ABSTRACT

| | |
|---|---|
| Title of Thesis: | RTOS PERFORMANCE AND ENERGY CONSUMPTION ANALYSIS BASED ON AN EMBEDDED SYSTEM TESTBED |
| Degree candidate: | Tiebing Zhang |
| Degree and year: | Master of Science, 2001 |
| Thesis directed by: | Professor Bruce. L. Jacob<br>Department of Electrical and Compupter Engineering |

Embedded systems play a very important role in this information based society. The requirement of embedded systems is different from desktop systems in that processing power relative to energy expended is considered instead of raw processing power. Embedded operating systems still are not adequately able to use minimum energy to achieve the required processing power. In this paper, a full-featured embedded system test-bed (Simbed) is constructed to help analyze different kinds of real time operating systems in terms of power consumption and performance. A home-made multi-tasking scheduler and a public domain commercial RTOS are studied. Efforts were also made to utilize speed-setting and voltage scaling techniques to achieve a better trade-off between energy consumption and performance.

# RTOS PERFORMANCE AND ENERGY CONSUMPTION ANALYSIS BASED ON AN EMBEDDED SYSTEM TESTBED

by

Tiebing Zhang

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2001

## DEDICATION

To My Parents and My Wife

# ACKNOWLEDGEMENTS

First I would like to thank Dr.Jacob for his support, encouragement, and assistance. Without his help, this thesis would not have been here.

Kudos go to Brinda , Aamer and Vinodh. They have been giving me great help during my research time.

I also appreciate all the help I got from the Computer Architecture group, including professors and graduate students.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

# Introduction and Motivation

Embedded systems, more than anytime in history, are becoming the driving force of the computer and electronics industries. As desktop computers push their performance close to physical limits and provide more computing power than most applications need, the era of pursuing performance by any means is fading out, while the new era, which is the pursuit of both performance and energy saving, is rising above the horizon. Steve Leibson [17]mentioned that "it is rather apparent that the PC market is no longer willing to accept the 'performance at any price' development curve that the industry has ridden for the past two decades". The two reasons he gave are "lack of need" and "processing power delivered relative to energy expended, instead of just raw processing power".

As new semiconductor technologies progress, more and more transistors can be packed into one small chip. Today, in 2001, manufacturers such as AMD,IBM, INTEL and Motorola are mass-producing chips based on a 150nm (0.15 micron) technology. 100nm chips have already debuted. To utilize this capacity to pack a huge amount of transistors on a chip, nowadays most digital systems, e.g. processors and DSPs, are typically built around a processor "core" (i.e., a processor without peripherals and memory provided as a design primitive, subsequently called core). The off-the-shelf processor chips are adapted to their particular function by providing adequate peripherals, ROM, RAM, and embedded software. This high density integration makes putting probes directly on the pins of a chip and measuring signals more and more difficult, and as a result, it is no longer easy to understand what is going on inside the chip.

At the same time, the more and more powerful desktop computers provide us with increasing potentials of simulation. With a fairly good understanding of the architecture of a processor core and its integrated peripherals(like memory, MMU, interrupt controller, timer, etc), it is totally possible to use a piece of code to simulate the detailed activity inside a system-on-chip processor. Actually, much research has been done this way, but mostly to simulate a desktop processor and the goal was to get highest performance.

Because of our consideration of the important role that embedded systems will play in the near future, and because of our understanding that the goal of embedded systems is different from desktop systems, we decided to develop a full-featured embedded system test-bed to provide us with a good embedded system research environment. This test-bed will be a handy tool for us in our studying of embedded processor architecture, embedded operating systems , and software/hardware co-design. So the first goal of our work was to finish developing such a test-bed and make sure it was working correctly.

As mentioned before, the design goal of embedded systems usually is not just high performance. Instead, adequate performance, good response time and minimal possible energy consumption will have a balance in such systems. Based on the mentioned test-bed, we wanted to conduct an in-depth study on comparing the performance of two different real time operating systems (RTOS) and a home-made multi-tasking scheduler. So our second goal was to port two real time operating systems to our test-bed, design a method to measure the performance of them, and do a comparison.

As we all have known, energy consumption plays a very important role in embedded systems. However, it is anything but easy to come up with an energy consumption model of a specific processor. Much research work has been done in this area and several methods, on different levels, have been proposed[1, 2, 24, 7]. These papers have stated clearly the advantages of using an emulator instead of a piece of real hardware in doing research work. Simply put, it is much more flexible and much easier to work with a model, in the sense that we will be able to look into the system and see which part is most energy-thirsty and will be able to do some optimization on energy. So our third goal was to construct a relatively accurate energy model for the target processor based on the simulator.

With the construction of the energy model, we were able to do energy consumption research on our three guest operating systems and see how they performed. First, we acquired the breakdown of power consumption numbers of each operating system; after analyzing them, we decided to push our research a little bit further and do some research on how to minimize an embedded operating system's energy consumption. We chose the voltage scaling technique to be our entry point. Several studies have shown that voltage scaling is an effective way to lower the energy consumption of a system; however, research work in this area is still at the beginning stage. We would like to contribute our two cents on finding an effective way to implement the voltage scaling technique in embedded systems.

## 1.1 Goal of The Work

This research work is long-term-oriented. The ultimate goal of the work is to find an effective way to lower the energy consumption of an embedded system while keeping the performance at an acceptable level. Actually the evaluation criteria itself is part of the research work, because it is always a compromise between energy and performance. It is not very obvious to say, "This method is better than that one" unless we can come up with overall criteria that include all these factors and give an overall evaluation of the system. Therefore, coming up with overall evaluation criteria is part of our research work.

Chris Collins has done a great job in starting this research project. The starting point of this paper is that we already had a bare-bone cycle-accurate emulator and co-operative real time operating system Echidna running on it.

The contributions of this thesis includes the followings:

1. Finished developing a full-featured cycle-accurate instruction level emulator of Motorola MCORE processor (the test-bed). Also finished developing an energy consumption model of this processor and adding speed-setting and voltage scaling functions to the test-bed. The test-bed will provide most of the functions needed for future research work (including debugging method), and also provide good documentation so that even if the future research work needs to modify the test-bed or add more functions, it should be very straight-forward.

2. Finished porting one non-preemptive and one preemptive real time operating system to the test-bed. The preemptive RTOS we chose is $\mu C/OS$. We chose it because it is open source and it is widely used. Also, its internals are well-documented[16]. $\mu C/OS$ can be a representative of the group of preemptive RTOS. For the non-preemptive RTOS, we decided to make our own. It is easier

to understand and debug the RTOS if we develop it, and this also represents about 50% of the industry, who prefer to make their own RTOS (or basically a multi-tasking scheduler)[8]. We call our non-preemptive scheduler "NOS", for its simplicity. So $\mu C/OS$ and NOS are the representatives of preemptive and non-preemptive RTOS respectively. Also, they are representatives of commercial RTOS and home-made RTOS respectively.

3. Designed a method to benchmark the performance and energy consumption of these RTOSs. Because this includes the compromise of performance and power consumption, there is no one absolutely correct way to say which is better. We proposed generic metrics to evaluate these systems and hope this can be the stone that leads to the gold mine. Please refer to section 5.4.4 for details of the metrics we proposed.

4. Compared the performance of uC/OS with NOS in terms of period tasks jitter and interrupt response delay. Also Compared the power consumption of uC/OS and NOS with and without using speed-setting and voltage scaling algorithms.

5. Proposed a new algorithm about speed-setting and analyzed the results. Provided a good base for future research on voltage scaling.

## 1.2 Organization of The Paper

The following chapters of this paper are organized as following: Chapter 2 is background introduction, describing what has been done in the areas that relating to this work. Chapter 3 describes the test-bed, including a simple description of the target processor, structure and kernel of the simulator, and some basic peripherals that make it possible to run a real OS on the test-bed and make it easier to debug the OS and the applications that run above the OS. Also included in this chapter is a description of the two RTOSs we have been using in our research work. Chapter 4 mainly focuses on the performance analysis of the RTOSs after we systematically ran several benchmark programs on them. It shows the difference between pre-emptive OS and non-preemptive OS and also shows our own way to benchmark the performance of a RTOS. Chapter 5 describes the energy consumptions of NOS, including the energy consumption without any optimization method and energy consumption with voltage scaling algorithm. Chapter 6 gives a conclusion based on all research work we have done and suggests some work that can be done in the future to strengthen this research.

# Chapter 2

# Background

## 2.1 Embedded Systems

An embedded system is a very concrete concept for embedded engineers, but it is not that concrete and tangible for people out of the embedded field, even for some computer science people. Maybe one can say that, "An embedded system is any system that uses one or more microprocessors but does not belong to desktop systems", but this is still not very clear about what it really is. Here, rather than attempting to define embedded systems, we will describe their relevant properties.

First of all, embedded systems usually implement *dedicated functions* such as control of anti-blocking brakes, the instrumentation and control of an assembly line, encoding and decoding of audio or video, hand-held network traffic monitor, and so on. Thus, the function is well defined in advance, and the embedded system is installed once and for all. Once installed, little or no changes are allowed.

Also, embedded systems are usually required to be operational during the life span of the host system which may range from a few years, e.g., a low end audio component, to decades, e.g., an avionic system. *Correctness* of the design is a very important point, since embedded systems often perform safety-critical tasks in host systems such as airplanes or trucks. Hence, a malfunction might cause a major accident.

*Real time* is usually another characteristic of embedded systems. Real time means that an embedded system has to meet deadlines dictated by its environment defined by the host system. Temporal properties are relevant to system correctness. Temporal requirements may be periodic and characterized by a frequency, e.g., the sampling frequency in an audio system or a periodic observation of the state of the controlled object. Temporal requirements may also imply a maximal reaction time to given events, e.g. the maximal time to open an emergency valve in a cooling system if overheating occurs.

Real time itself has two categories: *soft real time* and *hard real time.* Soft real time means that a task is expected to finish before some deadline. However, if the system, because of some unexpected reason, does not meet the expected deadline, there will not be disastrous results, such as human beings' lives being threatened. An example of soft real time embedded system will be an MP3 player. The system is expected to decode the data stream within a certain time limit, however, even if occasionally the system does not meet the deadline, the worst result is that we hear some glitches when listening to the song. Usually this will not lead to catastrophic results. Hard real time is the opposite of soft deadline, i.e. it is not acceptable for the system to miss any deadline that it is not supposed to miss. Once the system misses the deadline, the result will be too expensive ( or too deadly) to be accepted. A good example of hard real time system is airplane control system. No exception is allowed in such systems. Every thing must operate exactly as it is designed. The possible results of such

| Embedded Systems | Desktop Systems |
|---|---|
| Dedicated Functions | Generic Functions |
| Once Shipped, Rarely Changed | Functions defined by software |
| Usually Real Time | Usually not real time |
| Cost Varies | Relatively fixed cost |
| Examples: MP3 Player, Printer | Examples: Mac, Sun workstation |

Table 2.1: Comparing characteristics of embedded systems and desktop systems

system missing its deadline are disastrous and not acceptable.

Table 2.1 is a comparison of the characteristics of embedded systems and desktop systems. Included in the table are only the major differences of these two kinds of systems.

## 2.2 Energy Consumption In Embedded Systems

Energy consumption plays an important role in embedded systems. In battery-driven systems it is critical to system design. Battery life is an obvious reason of low power design in embedded systems; however, there are many subtle reasons besides this, including reliability, performance and cost. As the frequency of the clock that drives a microprocessor rockets up, the power consumption of the chip goes dramatically up, too. For instance, the new Pentium IV has a power consumption of 55 Watt. It needs a 450g(one pound!) heat sink and fan unit, and needs 50Amps(ouch!) of current[6]. This imposes big challenges on system's reliability. Also, enormous heat dissipation will make packaging the silicon difficult. “In some applications, the device integration density is limited by thermal conditions, not by lithography, or processing.”[20].

Power dissipation in CMOS circuits arises from two different mechanisms: static power, which results from resistive paths from the power supply to ground, and dynamic power, which results from switching capacitive loads between two different voltage states. Dynamic power is dependent on frequency of circuit activity, since no power is dissipated if the node values do not change, while static power is independent of frequency of activity and exists whenever the chip is powered on. For uses where the electronics will be inactive for much of the time (most portable applications), the static power must be made very low in the inactive state.

Since static power consumption is typically much smaller than dynamic power consumption, and there is not much that can be done on the software side to reduce it[12], we will focus on dynamic power consumption from now on.

Dynamic power consumption mainly stems from charging and discharging capacitors. In charging a load capacitor C up $\Delta V$ volts, and then discharging it to its original voltage, a gate pulls C $\Delta V$ from the $V_{dd}$ supply to charge up the capacitor, and then sinks this charge to Gnd to discharge the node. So at the end of a cycle, the gate / capacitor combination has moved C $\Delta V$ of charge from $V_{dd}$ to Gnd, which uses $C\Delta V V_{dd}$ of energy and is independent of the cycle time. The dynamic power of this node is the energy power cycle, times the number of cycles it makes a second, or

$$P = C\,\Delta V\,V_{dd}\,\alpha F \tag{2.1}$$

where $\alpha$ is the number of times this node cycles each clock cycle and is usually called the activity ratio. The dynamic power for the whole chip is the sum of equation 2.1 over all the nodes in the circuit[12].

