on another machine for the same application. So we must have some mechanism to adjust the program counter according to the new code segment addresses. In Rtlinux memory space for the stack is allocated using `kmalloc` and the base address of the stack is the return value of `kmalloc`. This implies that the address of the stack segment will be different for the same application on different machines. Thus, there must be some mechanism to make the address of the stack segment the same for a particular application across the machines. Another problem is that we cannot use `setjmp` and `longjmp` in Rtlinux to save the register context. So alternative functions have to be implemented.

In our implementation we have taken care of the above-mentioned problems to implement checkpointing in Rtlinux in the following way:

### 5.3.1 Program Counter Adjustment

If the application is running in the normal mode, we just store its PC at a fixed point (say PC1). Then while taking the checkpoint we store the program counter from where we have to restart the application (PC2) during recovery and we calculate

$$Disp = PC2 - PC1$$

When the same application is running in the recovery mode on some other node it again calculates the PC at the same fixed point and then the actual PC (for restarting) is calculated using

$$PC2 = PC1 + DISP$$

In this way we were able to solve the problem of different program counters across machines.

### 5.3.2 Stack Segment

As mentioned above, the stack and frame pointers may be different for the same application on different machines. So in our checkpointing scheme, we save the difference of the stack and frame pointers with respect to the stack base and then add

these displacements in the base address of the stack allocated to the application on the other node to get the right value. This, however, does not work for applications with arrays. The start address of an array is at some location in the stack and must be changed according to the new address space, which is difficult to implement.

A possible solution of the above problem is to allocate the same address space to the stack segment of the Rtlinux application across machines. However doing so is a difficult task particularly when `kmalloc` is used for memory allocation. A brute-force solution is to modify the existing `kmalloc` for this purpose. The kernel for memory allocation also uses `kmalloc` and thus modifying it can introduce unpredictability in the system. Our solution is fairly simple. It is based on the concept of reserving some memory at boot time and then using it for stack allocation instead of using `kmalloc()`. This concept is widely used in Rtlinux for allocating shared memory for communication between a Rtlinux application and a user space application and is taken from there. The problem with this solution is that it guarantees that the physical address will be the same but the virtual address that we get after remapping need not be the same across machines. But this does not restrict us in anyway because any difference in the addresses can be adjusted, as that difference will remain the same as long as the kernel remains the same.

### 5.3.3 Registers

`setjmp` and `longjmp` are widely used library calls for storing register context in checkpointing [9, 10] but unfortunately Rtlinux does not support them. We have defined our own functions `rtl_save_registers()` and `rtl_restore_registers()` for this purpose. `rtl_save_registers()` stores the values in registers on the stack during checkpoint and `rtl_restore_registers()` pops them out from the stack during recovery.

## 5.4 Transfer of Checkpointed State

In order to transfer the checkpointed state, the entire process state needed for restart, is first written into shared memory and this shared memory is linked to a Linux application running as a server which is responsible to transfer it to another Linux application running as a client on some other machine. This client application then writes this state in a shared memory region, which is also linked to the Rtlinux application.

## 5.5 Recovery process

The main aim of checkpointing is to reconstruct the recovery point. In checkpointing, the recovery point includes creating the process and restoring data state, processor state [1] which involves the following steps:

1. Process creation is implemented by invoking the checkpointed program with a special command line argument for recovery. This restores the code segment of the process's state and begins execution. The application itself detects the argument for recovery in `rtl_check_for_recovery()` and then calls the recovery routines.
2. The recovery routine reads the checkpoint from the shared memory and recreates the process's stack segment.
3. Processor state restoration requires the processor registers, including the program counter and stack pointers to be restored to their values when the checkpoint was taken. In our implementation, we are using `rtl_save_registers()` to store the processor state and `rtl_restore_registers()` to restore the processor state. Hence, the recovery routine never returns and execution continues as if it is returning from `rtl_ckpt_here()`.

The system's state (fixed resources) such as the shared memory, FIFO, and sockets is unrestorable in our implementation as is also the case with other implementations of checkpointing.