From equation 2.1 it is clear what can be done to reduce the dynamic power consumption of a system. We can either reduce the capacitance being switched, the voltage swing, the power supply voltage, the activity ratio, or the operating frequency. For a specific chip, the voltage swing $\Delta V$ is usually proportional to $V_{dd}$, so equation 2.1 will become the following:

$$P = CkV_{dd}^2\alpha F \tag{2.2}$$

where $k$ is the ratio of $\Delta V$ and $V_{dd}$.

## 2.3 Basic Design Methods of Low Power Systems

### 2.3.1 Reducing Supply Voltage

The fact that the power consumption is proportional to the square of supply voltage is good news: great power savings are realized for small reductions in supply voltage.

Compare a system operating at 5V supply versus 3.3V. Reducing the voltage supply from 5 to 3.3V reduces the dynamic power consumption to $(3.3/5)^2$ of the original, which is a power saving of 57%. Figure 2.1 shows this graphical square relationship.

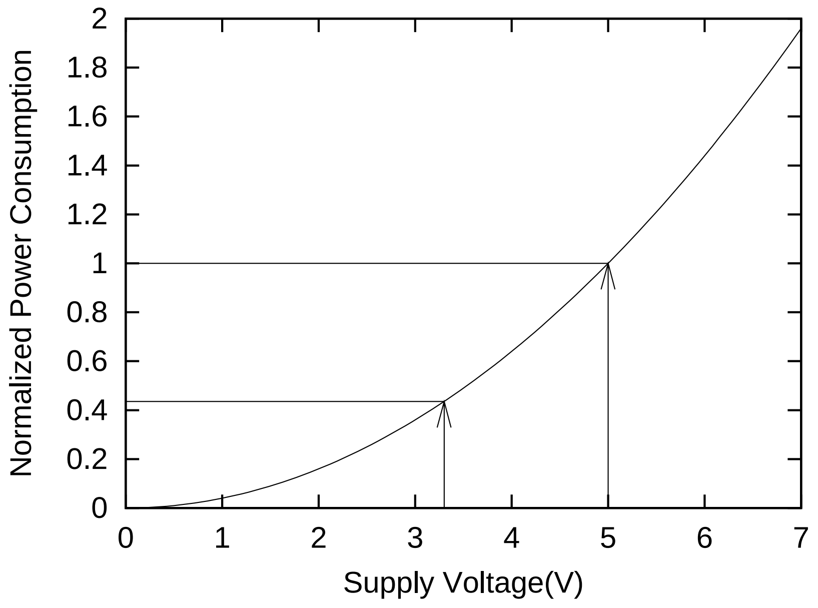


Figure 2.1: Reducing the supply voltage from 5.0V to 3.3V results in a 57% power saving

### 2.3.2 Fast Clocks

Power consumption is proportional to operating frequency, which is the $\alpha F$ in Equation 2.2. As the operating frequency goes to zero, the dynamic portion of the power consumption also approaches zero. This situation leaves only the static power consumption, which typically is in the micro-watt range for CMOS ICs.

Because power consumption depends heavily on clock speed, a guideline in designing low-power embedded systems is to choose a processor speed as fast as the application needs, and no faster. Running the clock at a frequency higher than necessary wastes valuable energy.

Also, substantial power savings can be realized from intelligently managing the CPU clock speed. If the CPU clock is fixed at a blazingly high speed to accommodate a compute-intensive task(e.g., data processing), then considerable power is wasted when the system performs a less CPU-intensive task(e.g., acquiring data). A solution to this problem is: the scalable clock. Control the frequency of the processor's clock (either by software or by hardware) to match the processing speed of the task being performed. Thus, for a minimal amount of additional hardware and software, the programmer can lower the system power consumption dramatically by scaling the clock based on the computational load. Some processors, like Intel's Mobile Pentium III and Transmeta's Crusoe, have their own on-chip scalable clocks. For these systems, scaling the clock can be done by some special instructions. For those processors that do not have on-chip scalable clocks, an external clock divider can be designed to accommodate such need. The circuit should not output glitches when the frequency is switched, as this causes improper operation of the microprocessor. Also, if the application uses internal timers or counters driven by the scaled clock for precise timing, special attention should be paid when writing such code.

### 2.3.3 Slowing The Clock

In many applications, the clock needs to be stopped completely, and most microprocessors have this feature. Modes that turn off the clock (referred to as sleep, doze, snooze, shutdown or halt) are usually invoked by writing to a special register. Normal operation is restored by a stimulus event such as an interrupt (external, or internal if on-board peripherals are permitted to operate) or reset.

Brian Kurkoski [15] categorized three possible zero-frequency clocking modes based on their technical differences. In Mode 1, the oscillator continues to operate, but the core is not clocked. In mode 2, the oscillator is off, but the processor is still powered. And in mode 3, power to the processor is removed completely.

The advantage of mode 1 is that it can respond quickly to a stimulus event, even though current consumption is high. This is because typically in this mode the clock itself is not stopped, but the supply is gated so as not to go into the processor core. When needed, this gate can be immediately opened. In mode 2, current consumption is reduced to the quiescent current of the microprocessor and the components, but it takes much longer to restart the processor. It is usually controlled by the reset generator, which has a 50 to 200 ms pulse. For example, the TPS3705-50 processor supervisory circuits with power-fail from Texas Instruments has a fixed delay time of 200ms. That is a generous amount of time for the oscillator to restart, but a long time for a real-time event. In mode 3, the quiescent current of the sleeping components is eliminated entirely by disconnecting power to the system via a switch, such as a p-channel FET. However, some additional circuitry is required to switch power.

Modes 1 and 2 rely on the features of the microprocessor, but the power saving mode 3 can be implemented using any microprocessor and it affects the entire system. In modes 1 and 2, the restart depends on the microprocessor. In mode 3, the restart is implemented in hardware.

As an alternative to putting the unit into sleep mode 1, consider slowing the clock to the tens to hundreds of kHz range. Power consumption at these clock frequencies may be comparable to sleep mode operation and the problems of oscillator restart time are avoided. While in slow speed mode, the microprocessor performs a simple sampling task, like monitoring the keypad for user input or sampling the real-time clock to wake the system at a specified time.

For a battery powered system, low power design will provide another plus: lengthening the battery life. Tajana Simunic et al. [22] provides a battery model

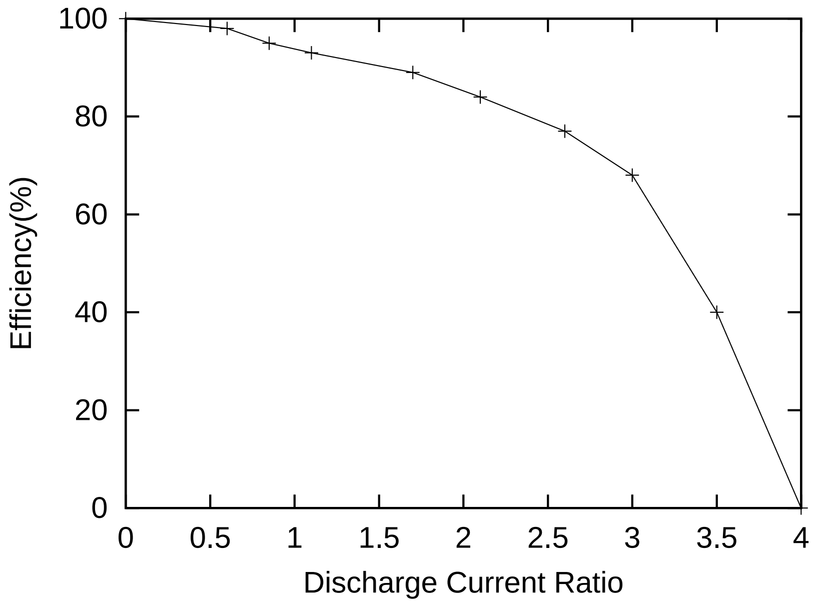


Figure 2.2: Battery efficiency with different discharging ratio

that indicates when drawing current is greater than the rated current from the specification of the battery, the battery's capacity will be lower. Figure 2.2 shows that when the discharging ratio(the ratio of actual discharging current to battery rated discharging current) is equal to 2, the battery's capacity will become only 80% of its original capacity. An example that was given in the paper was if a battery has a rated current of 100mA at 1V with a capacity of 100mA-Hour, i.e. if drawing current less than or equal to 100mA, the battery's life is 1 hour. However, if the drawing current is 300mA, the battery life will be 12 minutes instead of 20 minutes. Thus, with low power design the battery will run longer.

## 2.4 Power Modeling Tools

Now that we know power consumption is a very important issue in embedded systems, we want to find ways to lower the power consumption of such systems.

To do so, we must first find a way to tell how much power the target system is consuming. The obvious way is to hook up a voltage meter and an ammeter to measure it. Surely this will tell us something, however, once we are able to do this, we have almost reached the final step of a product and we can do little to improve. Most times we want to know beforehand how much power the system is going to consume, and if it is too much, we will need to find a way (either software or hardware) to reduce the power consumption. Also, when we have some ideas that might work, we need something to tell us that they will work on the real hardware. This brings us the well-known powerful tool: Emulator. By running our software on the Emulator, the Emulator will tell us how much power the system consumes.

Power simulation has been used in VLSI design for a long time. However, until a few years ago, most of the power simulation is only conducted after the design of the chip and only for packaging consideration. As more and more attention is drawn to low power embedded chips and systems, EDA tool developers begin to put more effort on power consumption considerations of other design phases. Now there are power estimation tools at most levels of design. The following is a brief summary of the major tools that are available at this time.

### 2.4.1 Transistor Level Tools

Tools at this level are generally very accurate and mature, and fall into two general categories: the circuit analysis tool, like SPICE and its many variants, and the switch analysis tools like PowerMill(Epic), ADM(Avanti/Anagram) and Lsim Power Analyst(Mentor).

The primary advantages of transistor level tools are accuracy, within a few percent of silicon, and well-accepted abstraction - most IC designers understand transistor level analysis and rely upon it. However, these tools have significant issue in their applicability to lower power design at higher levels; capacity and run time characteristics limit their use to small circuits, or very limited depths of simulation vectors for larger circuits.

### 2.4.2 Logic Level Tools

Numerous logic level power analysis tools are currently available from a number of vendors: DesignPower and PowerGate(Synopsys), WattWatcher/Gate(Sente), PowerSim(System Sciences), POET(Viewlogic) and QuickPower(Mentor) among others. Each of these tools operates on a gate level net-list, such as Verilog, and assumes the availability of a gate level power library.

Compared to the transistor level tools, gate level tools trade off accuracy for significant improvements in run time and capacity. For example, gate level power simulations are generally claimed to be within 10 to 15% of the accuracy of switch level tools, but run at least an order of magnitude faster.

### 2.4.3 Architecture Level Tools

The architectural level, or RT level, is the design entry point for most digital designs today. The design decisions made at this level can have a dramatic impact on the design's overall power characteristics. Thus the use of tools at this abstraction level is of the utmost importance.

WattWatcher/Architect(Sente) is the first commercially available tool operating at the architectural level. WattWatcher/Architect reads Verilog and VHDL RT level description. It utilizes a conventional gate-level library and simulation data from an RTL simulation to compute a power estimate for the entire chip, including all the peripherals.

There are several research works going on in academics at this level. Vivek Tiwari et al.[24] have done a great job in starting a new method of doing power estimation(please refer to section 3.4 for details). While providing reasonable accuracy (20%), this experiment-based method is straight forward to implement, and is also easy to glue with a simulator for doing performance analysis. N. Vijaykrishnan et al.[26] has done research work on coming up with a simulator that is based on a RTL library. This method combines the estimator with a simple-scalar simulator. Also it simulates the memory and bus power consumption. Another relevant architectural level power estimator is from David Brooks and Vivek Tiwari, et al. [1]. This estimator is based on a suite of parameterizable power models for different hardware structures and on per-cycle resource usage counts generated through cycle-level simulation.

Similar to the comparison between transistor level tools and gate level tools, architectural power estimation trades off accuracy for even larger improvements in run time and capacity. Reasonable accuracy (WattWatcher/Architect claims to be within 20% to 25% of silicon)is maintained while execution speed and capacity are significantly enhanced, thereby enabling design space exploration which would be too slow or tedious to do efficiently at the gate level.

### 2.4.4 Behavior Level Tools

Generally the least explored of the abstraction layers in terms of power, the behavior level is currently unsupported by commercial tools. Being an active area of academic research, power estimation at this level is typically accomplished by

| Level | Accuracy | Speed |
|---|---|---|
| Transistor - Switch | 5-10% | $10^7$ v-t/min |
| Logic - Gate | 10-15% | $10^8$ v-t /min |
| Architecture - RTL | 20-25% | $10^9$ v-t /min |
| Behavior(speadsheet) | 50-100% | minutes |

Table 2.2: Power estimation tools comparison by abstraction level

using spread sheets. These spread sheets are quick and easy to use, but suffer from a very large variance in accuracy.

Table 2.2 shows some of the key characteristics of low power design tools, by abstraction level.

## 2.5 Speed Setting And Voltage Scaling

From the above discuss we know that power supply voltage is a very critical factor in determining the whole system's power consumption. That is why when the power consumption of Intel's Pentium IV exceeds 50W, Intel introduces its SpeedStep technology into the Mobile Pentium III microprocessors, in order to provide a low power chip. That is also why Transmeta(CA) can be a very successful company in its leading-edge Crusoe low power microprocessors where the similar techniques were used to lower the supply voltage of the chip.

Microprocessor's maximum clock speed has some relationship with the supply voltage. Reducing supply voltage results in increased circuit delay, and to a good

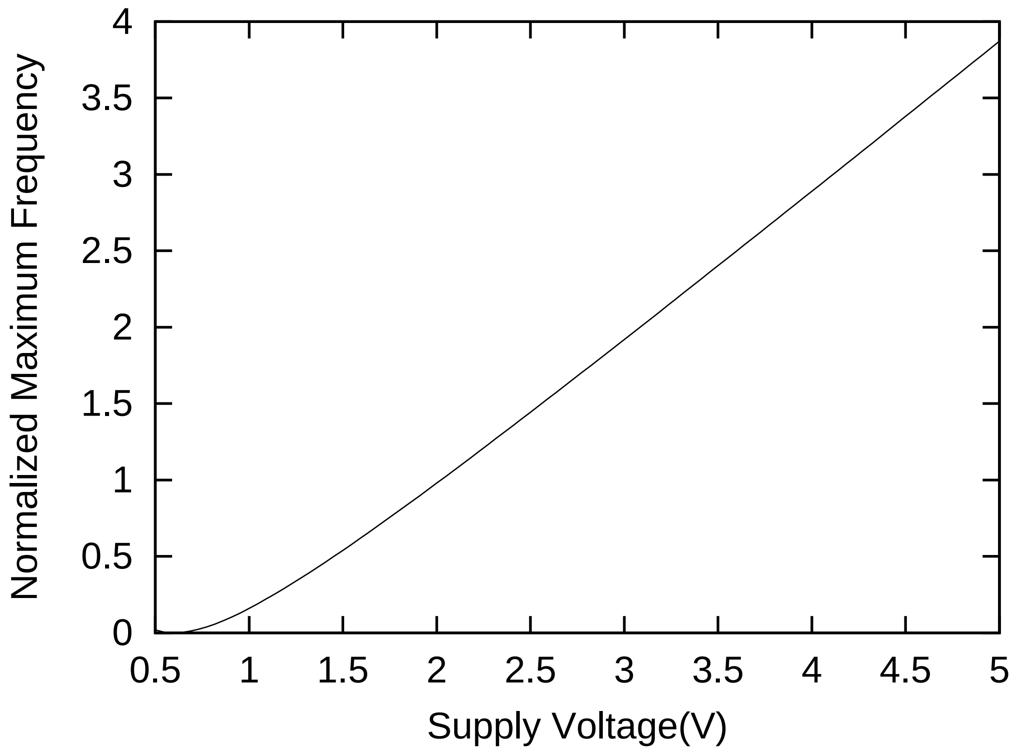


Figure 2.3: As supply voltage decreases, circuit delay will increase and maximum processor speed will decrease

accuracy, the circuit delay is given by

$$t_{delay} = k * (V_{dd})/(V_{dd} - V_t)^2 \tag{2.3}$$

where $V_t$ is the threshold voltage, and k is a constant. With $V_t$ being a typical value of 0.6V for CMOS circuits, we have the relationship indicated in figure 2.3. The Y axis is not absolute frequency value, instead, it is a value that is proportional to the frequency. This graph is to show when the supply voltage drops, the maximum processor speed will drop.

In order to achieve both low power and high performance, we need the processor to run at low supply voltage, i.e. slow speed clock, when the processor's load is low, and to run at high supply voltage,i.e. high speed clock, when the processor's load is heavy. This inspired the *voltage scaling* technology.

Voltage scaling means that a processor can runs at different level of supply voltage and clock speed. As for the optimal number of levels, there has not been a clear research result. Intel is using 2 levels in its Mobile Pentium processor, while Transmeta is using 32 steps in its Crusoe processor. Certain instructions can cause the processor to jump from one level to another level. The processor needs a certain amount of transition time before it can start running at the new level. For Crusoe processor, the transition period length will increase as the number of transition levels increase.

Mark Weiser et al. [27] were one of the few early groups that contributed to voltage scaling algorithms. They used task-driven simulation to gather the power consumption data where fine grain control of CPU clock speed was imposed. The PAST algorithm they proposed also serves as the reference for much of the later research work. The PAST algorithm basically predicts the processor load by looking back at a fixed time period. If the prediction is too aggressive and the work is not finished in the prediction period, the rest of the work will be added to the next prediction computation. This is a relatively simple algorithm, and it gives out considerably good results considering the simplicity of the algorithm.

Dirk Grunwald et al. [10] implemented the PAST and other algorithms on an experimental pocket computer that runs a complete, functional multitasking operating system(a version of Linux 2.0.30). They used these algorithms to adjust the processor speed to reduce the power used by the processor. Their results show that the algorithms tested consistently failed to achieve their goal of saving power while not causing user applications to change their interactive behavior.

Kinshuk Govil et al. [9] have proposed several variations of the PAST algorithm and compared them, including PAST, FLAT,LONG_SHORT, CYCLE,PEAK etc. They tried to find a pattern of the past power consumption and predict the future load based on the pattern. Their result is that the PEAK algorithm, which is a specialized version of pattern based algorithm, has the strongest performance.

Inki Hong et al. [11] developed a design methodology for the low power core-based real-time system-on-chip based on dynamically variable voltage hardware. They addressed the issue of how to develop effective scheduling techniques that treat voltage as a variable to be determined, in addition to the conventional task scheduling. They also addressed the selection of the processor core and the determination of the instruction and data cache size and configuration. The highlight of their work is that their proposed approach, which is a non-preemptive scheduling heuristic way, achieved results that are close to optimal for many test cases.

## 2.6 Real Time Operating System

Real Time Operating Systems (RTOS) are widely used in the embedded system industry. In this section, the requirement of an RTOS is given, and some concepts in real time systems are discussed.

### 2.6.1 Requirement of Real Time Operating Systems

Liu [18] proposed that a good RTOS at least offer the following things: First an RTOS needs to offer a method to schedule tasks. The second one is timing maintenance, i.e. the RTOS needs to be responsible for both providing and maintaining an accurate timing method. The third one is to offer user tasks the ability to perform system calls, i.e. the RTOS offers facilities to perform certain tasks that the user would normally have to program himself. The last thing that an RTOS needs to offer is a mechanism for handling interrupts efficiently, in a timely manner, and with an upper bound on the time it takes to service those interrupts.

### 2.6.2 Several Concepts of Real Time Systems

**Task** A task, also called a thread, is a simple program that thinks it has the CPU all to itself. Note that in desktop systems task and thread are totally different concepts, but in embedded system, usually they can be used interchangeably. The design process for a real-time application involves splitting the work to be done into tasks responsible for a portion of the problem. Each task is assigned a priority, its own set of CPU registers, and its own stack area.

**Multitasking** Multitasking is the process of scheduling and switching the CPU between several tasks. Multitasking maximizes the utilization of the CPU and also provides for modular construction of applications. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherent in real-time applications. Because the complex applications are usually divided into several tasks, they are typically easier to design and maintain using multitasking.

**Context Switch** When a multitasking kernel decides to run a different task, it simply saves the current task's context(a couple of CPU registers) in the current task's context storage area, typically the stack in embedded operating systems like $\mu C/OS$. Once this operation is performed, the new task's context is restored from its storage area then resumes execution of the new task's code. Context switch adds overhead to the application.

**Kernel** The kernel is the part of a multitasking system responsible for the management of tasks and communication between tasks. The fundamental service provided by the kernel is context switching, but most kernels provide some basic facilities, like resource share protection, inter-process communication, interrupt handling etc. The use of a real-time kernel usually simplifies the system design, but the price is the ROM and RAM consumed by the kernel. Also the kernel will consume CPU time.

**Priority based Scheduling** Each task in the system is assigned a priority based on the importance. With priority-based scheduling, the control of the CPU is always given to the highest priority task ready to run. Priority based scheduling includes static priority and dynamic priority scheduling. Static priority scheduling is that the priority of each task does not change during the application's execution. Each task is thus given a fixed priority at compile time. All the tasks and their timing constraints are known at compile time in a system where priorities are static. Dynamic priority scheduling is that the priority of tasks can be changed during the application's execution; each task can change its priority at run time. This is a desirable feature to have in a real-time kernel to avoid priority inversions. Please refer to [16] for details on priority inversion.

**Non-preemptive and Preemptive Scheduling** Non-preemptive scheduling requires that each task does something to explicitly give up control of the CPU. This is also called cooperative multitasking. Tasks cooperate with each other to share the CPU. A new higher priority task will gain control of the CPU only when the current task gives up the CPU. The most important drawback of non-preemptive systems is responsiveness. With preemptive scheduling, the highest priority task ready to run is always given control of the CPU. When a higher priority task is ready to run, the lower priority task is preempted(suspended) and the higher priority task is immediately given control of the CPU.

# Chapter 3

# Test-bed Construction

Part of this project's goal is to construct a solid embedded system test-bed for current and future research. The test-bed should have the following features:

Accuracy – the CPU emulator inside the test-bed should be a cycle-accurate emulator.

Real – Any binary programs, including operating system and applications that run on the real hardware should be able to run on the simulator without any need to change them.

Full-functional – The test-bed should support sophisticated operating system to run on it.

Ability of logging – The test-bed should provide methods to reflect all kinds of statistics and other information that is going on inside the chip while the code is running.

Flexibility – New modules, like power modeling modules, or Speed setting modules, should be able to be added easily. If some part of the chip is needed to be excluded from the experiment, it should easily be done.

Portability – The code should be written in ANSI C.

Maintainability – The code should have good format and documentation, for future modification.

## 3.1 Brief History

This part is a brief history of the test-bed evolution. The research is not described here.

The whole project started from scratch. Chris Collins, Eric Fiterman and I started this in Feb. of 2000. At the end of May 2000, the test-bed ran the first Real Time Operating System - NOS, which is our home-made operating system. Also, we finished verifying the accuracy of the emulator inside the test-bed by using the real MCORE evaluation board and its developing system. By the end of May, we got another RTOS – Echidna, which is a cooperative RTOS, running on the simulator. At that time, the test-bed was in a working-but-messy state. In June of 2000, I started working to make this a serious embedded system test-bed. First, part of the emulator and the object file downloading system were rewritten to make it clean. Then during the summer, Katie Baynes and Christine Smit joined into the group. We investigated several power modeling techniques and added a power modeling function to the test-bed. Then a screen terminal emulator and a display library were added by me so that the application can dump data onto the screen. This dramatically improved the chance to port real-life operating systems to the simulator. Also this made debugging applications running on the top of the test-bed much easier. Then I ported a commercial RTOS $\mu C/OS$ to the test-bed. With both cooperative operating system and pre-emptive operating system running on the test-bed, we began research on the performance of these operating systems. This resulted in a systematic design of logging information(please see section 3.5 for more).

Then I added speed-setting and voltage scaling modules to the test-bed. The functionality of the test-bed was now different from the original physical processor functionality. I added the speed-setting and voltage scaling capability to the chip, according to the data-sheet of other processors. This gave us a great opportunity to look into the area of voltage scaling optimization.

## 3.2 The Cycle-accurate Emulator

### 3.2.1 Why MCORE?

MCORE was a new design of four years work targeting the area of low-power embedded microprocessors. The current existing architectures are not suitable for low-power applications due to their inefficiency in code density, memory bandwidth requirements, and architectural and implementation complexity[21]. MCORE architecture was designed from the ground up to achieve the lowest milliwatts per MHz. The MCORE instruction set was optimized using benchmarks common to embedded applications coupled with benchmarks targeted specifically for portable applications. However, due to business reason, Motorola is stopping supporting the MCORE architecture in favor of StrongARM.

### 3.2.2 The Basic Structure Of MCORE Processor

The Motorola MCORE architecture is a 32-bit Load/Store architecture with a fixed 16-bit instruction length and 32-bit data length. It has a 16 entry 32-bit general register file, a 16 entry 32 bit alternate register file to allow fast interrupt support, and a 13 entry control register file accessible only by the supervisor mode. Its execution pipeline's four stages are completely hidden from the application software. Most instructions execute in a single cycle with two cycle execution for loads, stores, and taken branches and jumps. The address space is byte, halfword, and word addressable, and allows both fast and normal interrupts, allowing those interrupts to be either vectored or auto-vectored interrupts.

The pipeline for the MCORE consists of four states: instruction fetch, instruction decode/register file read, execute, and write-back. All of these stages operate simultaneously, making single cycle instructions possible. All sixteen general purpose registers can be used as source operands and instruction results.

The MCORE programming model is defined for two privilege modes: supervisor and user mode. There are certain operations not available in user mode. User programs can only access registers in the general register file, whereas supervisor mode programs can access all registers, using control registers to perform supervisory functions.