## 5.6 Experimental results and analysis

To validate our implementation, we have tested it on a number of applications without encountering any problems during recovery. These experiments were performed on an Intel Pentium II machine running Rtlinux version 3.0 running on them. A total of 10 readings were taken and then averaged. Below we are showing the results for some applications:

1. Matrix Square: This is a straightforward matrix multiplication. An  $100 \times 100$  matrix is initialized using a random number generator and then its square is calculated and stored in another  $100 \times 100$  matrix.

Execution time without checkpointing=29,731,360 ns

Application size without checkpointing (after loading as kernel module)=1200 bytes

Stack Size=100 KB

Checkpoint taken at row=60

Execution time with checkpointing=30,583,488 ns

Application size with checkpointing=3400 bytes

Time spent in checkpointing routines=8,27,136 ns

Upon restarting it on another node:

Time spent in recovery routines=1,642,880 ns

2. Insertion Sort: This is a standard sorting algorithm. An array of size 10000 was randomly initialized and then it was sorted using Insertion Sort.

Execution time without checkpointing=12,06,900,020 ns

Application size without checkpointing (after loading as kernel module)=800 bytes

Stack Size=60KB

Checkpoint has been taken when array index was=6000:

Execution time with checkpointing=12,07,405,000 ns

Application size with checkpointing (after loading as kernel module)=3000 bytes

Time spent in checkpointing routines=4,69,312 ns

On restarting it on another node:

Time spent in recovery routines=8,46,816 ns

So the total execution overhead per checkpoint for matrix multiplication is 2.7% and for insertion sort is .04%. This difference is because insertion sort is very computation intensive application and also the stack size needed for this application is smaller compared to matrix multiplication. The increase in size of application code appears to be large but this application is a small application and the size of checkpointing code is fixed so with bigger applications, the increase in size will be insignificant.

## 5.7 Limitations

1. Checkpointing can be done in `main()` only since if done in some function, the return address of that function, saved on the stack, needs to be changed, which is difficult to implement in the case of recursive functions.
2. No function pointers are allowed in the application, as function pointers have addresses of the code segment, which are different on different machines. We need to modify them to point to the correct location. This cannot be generalized and will make our implementation application-specific.

## C H A P T E R 6

### CONCLUSIONS AND FUTURE DIRECTIONS

In this thesis we have looked at the problems of power-aware scheduling and fault tolerance for real-time systems. We have also simulated a low power fault tolerant system and studied various tradeoffs between reliability and energy consumption of a system.

We have implemented a dynamic voltage scaling based scheduler in eCos. This scheduler provides a way to claim run-time slack in the system without missing any hard deadline and can be used as an effective solution for power constrained hard real-time system for e.g seaborne systems. We have modified an Intel Xscale board for DVS to validate our algorithm.

We have demonstrated that the fault tolerance of a system can be traded off for energy consumption. This algorithm is an extension to our voltage scaling algorithm. We reduce the power consumption during normal conditions by running the cpu in lower speed (claiming both run-time and static slack) and try to reduce the chances of missing the deadlines by running the processor at highest speed in an event of failure. We have also presented simulation results for the tradeoff between fault tolerance and energy. We have also discussed the implementation issues related with this algorithm.

We have also implemented a checkpointing library `rtlckpt` for Rtlinux applications that provide fault tolerance for real time applications. It is very easy to use and has very low overhead associated with it. We have described various problems in implementing checkpointing in an RTOS and how we have overcome them. Various limitations to our implementation are also described.

There are several directions in which future work can go. One of them is to incorporate checkpointing and DVS in a system and then study various tradeoffs between reliability and energy consumption from a practical point of view. Implementing checkpointing for a multithreaded application in an RTOS is another avenue of research.

We have implemented a system that has two voltages level and can be modified for multiple voltage-speed settings. Another direction of research is to implement a DVS based scheduler for IRIS tasks and tasks with precedence graphs.

Overall, we have demonstrated the energy savings achieved by incorporating dynamic voltage scaling on an commercial microprocessor and using a power aware embedded OS. Our DVS-enabled system (OS and hardware) when used for embedded applications, can lead to significant energy savings. We have also presented checkpointing library that can be linked with an real time application for fault tolerance purposes.

## REFERENCES

- [1] C. M. Krishna and Yann-Hang Lee, "Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real Time Systems," *IEEE Real-Time Technology and Applications Symposium*, May 2000, pp. 156–165.
- [2] Y. Shin, K. Choi, and T. Sakurai, "Power optimizations of real-time embedded systems on variable speed processors," *Proc. Int. Conf. Computer-Aided Design*, pp. 365-368, June 2000.
- [3] "Energy efficient fixed-priority scheduling for real-time systems on variable speed processors," *Proc. Design Automation Conference*, pp. 828-833, June 2001.
- [4] T. Ishihara, and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," *Proc. Int. Symp. Low Power Electronics and Designs*, 1998, pp. 197–202.
- [5] C. M. Krishna, Kang G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Trans. on Computers*, Vol. C-35, NO. 5, May 1986.
- [6] Krishna CM, Singh AD, "Reliability of Checkpointed Real-Time Systems using Time Redundancy," *IEEE Trans. on Reliability*, Sep. 1993, pp. 427–435.
- [7] S. W. Kwak, B. J. Choi and B.K. Kim, "An Optimal checkpointing strategy for real time control system under transient faults," *IEEE Trans. Reliability*, Sep. 2001, vol. 50, pp. 293-301.
- [8] Kang G. Shin, Tein-Hsiang Lin, Y.H. Lee, "Optimal Checkpointing of Real-Time Tasks," *IEEE Trans. on Computers*, Vol. C-36, NO. 11, Nov. 1997.
- [9] J. Plank, M. Beck, G. Kingsley. , "Libckpt: Transparent Checkpointing under Unix." Princeton University".
- [10] William R. Dieter, James E. Lumpp Jr, "A User-level Checkpointing Library for POSIX Threads Programs, University of Kentucky." *Academic Press Inc, New York, NY*, 1997.
- [11] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny, "Checkpoint and migration of unix processes in the condor distributed processing system," *Technical Report 1346, University of Wisconsin-Madison Computer Science*, April 1997.

- [12] Eduardo Pinheiro, “Truly-Transparent Checkpointing of Parallel Applications (Working Draft),” *Federal University of Rio de Janeiro UFRJ*.
- [13] Hua zhong and Jason Nieh., “CRAK:Linux Checkpoint/Restart As a Kernel Module,” *Technical Report, department of computer science, Columbia University*.
- [14] The RT-Linux Operating System. <http://www.fsmlabs.com/community/>.
- [15] Jayanta K. Dey, Donald F. Towsley, C. M. Krishna, and Mahesh Girkar, “Efficient On-Line Processor Scheduling for a Class of IRIS (Increasing Reward with Increasing Service.),” *Real-Time Tasks.SIGMETRICS*, 1993, pp. 217–228.
- [16] Transmeta Corporation. <http://www.transmeta.com/>.
- [17] <http://www.intel.com/design/intelxscale/>.
- [18] ADI Engineering Inc. BRH reference platform. <http://www.adiengineering.com/productsBRH.html>
- [19] Liu, C. L., and Layland, J. W., “Scheduling algorithms for multiprogramming in hard real-time environment,” *J. ACM* 20, 1 , Jan. 1993, pp. 46–61.
- [20] Osman S. Unsal, Israel Koren, C. Mani Krishna., “Towards Energy-Aware Software-Based Fault Tolerance in Real-Time Systems,” *Proc. International Symposium on Low Power Electronics and Design, ISLPED’02*, August 2002, pp. 124–129.
- [21] Y. Zhang and H. Chakrabarty, “Energy-aware adaptive checkpointing in embedded real time systems,” *Proc. Designs Automation and Test in Europe Conference*, 2003, pp. 918–923.
- [22] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez, “Determining optimal processor speeds for periodic real-time tasks with different power characteristics,” *Euromicro Conference on Real-Time Systems*, 2001.
- [23] T. Pering, T. Burd, and R. Brodersen, “The simulation and evaluation of dynamic voltage scaling algorithms,” *International Symposium on Low-Power Electronics and Design*, 1998, pp. 76–81.
- [24] A. Chandrakasan and R. Broderson, “Low Power Digital CMOS Design, Kluwer Academic Publishers, Norwell, MA, 1995.”
- [25] C. Hwang and A. C-H. Wu., “A predictive system shutdown method for energy saving of event-driven computation,” *Proc. Intl. Conf. Computer-Aided Design*, 1997, pp. 28–32.
- [26] M. B. Srivastava, A. P. Chandrakasan and R. W. Broderson, *IEEE Tran. VLSI Systems*, vol. 4, pp. 42-45, 1996..