### 3.2.3 The Emulator And Its Validation

The emulator simulates the pipeline of the MCORE processor. The basic structure is that at every cycle all the pipeline stages are simulated by the code, and the register status is kept in variables. The functionality of each instruction is finished by calling the corresponding functions in C code during Execution stage. This is done by using a pointer to function name, which eliminates all the overhead if "switch" clause is used. Figure 3.1 is a simple flow chart of the emulator itself.

Exceptions are checked every cycle. If there is an exception, then a series of steps are triggered, the relevant registers are copied to some shadow registers and the pipeline is partially flushed according to the MCORE hardware specification.

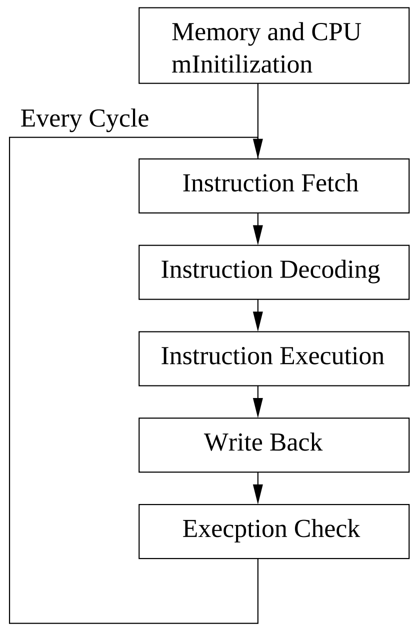


Figure 3.1: The flow chart of the emulator

After a certain number of cycles, the CPU starts fetching the instructions of the exception handler.

The validation of the emulator was done by two steps. The first step was to run a set of benchmark applications both on the hardware and on the emulator. The register values were compared after the application was finished. This was mainly for validation of the functionality of the instruction set. The second step was to run an operating system – Echidna[23] on the emulator and on the hardware. This was mainly for the testing of interrupts and exceptions. Also during this step the accuracy of the emulator was determined. The cycle error of the emulator compared to the real hardware was about 0.01%. So this emulator is a cycle-accurate emulator.

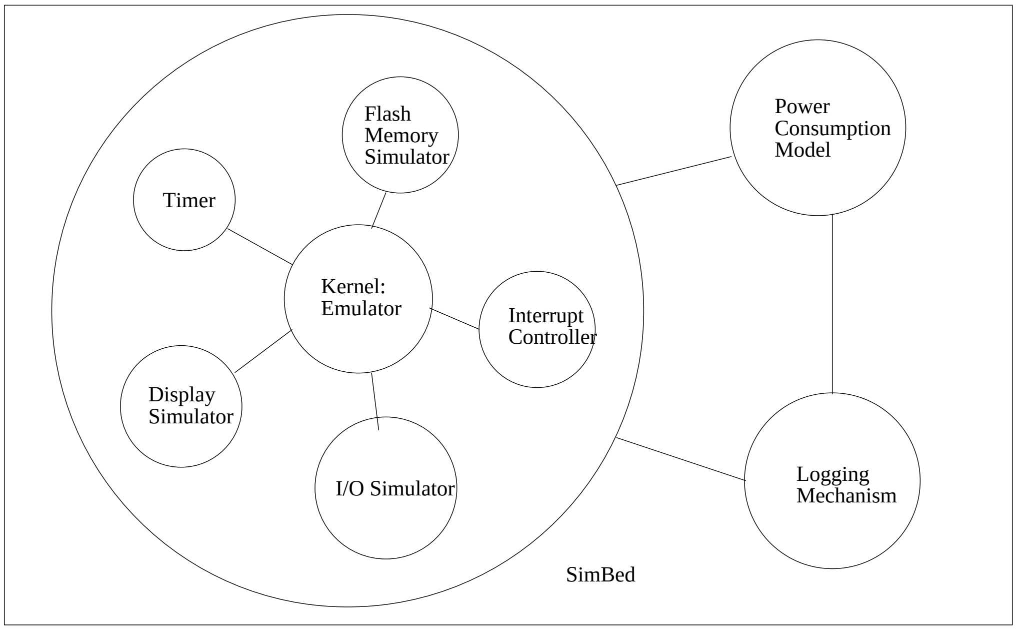


Figure 3.2: The structure and elements of the whole test-bed

## 3.3 Advanced Feature Of The Test-bed

To make the emulator a truly embedded system test-bed, which means that serious operating systems should be able to run on it, and useful data can be grabbed out of it, several parts need to be added. Following is the description of the features, including timer, file downloading tools, display emulator and I/O simulation. All these peripherals were added according the MMC2001 microprocessor, which is a low-end product based on MCORE architecture. Figure 3.2 is the organization of the whole test-bed.

### 3.3.1 Interrupt Controller

Following the MMC2001 processor specification, an interrupt controller was implemented in the test-bed. All the external interrupts are controlled by this controller. Each interrupt source has its own mask bit and source bit. Normal interrupts and fast interrupts are implemented. This interrupt controller will be used by the PIT timer and other I/O devices. Please refer to MMC2001 reference manual [13] for details.

### 3.3.2 Timer

A counting timer is an essential unit for most of the operating systems. Task scheduling, context switching and many other system functions solely depend on the functionality of a timer. Usually there are several timers in a microprocessor. One of them is used by the operating system as system time tick. In MMC2001, PIT(interval timer) timer is used for this purpose. The interval timer is a 16-bit "set-and-forget" timer that provides precise interrupts at regular intervals with minimal processor intervention. The timer can either count down from the value written in the modulus latch, or it can be a free-running down-counter. This PIT timer is implemented in the test-bed.

### 3.3.3 Downloading Tools

With a MCORE emulator that can execute MCORE instructions and a timer that can do timeout, running code on the emulator is possible. However, the GNU compiler only generates ELF format object files, which is a fairly complicated format for execution and linking. We need a tool to extract all the executable code from the ELF file and put it in the right place so that our emulator can run them. Also, on real hardware, when the system power cycles, it usually gets the code from FLASH memory, which is one kind of non-volatile memory, and then either copies the code to RAM or just executes directly out of FLASH memory.

The FLASH memory method is implemented in this test-bed. A big all-0 file,

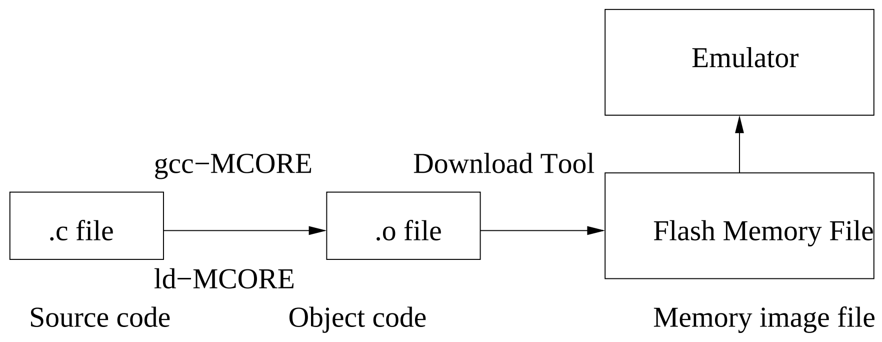


Figure 3.3: The procedure from writing source code, compiling and linking, downloading to the virtual flash memory, to running it on the test-bed

2MBytes for now, is dedicated as the flash memory. The flash memory address is set to the address where the first instruction after processor power-up is fetched. Every time the test-bed starts, which acts like the power-cycling of real hardware system, the emulator fetches the code from the FLASH memory and executes out of it. Also, after power up, this FLASH memory acts like ROM, instead of normal FLASH, which means no write is allowed to this part of the memory. This helps the user debug the system when the code is manipulating the FLASH while it is not supposed to do so. In order for the emulator to execute the code, all the user needs to do is to use the provided tool to download the code to the FLASH memory and start the emulator.

A separate C program, the download tool, is written to take the ELF file, which is generated by the target compiler, as the input and write the executable code to the right place in the FLASH memory. Basically, it parses the head of the ELF file and finds the executable segments and their memory address, and then output them to the FLASH memory file image. Figure 3.3 is the data flow from writing a C source code, to downloading it to the virtual flash memory, to running it on the test-bed. Note that in reality most of the RTOSs and applications are compiled together and then are downloaded to the memory.

### 3.3.4 Display Emulator

With a bare-bones CPU and the downloading tools, code is able to run on the testbed. However, we have not yet found a way to see the result of our program. The only way we have to inspect the execution of the code so far is to put the emulator into debug mode so that it outputs some information, such as registers values, at every cycle. The information is too much, and we need to spend a lot of effort finding out what we need. We need some kind of display method so that we can write some code in our application program to say "I am here!". This will greatly expedite the debugging of applications and ported operating systems. Note that printf is still one of most powerful debugging tools for programmers.

A virtual screen is implemented in the test-bed. The idea is to allocate some memory out of the main memory of the processor as display memory, like an old x86 with DOS. With direct writing to the display memory, one can output whatever she wants onto the screen. Ncurses library and pthread library in Linux were used to develop the virtual screen. Basically after the emulator starts running, a new display thread is created and running in parallel with the main emulator program. The only function of the display thread is to read the display memory and dump the data to the host machine screen as fast as possible. The use of ncurses library makes the display very stable, without any feeling of flash.

It is not comfortable enough for a programmer to use a primitive display method like what we have so far. It will be awkward since the user needs to find out where the display memory is and find out what value to write there in order to display a character. A basic library is needed. In order to prevent the application code from getting too big, the standard IO library is not supported. Instead, a series of concise but useful display functions are home-made by the author to support display, including functions to write a character, string, or integer to the screen, also including cursor manipulation functions, like moving the cursor to a specified place, and getting the cursor's position, etc. With these powerful but very small foot-print functions, user application can display almost anything on the screen.

### 3.3.5 I/O Simulation

Most embedded applications involve more or less input/output operation. Typical applications are GSM encoding/decoding algorithm, MPEG algorithm etc. To be able to run such applications on the test-bed, we need to include the input/output capability into the test-bed.

Most I/Os have a buffer and the physical interface. While transmitting data, the buffer will temporarily hold the data while the physical interface transmits them out synchronously or asynchronously. While receiving, similarly, the buffer will serve as a temporary place for the incoming data to stay before the upper layer driver or application retrieves the data.

Considering this generic structure of I/O, I/O simulation is implemented in the simulator in this way. For simplicity, the buffer size is set to the size of the port size. In other words, there is literally no buffer space for transmitting or receiving more than once. While receiving, the test-bed reads the data from a file in the host system which contains all the data. Every read size is the port size of the input. After the read is done, an interrupt is generated to tell the system the arrival of the data. The input can operate at the specified speed. If the data in the buffer is not retrieved by the software when the next data arrive, it simply overwrites the previous data. While transmitting, every time the test-bed finds that there is new data in the transmitting buffer, it delivers it immediately. So the transmission is operating at ideal state, meaning that it can transmit as fast as the processor clock speed. The transmitted data is stored to a local file in the host system.

With this implementation, some classical embedded system applications can be ported to run over the test-bed, including the GSM algorithm and MPEG algorithm.

## 3.4 Power Consumption Model

Unlike the instruction set simulation, which can be done simply by using some simple computation statements in the function, there is no obvious way to tell the power and energy consumed by the system. Instead, in order to get these numbers, we need to find out all the relevant factors and construct a mathematical model which will approximately reflect the power and energy consumption of the system.

Considering what we discussed in the previous chapter about the different power models that can be used in doing this estimation, basically, for an instruction level emulator, there are two ways to do this. The first way is to construct a library of the power consumption of all the modules in the processor. This library can be constructed by using some low level simulation tools, like SPICE, which can give precise power consumption numbers for a small circuit. After running simulation for all the modules in the processor, the library is ready. Then it is decided which modules will be involved for every single instruction's execution.

Then when a piece of code is executed, all the modules' activity will be in known state. With some calculation, the power consumption will be found.

The second way to add the power estimation function to an instruction level emulator was first published by a Princeton group [24]. Instead of doing simulation, this method is based on experiment data. The power consumption of each single instruction is measured by using an infinite loop with only this instruction inside. Because it is an infinite loop, a jump instruction has to be used. In order to minimize the influence of this instruction, several hundreds of the testing instruction are included inside the loop. The power number will be the base power consumption number of this specific instruction. This number multiplied by the execution time of the instruction will serve as the basic energy consumption of the instruction.