- [27] Advance Configuration and Power Interface (ACPI), "<http://www.acpi.info/>" .
- [28] V. Raghunathan, P. Spanos and M. B. Srivastava., "Adaptive power-fidelity in energy-aware embedded systems," *Proc. Design Automation Conf.*, 2001, pp. 106-115.
- [29] The embedded Configurable operating system (eCos), "[www.redhat.com/embedded/technologies/ecos](http://www.redhat.com/embedded/technologies/ecos)" .
- [30] Padmanabhan Pillai and Kang G. Shin., "Real-time dynamic voltage scaling for low power embedded operating systems," *18th ACM Symposium on Operating Systems Principles*, October 2001, pp. 89-102.
- [31] Y.-H. Lee and C. M. Krishna, "Voltage-clock scaling for low energy consumption in fixed-priority real-time embedded systems," *Real-Time Systems*, to appear.
- [32] Cristiano Pereira, Vijay Raghunathan, Shalab Gupta, Rajesh Gupta, and Mani Srivastava, "A Software Architecture for Building Power Aware Real Time Operating Systems," *Technical Report #02-07*, March 14, 2002 .
- [33] Sandeep Padmanabhan, *Private communication*.

## A P P E N D I X A

### AN IMPLEMENTATION OF FOUR TASKS' TASKSET WITH DYNAMIC VOLTAGE SCALING THREAD IN eCos

In this section an example of a four tasks' taskset is presented. This is a synthetic task set and is same as in [32]. The function `cyg_user_start` creates task instances using `cyg_edf_thread_create`. This function is different from traditional `cyg_thread_create` in a way that it takes WCET and PERIOD of a task as an input parameters and pass it on to EDF scheduler implemented inside the eCos Kernel. DVS Thread implements our voltage scaling scheme and its core functionality is defined in `dvs_program`.

```
/*-----  
  
Author(s): Akash Goel  
Date: 09/09/2004  
  
-----*/  
  
#include <cyg/kernel/kapi.h>  
#include <stdio.h>  
#include <math.h>  
#include <stdlib.h>  
#define NUMBER_OF_TASKS 4
```

```

/* space allocation for kernel objects, for various threads in
the system*/

/* We will always use NUMBER_OF_TASKS + 1 while allocating space,
in order to create space for DVS thread in the system*/

cyg_thread thread_s[NUMBER_OF_TASKS + 1]; /* Allocating space
for thread objects*/

char stack[NUMBER_OF_TASKS + 1][4096]; /* Allocating space for
stacks */

cyg_handle_t simple_threadA, simple_threadB, simple_threadC,
simple_threadD, dvs_thread;

/* to generate ACET such that it varies uniformly between a*WCET
and WCET*/

double a =0.75;

double unif[10]={0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0};

cyg_thread_entry_t dvs_program;
cyg_thread_entry_t simple_program1;
cyg_thread_entry_t simple_program2;
cyg_thread_entry_t simple_program3;
cyg_thread_entry_t simple_program4;

```

```

cyg_mutex_t cliblock;

void cyg_user_start(void)
{
int i;

/* first setting cpu's speed to max 733 Mhz. so current settings
are:733Mhz, 1.5V, below are the Intel assembly instructions to set
cpu's speed to 733 Mhz*/

__asm__("MOV R1, #9");
__asm__("MCR p14, 0, R1, c6, c0, 0");

printf("Entering cyg_user_start() function\n");

    cyg_mutex_init(&cliblock);

/*The highest priority thread is DVS thread*/

/*cyg_edf_thread_create(priority, thread's routine, data, thread's
name, thread's routine, stack pointer, stack size, thread's handle,
pointer to thread's data structure, PERIOD, WCET)*/

    cyg_edf_thread_create(1, complex_program, (cyg_addrword_t)0,
        "Thread DVS", (void *) stack[0], 4096,
        &complex_thread, &thread_s[0], 0, 0);

```

```
cyg_edf_thread_create(2, simple_program1, (cyg_addrword_t) 1,
    "Thread A", (void *) stack[1], 4096,
    &simple_threadA, &thread_s[1], 300, 60);

cyg_edf_thread_create(3, simple_program2, (cyg_addrword_t) 2,
    "Thread B", (void *) stack[2], 4096,
    &simple_threadB, &thread_s[2], 400, 40);

cyg_edf_thread_create(4, simple_program3, (cyg_addrword_t) 3,
    "Thread C", (void *) stack[3], 4096,
    &simple_threadC, &thread_s[3], 500, 100);

cyg_edf_thread_create(5, simple_program4, (cyg_addrword_t) 4,
    "Thread D", (void *) stack[4], 4096,
    &simple_threadD, &thread_s[4], 600, 60);

cyg_thread_resume(dvs_thread);

cyg_thread_resume(simple_threadA);

cyg_thread_resume(simple_threadB);

cyg_thread_resume(simple_threadC);

cyg_thread_resume(simple_threadD);

}
```

```

/* this is a code which runs in a thread */

void simple_program1(cyg_addrword_t data)
{
    int message = (int) data;
    unsigned int i, j, number_of_ticks;

    /*This is to generate actual execution time, uniformly varies
between a*WCET and WCET */
    number_of_ticks= (a + (1-a)*unif[rand() % 10])*60;
    printf("Beginning edf execution; thread data is %d\n", message);

    for (;;) {

        cyg_mutex_lock(&cliblock); {
            printf("Thread %d is running\n",message);
        }
        cyg_mutex_unlock(&cliblock);
        for(j=0;j<number_of_ticks;j++)
/* a for loop with number of iteration=2390000 consumes one tick
at 733Mhz*/
for(i=0;i <2390000;i++)
;

```

```
/* cyg_thread_delay is replaced with cyg_thread_wait. This function  
put a thread in sleep state, finds out its next arrival, call  
scheduler to adjust threads' priorities based on their deadlines and  
schedule next highest priority thread*/
```

```
    cyg_thread_wait();  
}  
}
```

```
void simple_program2(cyg_addrword_t data)
```

```
{  
  
    int message = (int) data;  
    unsigned int i, j, number_of_ticks;  
  
    number_of_ticks= (a + (1-a)*unif[rand() % 10])*40;  
    printf("Beginning edf execution; thread data is %d\n", message);  
  
    for (;;) {  
        cyg_mutex_lock(&cliblock); {  
            printf("Thread %d is running\n",message);  
        }  
        cyg_mutex_unlock(&cliblock);  
        for(j=0;j<number_of_ticks;j++)  
    for(i=0;i <2390000;i++)
```

```

;

    cyg_thread_wait();
}
}

void simple_program3(cyg_addrword_t data)
{
    int message = (int) data;
    unsigned int i, j, number_of_ticks;

    number_of_ticks= (a + (1-a)*unif[rand() % 10])*10;
    printf("Beginning edf execution; thread data is %d\n", message);

    for (;;) {

        cyg_mutex_lock(&cliblock); {
            printf("Thread %d is running \n",message);
        }

        cyg_mutex_unlock(&cliblock);
        for(j=0;j<number_of_ticks;j++)
for(i=0;i <2390000;i++)
;

        cyg_thread_wait();
}
}