When a piece of code is running, all the instructions energy consumption numbers are simply added together to estimate the total energy consumption. However, according to the paper [1], this number will always be smaller than the real measured number. One explanation is that during the single instruction test, the state of all the modules inside the processor will not change as much as when the next instruction is different from the previous one. This extra power consumption is called inter-instruction overhead, which is different for different pair of instructions, and this accounts for a big part of the processor's overall power consumption. Measuring all the instruction pairs overhead power consumption is nearly impossible due to large number of pairs. It was found that this number stays at a similar level for most instruction pairs. Therefore, a simple substitute method is to add a constant value to all the executed instruction pairs to compensate this overhead value. With this added, the simulation power consumption

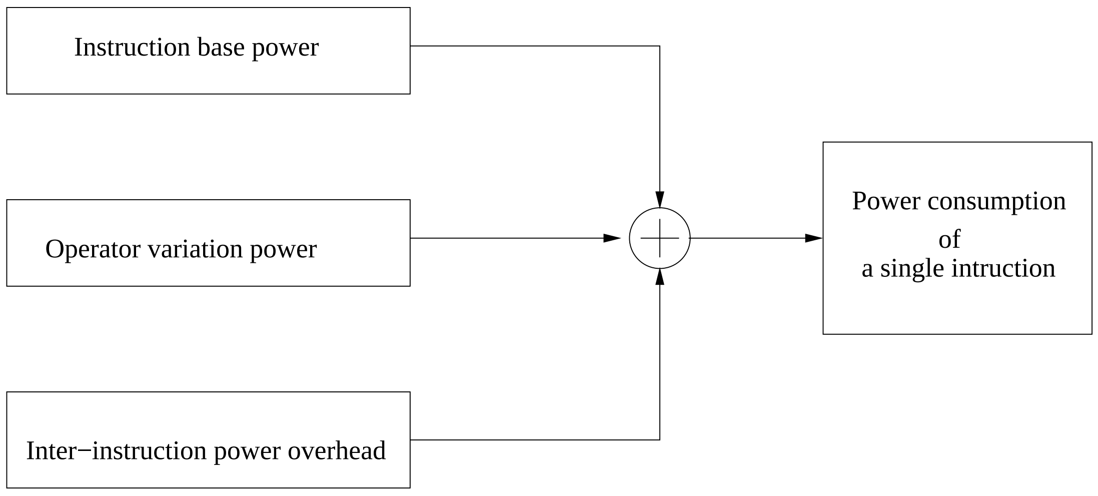

Instruction base power
Operator variation power
Inter−instruction power overhead
Power consumption of a single intruction

Figure 3.4: The power model for a single instruction

number is close to the measured number, and the error is within the acceptable region.

Another factor that influence the accuracy of this method is the changing of the operators of each instruction. Moving an all zero constant and moving an all one constant will result in different circuit state changes, and therefore different power consumptions. However, because the parameter of an instruction is random, we can use an average number to represent this fluctuation. Test results show that this method is good enough.

Validation of the power model is not an easy job, especially since we do not have a decent energy measurement tool other than a simple digital multi-meter. The way we validate our power model is to use a small program and let the program run in an infinite loop, and then use the digital multi-meter to measure the current that the board consumes. Because the multi-meter gives out an average current value and the program only needs several millisecond to run each iteration, the reading number will give out the average current consumption.

With a simple multiplication with the power supply voltage, the average power consumption is found. After several such tests and comparing the results with the simulation results, the error is within 15%. This is acceptable error considering that this is an architecture level tool and the primitive tools we used to measure the power consumption.

## 3.5 Logging

One of the most significant advantages of emulators over real hardware is that the emulator can expose much more information about what is going on inside the processor than a piece of a real hardware. The exposure of all these information is what we call logging.

Logging is different from debugging information. While both of these dump data about the inside emulator so that the user knows what is happening, debugging information is mainly used at the stage of emulator developing to make sure it is working correctly. After knowing the correctness of the emulator, most of this information is not very useful to the end user, i.e. a researcher that is doing an experiment on the emulator. On the other hand, logging tells what the end user cares about. It dumps raw data directly from the emulator, or it does some simple calculations and gives out more relevent data. The logging information can be tailored by the end users to include what they want to know and exclude what they do not want.

The logging information will be output to a file in the host file system in the specified format, so that after the simulation the user can do further processing to these data by other programs.

A very useful way to monitor the system is to do bus monitoring. In the

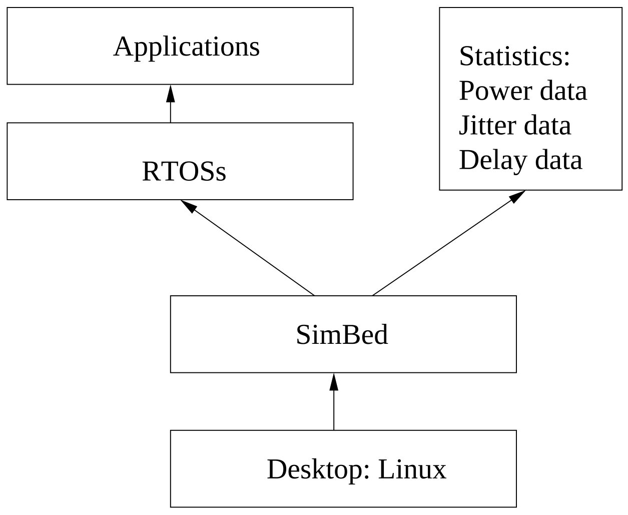


Figure 3.5: Overall structure of the test-bed

research, energy consumption of each part of the $\mu C/OS$ is needed. However, because $\mu C/OS$ is a preemptive operating system, timer tick interrupts and other interrupts can happen any time and anywhere in the code, Therefore, it is not possible to use flags to indicate which part the program is in, because the program can jump to another part right after the flag is set. In this case, memory bus monitoring will be very useful. By marking the beginning memory address and ending memory address of each section, and then monitoring the instruction memory bus, the simulator will know precisely where the program is and thus log the correct information. This is very difficult to do when running $\mu C/OS$ on a piece of real hardware, even with sophisticated instruments.

The logging feature in the current test-bed mainly output the performance relevant data and energy consumption relevant data, considering our research focuses. The details will be discussed in the following chapters. Figure 3.5 is the overall structure of the test-bed and the position of the logging module in it.

## 3.6 Running Real Time Operating Systems

We have successfully run three real time operating system on the test-bed: NOS, Echidna and $\mu C/OS$. Since Echida is similar to NOS in many ways, this paper does not discuss the Echida RTOS. Please refer to [23] for details of Echidan RTOS. Following is a detailed description of NOS and $\mu C/OS$.

### 3.6.1 NOS

NOS represents the type of "roll-your-own" RTOS often produced in the embedded systems industry. It was designed in-house and is based entirely on descriptions of home-grown embedded system software given by practicing engineers in the embedded-systems industry [8]. NOS is a fixed-priority multi-rate executive for periodic tasks [5] and handles interrupt-driven stimuli via masking interrupts and polling the interrupt status registers when idle. Its main control loop is shown in Figure 3.6. NOS's callout queue is taken from the callout table in UNIX [19]; events to happen in the future are placed in the queue keyed by the time at which they are expected to execute, and the delta field in the event structure represents the time difference between the event in question and the one before it in the queue. The delta field of the first event represents the invocation time relative to now. If the value is negative, the deadline for the first task (and perhaps following tasks as well) has been missed; if the value is zero, it is time to execute the first task; if the value is positive, the first event is to happen at some point in the future. One nice feature of this organization is that a periodic task can easily be created by having a function place itself back on the queue at the end of its execution.

NOS only handles a job or interrupt if there are no jobs or interrupts waiting

```
long update_timeoutq( long then )
{
  struct event *ep;
  long tnow = now();
  long delta = then - tnow;

  ep=(struct event *)LL_HEAD(timeoutq);
  ep->timeout += delta;
  return tnow;
}

int main()
{
    struct event *ep,*tp;
    unsigned long time, timeout,tnow;

    LL_INIT(timeoutq); LL_INIT(freelist);
    init_tasks(); time = now();

    while (1) {

        /* Handle high-priority tasks */
      LL_EACH(timeoutq, ep) {
        if (ep->timeout > 0)  break;
        if (ep->priority == HARD_DEADLINE) {
              tp=(struct event *)ep->prev;
              ep->go(ep, time);
                    /*accumulate the negative timeout */
              if (ep->timeout < 0){
                 struct event * p=(struct event *)LL_NEXT(timeoutq,ep);
                 p->timeout += ep->timeout;
              }
              ep = (struct event *)LL_DETACH(timeoutq, ep);
              LL_PUSH(freelist, ep);
              time = update_timeoutq(time);
              ep=tp; /* Keep doing high-priority tasks */
        }
      }

        /* Handle high-priority interrupts*/
      if (Byte(MEM_IO_INTSRC)==HIGH_PRIORITY) {
            handle_interrupt(Byte(MEM_IO_READ));
            time = update_timeoutq(time);
            continue;
      }

        /* Handle low-priority tasks*/
      ep = (struct event *)LL_HEAD(timeoutq);
      if (ep && ep->timeout <= 0) {
            ep->go(ep, time);
                    /*accumulate the negative timeout */
            if (ep->timeout < 0){
                 struct event * p=(struct event *)LL_NEXT(timeoutq,ep);
                 p->timeout += ep->timeout;
            }
            ep = (struct event *)LL_DETACH(timeoutq, ep);
            LL_PUSH(freelist, ep);
            time = update_timeoutq(time);

            continue;
      }
        /* Handle low-priority interrupt */
      if (Byte(MEM_IO_INTSRC)==LOW_PRIORITY) {
            handle_interrupt(Byte(MEM_IO_READ));
            time = update_timeoutq(time);
            continue;
      }
    time = update_timeoutq(time);
    }
    return (0);
}
```

Figure 3.6: The main control loop of NOS, a multi-tasking non-preemptive scheduler. High priority tasks are handled first, followed by high priority interrupts. Then low-priority tasks and low-priority interrupts are handled.

at higher priority levels. Therefore, at levels beneath priority 1 (HARD jobs that have reached their time to execute), only one job is executed before jumping back to the top of the control loop, e.g., only one interrupt is handled before checking the callout queue to see if any more HARD jobs are ready to run. It is a simple fixed-priority scheduler with the expected weakness that low priority jobs will be ignored indefinitely if there is enough work to do at a higher priority.

### 3.6.2 $\mu C/OS$

The $\mu C/OS-II$ real-time kernel is a full-featured preemptive multitasking RTOS [16]. It is portable, targeted at both micro-controllers and DSPs, and it currently runs on over fifty different instruction-set architectures. It is designed to have a small footprint: there are roughly 1700 lines of code in the OS (including comments), and modules are only compiled into the executable if used by the application. Multi-tasking is preemptive, The system can run up to 64 tasks, with 8 of those tasks reserved for the kernel's use. It provides traditional OS services such as IPC, semaphores, and memory management, and it also provides time-related features such as the ability to sleep until a specified time and callout functions, in which an application can specify code to execute on task creation, task deletion, context switch, and system timer tick. Because $\mu C/OS - II$ has no concept of a periodic task, we used two facilities within the kernel to implement periodic job invocations. Each job sleeps on a unique semaphore, and a user-level task is attached to the clock interrupt ($\mu C/OS - II$ allows user-level code to be attached to arbitrary events). This user-level task keeps track of the job invocation times and generates wakeup messages when the job periods are reached.

# Chapter 4

# Performance Experiments

In this chapter, experiments are conducted to benchmark the real-time attributes of the three real time operating systems. One aspect of the real-time attributes is, if a task is specified to run at period T, how well the OS can schedule to make this task run on the beat. If the task is not executed on time, we call the time error "jitter", without regarding whether it is earlier or later. Jitter tells the predictability of an OS, which is very important in embedded systems.

Another aspect of the real-time attribute is interrupt response time, which means the time from the moment an interrupt is delivered to the processor, to the moment the interrupt handling program responds. This is also a very important factor in embedded systems.

In this chapter, the first section will describe our benchmarks and how we run these benchmarks. The second section will discuss the experiment results.

## 4.1 Benchmarks

Three RTOSs were tested on the test-bed. Two sets of benchmarks were used: periodic inter-process communication(IPC) and a 256-tap Finite Impulse Response filter(FIR). To add some non-determinism to the evaluation of these two operating systems, and to offer more realistic simulations indicative of real-world systems, two different additional tasks were created. These tasks can be run concurrently with the above listed benchmarks to provide a background load. These two tasks are a periodic control loop (CL) and an aperiodic inter-process communication process(AP-IPC).

**Periodic Inter-Process Communication** Periodic inter-process communication (IPC) is the simplest of the benchmarks that were used to evaluate performance. The first job grabs data off of the input I/O port and stores it into shared memory. The second job takes that value from shared memory and writes it to the output I/O port. There is no computation, only the movement of data. This task represents the simplest possible two-job task possible.

**Finite Impulse Response Filter** The finite impulse response (FIR) filter is the most computation intensive of the four benchmarks. The second job runs a 256-tap filter on the data that has been collected by the first task. For each run of the second job, the last 256 values to be inputted by the first job are used to compute an inner product, and that value is output to the I/O port. In other words:

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + \ldots + b_{q-1} x_{n-q+1} \tag{4.1}$$

Where $y_n$ is the output and $x_k$ is the input. (q=256 in this case).

**Background Load** CL is defined as a 32Hz periodic task in this experiments. The task itself does not do much work. Writting to a specific memory address to register itself is one of the few things it does. AP-IPC is the kind of IPC that is driven by interrupts. It runs randomly, with the average period of 10 milliseconds. Again, the main reason to use AP-IPC is to disturb the system and observe the response of the operating systems.