```

```
}
```

```
void simple_program4(cyg_addrword_t data)
```

```
{
```

```
    int message = (int) data;
```

```
    unsigned int i, j, number_of_ticks ;
```

```
    number_of_ticks= (a + (1-a)*unif[rand() % 10])*50;
```

```
    printf("Beginning edf execution; thread data is %d\n", message);
```

```
    for (;;) {
```

```
        cyg_mutex_lock(&cliblock); {
```

```
            printf("Thread %d is running\n",message);
```

```
        }
```

```
        cyg_mutex_unlock(&cliblock);
```

```
        for(j=0;j<number_of_ticks;j++)
```

```
for(i=0;i <2390000;i++)
```

```
;
```

```
        cyg_thread_wait();
```

```
    }
```

```
}
```

```
void dvs_program(cyg_addrword_t mesg_a)
```

```

{
int mesg=(int) mesg_a;
volatile unsigned int x, offline_load, online_load;
unsigned int temp=0;
x=0;

/*initial online_curve_slope will be set as 1 because initial
settings are 733Mhz, 1.5 i.e. is high voltage*/

printf("Beginning DVS Thread's execution; thread data is %d\n",
mesg);

/*these function will initialize offline curve and online curve*/
cyg_init_offline_curve();
cyg_init_online_curve();

while(1){

/* Whenever we starts DVS thread, first things is to get online and
offline load and compare them*/
offline_load=cyg_find_offline_load();
online_load=cyg_find_online_load();

        cyg_mutex_lock(&cliblock); {

printf("offline_load is: %u and online load is: %u\n", offline_load,
online_load);

```

```

    }

    cyg_mutex_unlock(&cliblock);
if(offline_load<=online_load){

/*cyg_online_curve_slope return true if we are at high volatge else
false*/

if(!cyg_online_curve_slope()){

/* cyg_do_dvs chnages the cpu's speed-voltage settings*/

cyg_do_dvs(); /*change the voltage speed
settings to 1.5V and 733Mhz.*/

cyg_thread_delay(100000); //sleep forever
}
else{
cyg_thread_delay(100000); //sleep forever
}
}
else{

if(cyg_online_curve_slope()){

x=cyg_find_intersection_point();

```

```

temp= x - cyg_current_time();
//printf("intersection point is: %u\n", temp);

cyg_do_dvs(); /*change the voltage speed
settings to 1.2V and 400 Mhz.*/

cyg_thread_delay((cyg_tick_count_t)temp);

}

else{
/*cyg_find_intersection_point returns the time when offline curve and
online curve will intersect*/
x=cyg_find_intersection_point();
temp=x - cyg_current_time();
//printf("intersection point is: %u\n", temp);
cyg_thread_delay((cyg_tick_count_t)temp);
}
}
}

}

```

## A P P E N D I X B

### IMPLEMENTATION OF VOLTAGE-CLOCK SCALING SOFTWARE

In this section, we are presenting the implementation of function `do_dvs` inside Power aware EDF scheduler which provides an interface to Voltage-Clock scaling hardware.

```
void Cyg_Scheduler::do_dvs()
{
    CYG_REPORT_FUNCTION();
    unsigned char *com2;
    com2= (unsigned char *)0x3100000;
    //0x3100000 is mapped to RTS line

    /* A true value of high_slope indicates a 1.5V-733Mhz
       setting and a false value indicates 1.2V-400Mhz
       setting */

    if(high_slope)
    {
        /****** high to low voltage change******/
        /*code for speed change*/
        __asm__("MOV R1, #4");
        __asm__("MCR p14, 0, R1, c6, c0, 0");
    }
}
```

```
/*code for voltage switching*/
*(com2+4)=0x0a; //voltage change to 1.2V
high_slope=0;
}
else
{
/***** low to high voltage change*****/
__asm__("MOV R1, #9");
__asm__("MCR p14, 0, R1, c6, c0, 0");
*(com2+4)=0x08; //voltage change to 1.5V
high_slope=1;
}

    CYG_REPORT_RETURN();
}
```