In summary, the following parameters are varied when running the experiment:

- RTOSs: $\mu C/OS$,NOS
- Periodic tasks: IPC,FIR
- Workload: 1,2,4,8tasks
- Periods: 1,2,4,8,16ms
- Background load: AP-IPC+CL, none

## 4.2 Jitter Results

Jitter measurements represent the time deltas between successive output seen at the I/O device for a given executing task. When multiple tasks are executing simultaneously, each writes to a different I/O port, enabling the distinction between tasks, and each task contributes equally to the data in the graphs.

With the definition of jitter, it is clear that if the first time the task is scheduled $\Delta t$ later than it is supposed to be, and from that time on this task is always scheduled $\Delta t$ later than the theoretic time, only the first time there is jitter. This is what we want the system to be, because this way the system is predictable. Figure4.1(a) shows this kind of performance. What we do not want to see is that

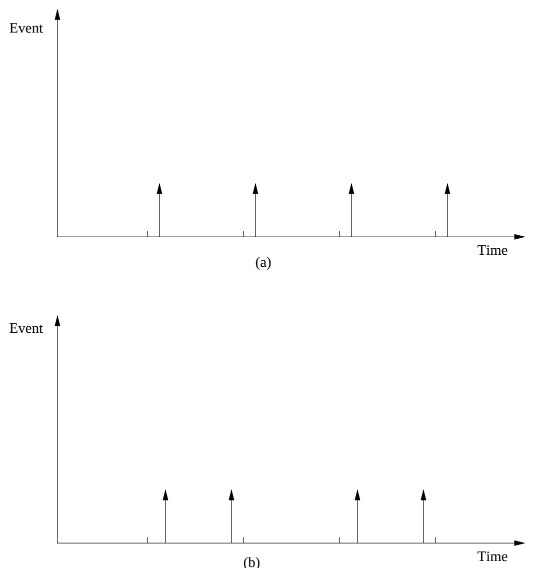


Figure 4.1: Jitter of periodic tasks

the task is sometimes on time, sometimes late, and sometimes early, like what is shown in (b) of Figure 4.1.In this case, the system is not predictable, although it is trying to maintain the task on time. In other words, jitter is the reflection of the predictability of a system.

Since Chris Collins has done a great job in his thesis describing the similar tests running NOS and Echidna, I am not going to go into details about the comparison of these two OSs. Please refer to his thesis [3] for details. In the following discussion, the main focus will be on comparing the results of NOS and $\mu C/OS$.

### 4.2.1 Low System Load

Let's look at Figure 4.2 about the performance of these two OSs when system load is low. The X axis is the error between the supposed scheduling time and actual scheduling time. This is what we called Jitter. The Y axis is the probability of

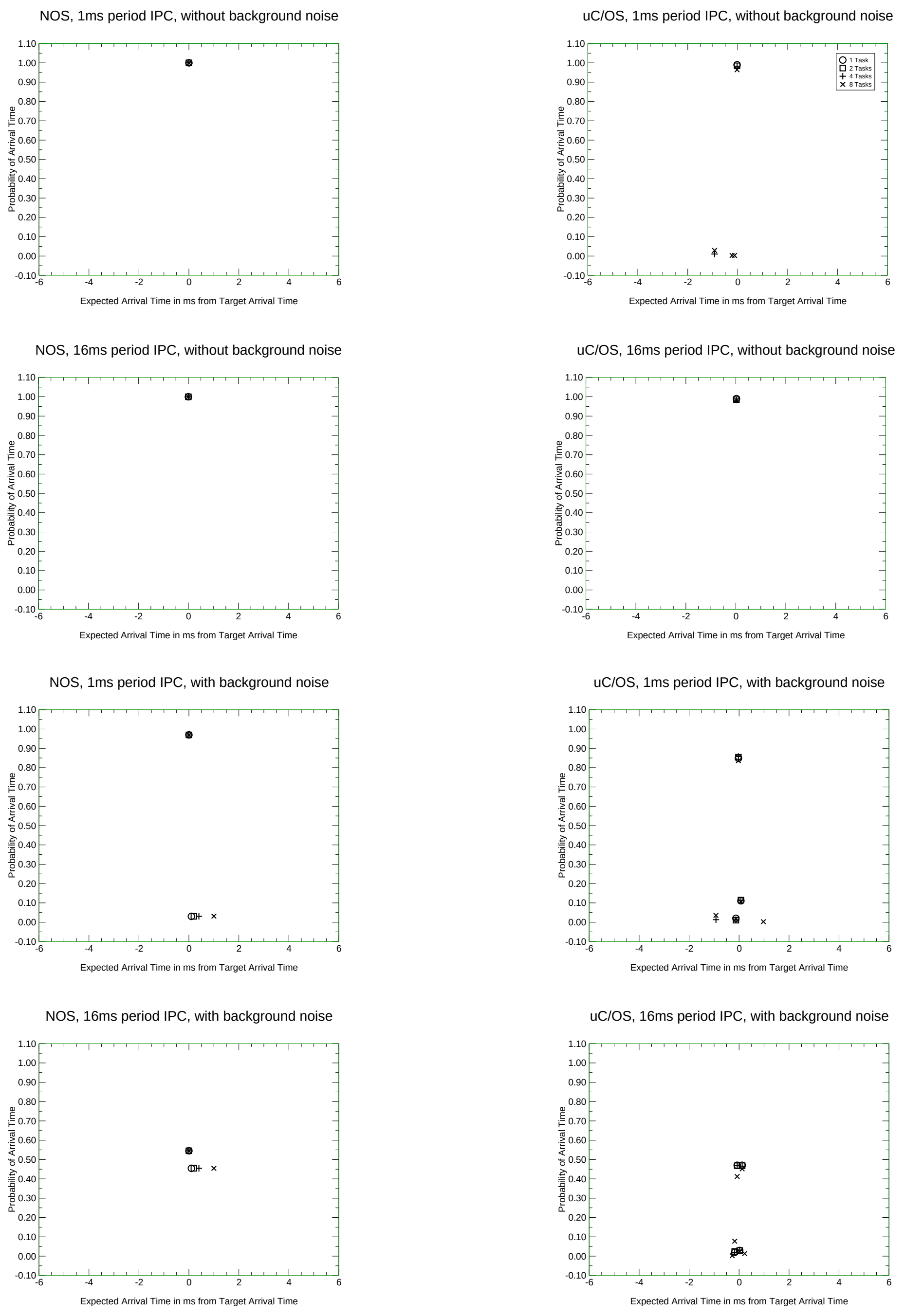


Figure 4.2: Jitter graphs: With relatively low load, both $\mu C/OS$ and NOS performed fairly well. While $\mu C/OS$ tried to stick to 0 with positive and negative jitter, NOS just simply scheduled tasks later.

the jitter. The best case will be 100% time the jitter is 0. When running IPC without background noise(including CL and AP-IPC), not surprisingly, both of the OSs performed pretty well. However, NOS is a little better than $\mu C/OS$ in that when the running period was lowered to 1ms, $\mu C/OS$ occasionally(with less than 1% probability) scheduled tasks earlier. When the running interval went from 2ms to 1ms, this probability increased, although by not much. With NOS, the graph is very clean. Every thing happens just on time.

Now let's take a look at Figure 4.2 to see what happened when the background load was added. Apparently this made a big difference on both systems. Look at NOS first. With 16ms running interval and 1 task, the system is on time 55% of the time, and the rest of the time it is a little bit later. Similar is the case of 2 tasks. However, as the task number increased to 8, more than the 45% of time tasks are late, with maximum delay of 1ms. The strange thing is that about half of the time the system is on time and half of the time it is not. This has to be caused by the background load.

When examining the background load, we found that the CL task was running at 32Hz, which is exactly half speed of the 16ms-interval task. That means every other time the target task will be disturbed by the background task, and thus half of tasks were late.

As the interval time strays from 31ms(the period of 32Hz), the background load has less impact on the system performance. This is shown in the Figure 4.2. When the interval is 1ms, the background load's periodic impact is almost gone(less than 3%), while there is still some influence when the interval is 2ms. Notice that 100Hz AP-IPC did not have a big impact on system performance.

With $\mu C/OS$, it's a similar story. The 16ms-interval tasks were greatly affected by the background load. As the interval decreased, the impact decreased too. However, there was always a small portion of time that the tasks were not on time.

### 4.2.2 High System Load

When the system load went high, e.g. running FIR instead of IPC as the target task, let's examine this comparison again. Please see Figure 4.3 for the experiment results.

First, without background load, both systems performed relatively well when the interval was long(16ms). As the interval decreased, unable to finish all the tasks on time, NOS just pushed all the system work late(the 'x' in Figure 4.3). Note that the running time of 8 FIR tasks is more than 1 ms, simply because the running time of 1 FIR task is more than 1/8 ms. In other words, the system is overloaded when running 8 FIR tasks in 1 ms interval. NOS's performance is very stable and predictable. On the other hand, $\mu C/OS$ successfully scheduled part of the 8 tasks on time, with the price of the rest of the tasks being late. These are just two different ways of dealing with the system being overloaded. Each of them applies to some specific situations.

When background load was added, from the experience of IPC, we expected some disturbance. And that happened. The 16ms interval tasks were greatly affected, both NOS and $\mu C/OS$. When we looked at the 16ms interval NOS graph, the system performance was very bad, though the system should be able to do a better job. Neither system predictability nor the real-time feature was there. Only half of the time that tasks happened on time. The background load accounts for most of this bad performance. As the interval decreased, the

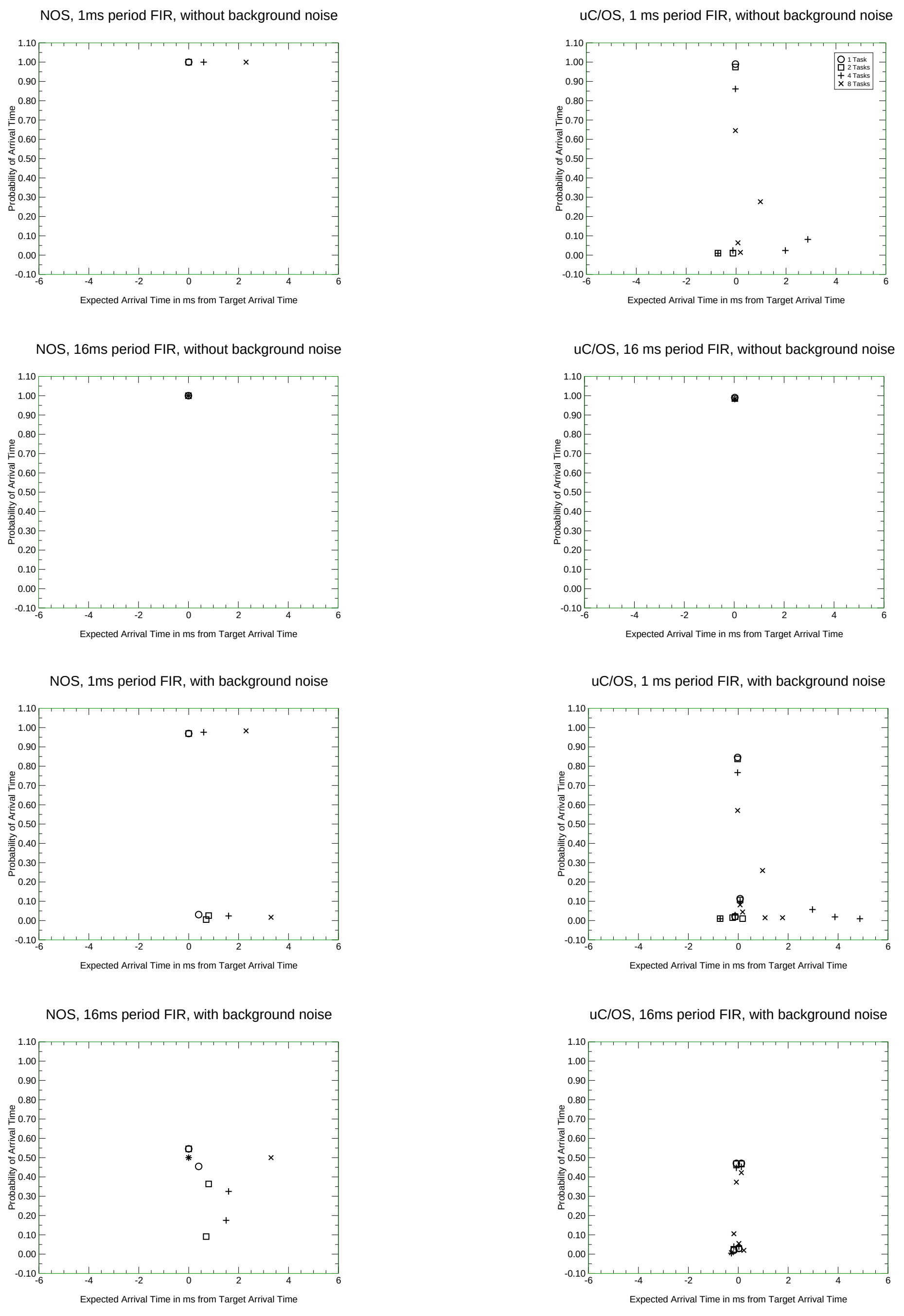


Figure 4.3: Jitter graphs: When system load is relatively high or very high, $\mu C/OS$ and NOS behave differently.

performance got better.

### 4.2.3 Observations

The following things were observed in this experiment:

1. Without background load and with reasonable system load , both NOS and $\mu C/OS$ performed fairly well, with good predictability and real-time feature.

2. When being driven with high load, $\mu C/OS$ tries to maintain part of the tasks on time, while NOS keeps all of the tasks late while providing very good predictability.

3. Background load will couple with the application task and greatly affect the performance of the system. The affect will be worst when the background noise has similar frequency as the application tasks.

## 4.3 Delay

Delay is another indication of a real-time system's performance. Our delay numbers represent the time between an AP-IPC interrupt and the moment that the I/O system sees the corresponding output from the AP-IPC task invoked as a result of the interrupt. Thus, the delay measures the response time of the system in terms of when the first reaction takes place.

The $\mu C/OS$ kernel handles interrupts preemptively, while NOS use a polling technique. What NOS does is that the hard-time task has highest priority. If there is hard-time task ready to run, the CPU always runs that first. The high priority interrupt will be handled when there is no hard-time task ready to run. Thus, when the CPU is heavy-loaded with hard-time tasks, the response time

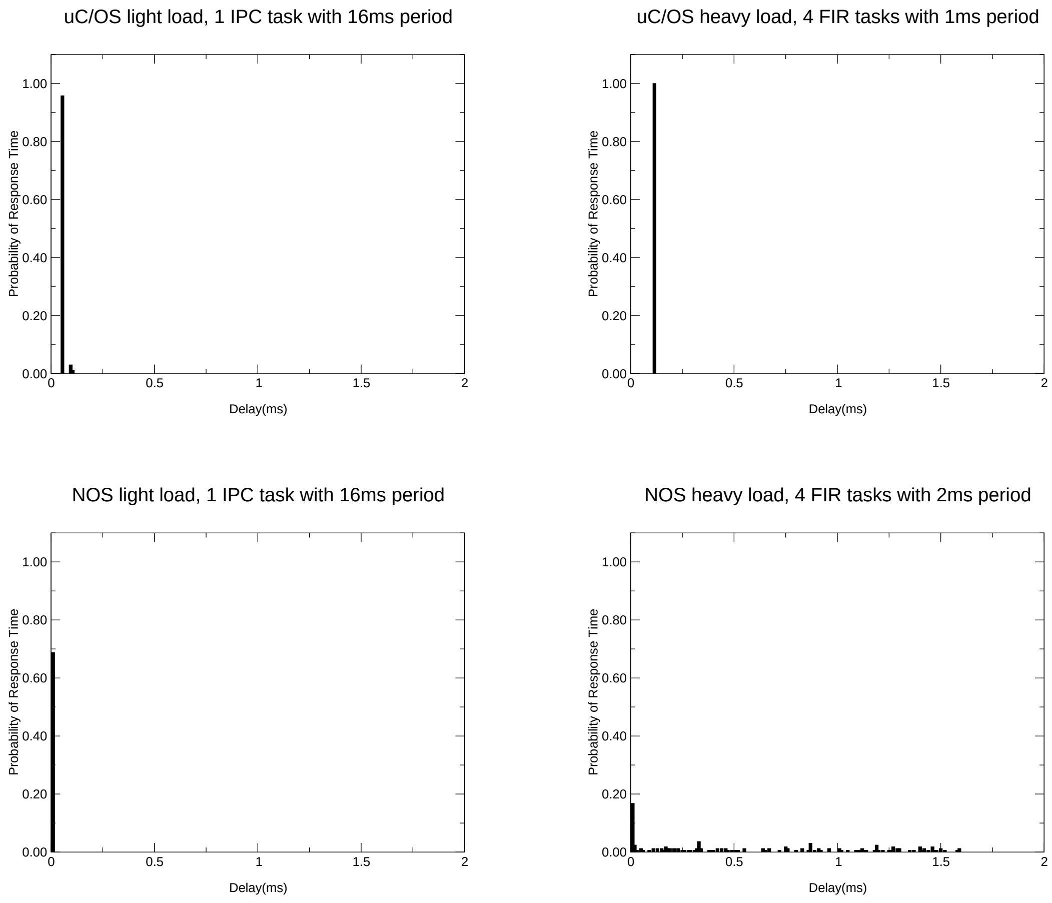


Figure 4.4: Delay graphs: $\mu C/OS$ consistently had a fixed interrupt response delay time, whereas NOS's response delay time is dependent on the system load

might be very bad. This can be clearly seen in Figure 4.4.

Figure 4.4 is the delay times for both $\mu C/OS$ and NOS. These represent the range of CPU load from very light(1 IPC task, 16ms period) to very heavy(4 FIR tasks, 1ms period). As expected of a preemptive OS, $\mu C/OS$ kernel handles interrupts with absolute precision that is independent of application load. The NOS's performance is interesting here. When the system load is light, it has very fast response time, even faster than the preemptive $\mu C/OS$ kernel. This is because NOS is a cooperative OS and when a task switches, there is no state that needs to be saved. As the system load increases, the average response time of the NOS system increases, and the response time spreads out along the time axis randomly. When the system load goes very high, as shown in Figure 4.4, the response time becomes unpredictable and thus unacceptable.

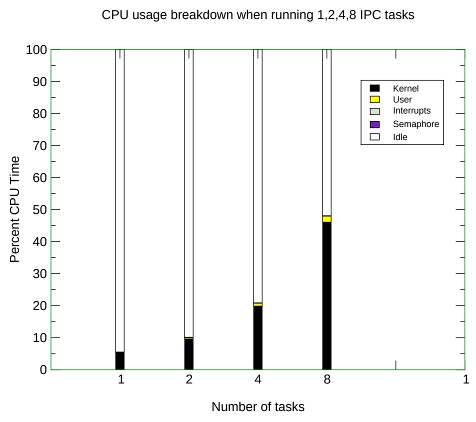
# Chapter 5

# Power Consumption and Speed-setting

In this chapter, the experiments and simulations that were conducted to benchmark the power consumption of NOS and $\mu C/OS$ are presented. First the breakdown charts of the RTOS's power consumption are shown to provide some clues as to which part of the OS is consuming how much power. Then the pros and cons of some obvious ways to lower power consumption are discussed. After that, focus is placed on a new technique that microprocessors just started to support and how this is going to help lower the power consumption of the system. Also an algorithm to utilize this technique is proposed and discussed.

## 5.1 Power And Time Consumption Ratio

The first thing we did after the power model was operational is to measure the CPU usage breakdown. Before we began to analyze the data, we found out that in our experiments, when no power-saving mode instruction is used, the power consumption ratio and the time consumption ratio of a piece of code are very close to each other. Please see Figure 5.1 for the data. Apparently, although each instruction consumes different energy, due to the large number of instructions



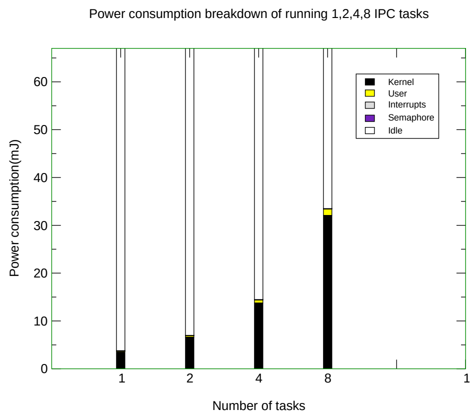


Figure 5.1: Power consumption ratio and time consumption ratio is very similar in a system

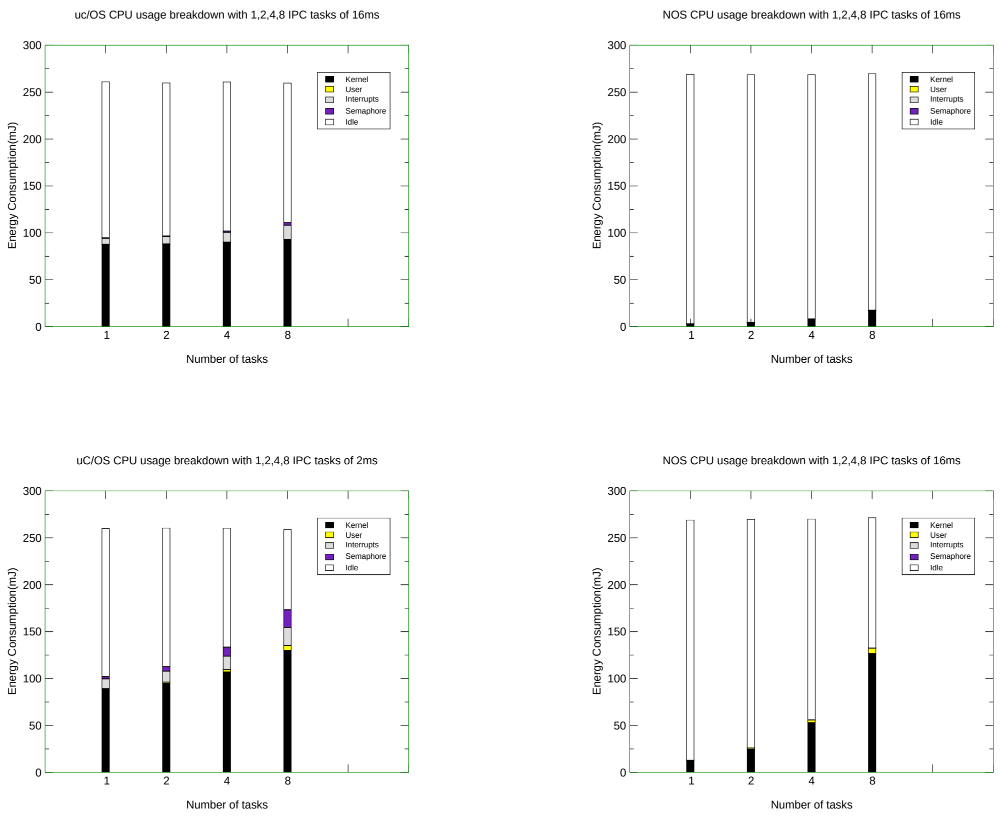


Figure 5.2: Power consumption breakdown of NOS and $\mu C/OS$ with IPC load

each section has(the section here means kernel part of the system, interrupt handling part of the system, user application part of the system, etc. ), this effect is compromised and it leads to the very close power and time consumption ratio. Also, this observation is supported by the data in paper [4], where the two graphs that the author gives are very close to each other. However, the author, R.P.Dick et al. , did not mention any thing about this.

## 5.2 Power Consumption Break-down

In order to see the whole picture of how OS power consumption is distributed during different running loads, we ran NOS with loads ranging from very light to very heavy. We will see how it handled all these loads.

Figure 5.2 shows the energy consumption breakdown of NOS and $\mu C/OS$ when the system is on relatively light load (running IPC). These are the kinds of situation when the tasks are running frequently but the tasks themselves are not very time-consuming. Obviously, when IPC is running at 16ms period, almost no computation load exists, and running an OS is overkill. $\mu C/OS$ spends most of the time doing kernel activity, while NOS is much more efficient in this case. The kernel power consumption of NOS scales with application load, but this is not the case with $\mu C/OS$. In the case of 2ms period, the energy consumption of the NOS kernel increased by about ten times from 1 task to 8 tasks, but the $\mu C/OS$ kernel energy consumption even dropped a little bit when the load increased (see Figure 5.3). This is a very interesting phenomenon.

Figure 5.3 are the breakdown graphs when FIR tasks run. The FIR task is the kind of task that consumes significant CPU time to do computation. Here $\mu C/OS$ is clearly showing that when the load increases, the kernel spends less time on its own activity. Also, interrupts account for a considerable part of the system energy consumption. Semaphores are used to do the periodic calling of each task, since that $\mu C/OS$ does not support periodic tasks by itself.

Several results can be seen in the data:

- The systems consume an enormous amount of energy doing nothing, as represented by the idle components. This is because neither NOS nor $\mu C/OS$ has an intelligent sleep mechanism that can use less power when there is

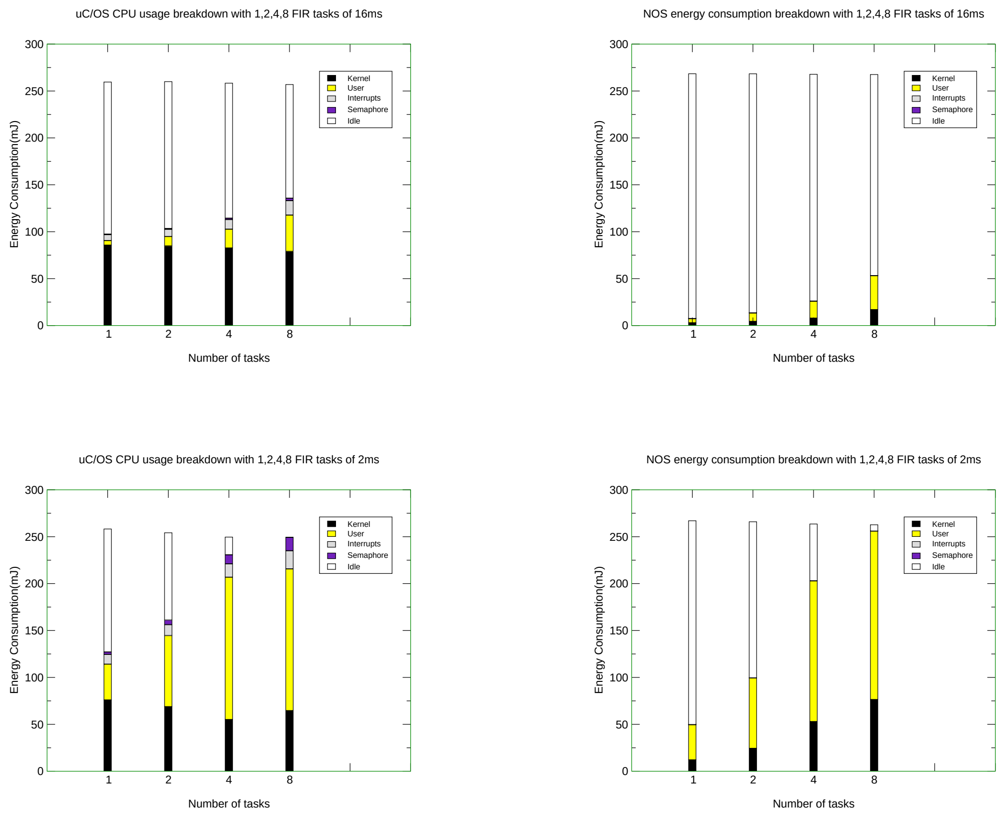


Figure 5.3: Power consumption breakdown of NOS and $\mu C/OS$ with FIR load

nothing to do; though the MCORE has such a facility (a doze mode that can be awakened by a timer interrupt), the system just does not use it. If implemented, this would save considerable energy resources in theory. Note, however, that there is very little idle time as the system is pushed up to but not beyond its limits, which is where embedded system engineers would like their systems to be, as this makes most effective use of the CPU resources.

- The kernel overhead in NOS scales with the application workload , while the kernel components in the other RTOSs are more constant. The more sophisticated RTOS does a better job of ensuring that all computations are deterministic in the time and energy it takes to perform them, which gives more predictable system behavior. The cost is obviously a higher starting point for energy consumption.
- If user applications are not very heavy-duty, using an operating system like NOS gives the system much more potential for power saving. This is because of the simplicity of NOS. Running a complicated RTOS could be an overkill in some cases.

## 5.3 Low Power-consumption States

From the data in Figure 5.2 and Figure 5.3, it is clear that there are many occasions where the system has a considerable amount of idle time . Without doing anything during this idle time, the system will just sit there spinning, waiting for next task to be ready. Apparently this is not very energy efficient, and we need to do something to utilize this idle part.

The first thing that comes to most people's mind is to use the power-saving modes that the processor provides. Most modern processors provide at least two kinds of power-saving modes. One is not-very-deep sleep, or doze. Usually this mode just stops clocking the core of the processor, while all the other parts of the processor are still working, including the clock itself. Once the processor is awakened, usually by an external interrupt, the processor simply starts clocking the core again. The advantage of this method is that it takes a very short period for the processor core to recover. For example, the Z180 processor from Zilog can recover from the lightest sleep mode within only 1.5 clocks[14]. The disadvantage of this method is that since only the core is stopped during sleeping time, while the oscillator is still running, limited energy consumption is saved. Jeff Scott et al. mentioned in their paper [21] that the clcok activity itself consumes about 35% of core's power.

The other power-saving mode is the deep sleep mode, or stop. In this mode, most parts on the processor chip are stopped, with only some essential parts, like the time-of-day timer, still running. When the system is awakened by the timer interrupt or an external interrupt, it needs a relatively long period of time to warm up before it is able to stably fetch and execute instructions. Usually this involves the physical oscillator stabilizing period, so it could be long. For example, in the new Z180 processor from Zilog, the wake-up time from deepest sleep mode is $2^{17}$ clock cycles. If the system clock is 20MHz, this will more than 6 milliseconds! Obviously, unless the system is known to be idle for a long time, this method does not fit our needs.

Therefore, there is no simple way for us to utilize the idle part of the system. In the next section, we will talk about a new technique that the hardware supports to enable the software to utilize the idle time more efficiently.

## 5.4 Speed-setting And Voltage Scaling

In this section, speed-setting technology will be discussed. An algorithm that utilizes this feature is proposed and the consequent power consumption and performance results are discussed. The first part of this section is dedicated to a general discussion of the desirability of speed-setting algorithms. The second part discusses the new algorithm and the simulation results. The last part gives out some possible future research directions.

### 5.4.1 Desirability of Speed-setting

From the discussion in the background chapter it is clear that the power consumption of a chip is proportional to the clock rate that drives it. Therefore, by slowing down the clock rate, the power consumption of the chip should be lowered proportionally. However, when talking about energy consumption, which is proportional to the time that the task executes, the energy consumption of executing the task does not change. Let's look at the following example. Suppose the original task needs $n$ cycles to get completed, and the original clock frequency is $f_{clock}$.So the time the task get completed is $n * 1/f_{clock}$ seconds. Thus We have:

$$Power = KV_{dd}^2 f_{clock} \tag{5.1}$$

and

$$Energy = KV_{dd}^2 f_{clock} n * 1/f_{clock} \tag{5.2}$$

which can be simplified as:

$$Energy = KV_{dd}^2 n \tag{5.3}$$

where K is a coefficient, and $V_{dd}$ is the supply voltage.

Therefore, theoretically simply slowing down the clock speed will just lower the power consumption while keeping the energy consumption unchanged.

Is this the case in real life? Is this true for the whole system, instead of just for one task?

The assumption we made above is that during the n cycles of execution time, the CPU was never idle, and we only care about the n cycles. With this assumption, the above statement, i.e., by only scaling down clock speed the energy consumption for executing this task is not changed, is true.

However, from the perspective of the whole system's view, the assumption is hard to maintain. The whole system has to have some time (possibly to a large degree, for example, 30% of total time) to be idle. In an embedded system, the maximum schedulable usage of CPU for RMA(Rate Monotonic Algorithm) scheduling is 70%. This means that when the system CPU usage is less than 70%, by using RMA it is guaranteed to be schedulable. When the CPU usage exceeds 70%, RMA may not be able to schedule it. Therefore, as a rule of the thumb, most embedded systems have at least 30% CPU idle time, if not more, although it might not be evenly distributed.

With consideration of idle time, the above statements then will not necessarily be true. Let's examine Figure 5.4.Part (a) is that the system is running at full-speed clock when there is something to do. When the task is done, the system will immediately go to sleep. When there is job to do again, the system will go back to work immediately. Part (b) is the power consumption situation when

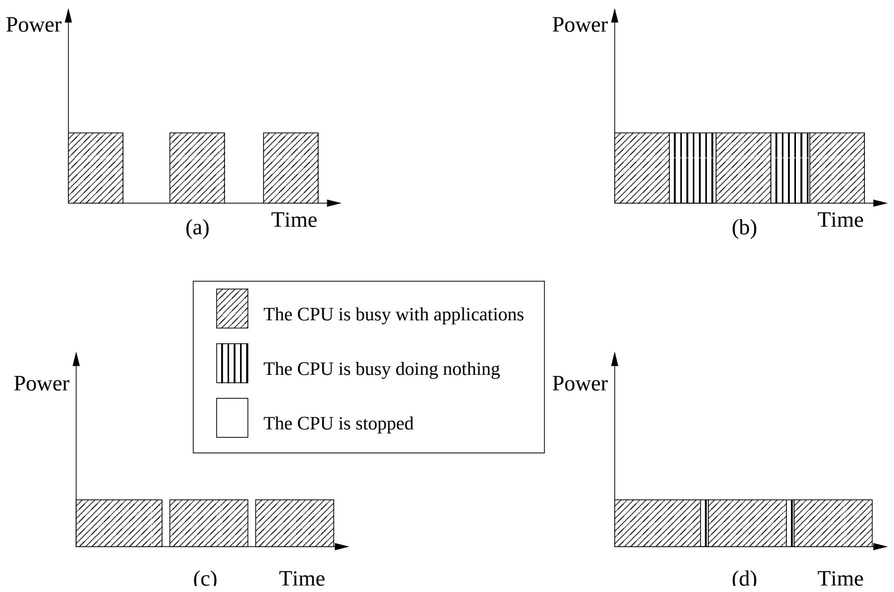


Figure 5.4: Four possible cases with ideal low-power mode and speed-setting. (a) the CPU runs at high speed with ideal sleeping mode. (b) the CPU runs at high speed with no sleeping. (c) the CPU runs at low speed with ideal sleeping mode. (d) the CPU runs at low speed with no sleeping.

the system is running at full-speed clock rate all the time, no matter the CPU is busy or idle. Part (c) is that the system is running at a partial-speed clock with an ideal stop function like in part (a). Part (d) is that the system is running at a partial-speed clock all the time, no matter the CPU is busy or not. Part (a) and (c) are ideal situations, while part (b) and (d) are realistic situations for many embedded systems.

One assumption for this figure is that the system has enough idle time so that even when the CPU slowed down from (a)(b) to (c)(d), it is still able to finish all its jobs. In other words, after slowing the CPU, like in part (c), the shaded area will not overlap.

From Figure 5.4 we know that the energy consumption of part (a) and part (c) are the same, and they consume the smallest amount of energy among these four options. However, from the discussion of previous sections in this chapter we know that deep sleep mode is very expensive in terms of recovering time. It might take up to several milliseconds. Also during the wakeup period, the power consumption might have glitch, i.e. a moment that the power consumption is greater than normal running state. With this in mind, stopping and restarting the processor while running a system is not a good choice unless the software knows that the system will be idle for a relatively long time. For instance, there is only 1 task running periodically and the period is 5ms. The execution time of the task is 1ms. That means the 80% of the CPU time is idle. However, the wakeup time of deep sleep mode for a 20MHz processor is

$$2^{17} * 1/20,000,000 = 6.55ms \tag{5.4}$$

If the CPU sleeps after the 1ms task execution, it will not be able to wake up until 7.55ms. It will miss the next starting point of this task. This makes part(a) and part(c) not eligible for most real-life embedded systems. Another case is that when, due to some reason, like the system is using a polling mechanism instead of interrupts, the system just can not go to sleep during idle time.

In the above cases, only two choices are left, which are part(b) and part(d). And clearly part (d) is consuming less energy than part(b) and therefore is a better choice. Again let's do the numbers. If we slow down the CPU to 25% of original speed in the above example, the power consumption of the CPU will be 25% of original one. Now the task takes 4ms to complete instead of 1ms, but it is still able to finish the task before the start of next period task. This way, all the work are finished as before, just a little slow, and the energy consumption is only a quarter of original value. Of course, this is based on the assumption we made above. If the CPU is fully used, there will be energy difference no matter we slow down the CPU or not.

The point of this subsection is to state that even the simple speed-setting technology ( instead of doing speed-setting and voltage scaling at the same time) will benefit a real embedded system in most cases. The negative effect of adopting speed-setting technology is the system response time. With full-speed clock running, the system can respond to an interrupt at full speed. With a partial-speed clock, the interrupt response time will be lengthened correspondingly.

To give itself more strength, speed-setting enabled the technique called voltage scaling. According to our discussion in the background chapter, when the clock speed goes lower, the processor supply voltage could be dropped correspondingly. Because power consumption has a square relationship with supply voltage, this will provide a quadratic drop of power consumption, hence energy consumption. Based on speed-setting technology, this saving almost comes for free. That's why speed-setting and voltage scaling is now becoming a very promising area.

### 5.4.2 Test-bed Modification

The target test-bed was modified to accommodate this new technology. Because MCORE itself does not have a speed-setting function, we borrowed these data from other chips like Transmeta's Crusoe.

The modified processor can run at speed of 20MHz down to 2MHz, with 36 averaged steps between. A special register is dedicated to change the speed of the processor. After a new speed number is written to the register, a certain amount of time will elapse before the processor can run at the new speed. During this transition period, no instruction is executed. The transition period depends on the steps the processor jumped. The bigger the step, the longer it takes. Also the energy consumption of the transition period is dependent on the transition steps.

When the processor is running at 20MHz, the power supply voltage is 3.3V. When the processor speed drops to 2MHz, the power supply will drop to 1.1V, according the relationship of these two factors discussed in the background chapter. The power supply voltage of all the clock steps in between will be determined similarly.

### 5.4.3 Group Scheduling Algorithm

Several groups have been working on coming up with scheduling algorithms to utilize the speed-setting and voltage scaling. Most of them did the system load prediction based on the past record, like what has been used in branch prediction in computer architecture design. The PAST algorithm[27], which is one of earliest and well-known algorithms, does the prediction at fixed periodic moments. The PEAK algorithm [9] uses the same basic idea with more effort on how to make good predictions.

Group scheduling algorithm is different from the above algorithms in that the prediction of future system load is not purely based on the history. Instead, it utilizes all the information that NOS provides about each and every task plus some degree of prediction to calculate the future system load and corresponding processor speed. Among NOS kernel data structures, there is a timeout queue to keep track of the tasks. Each task will insert (or append) itself into(onto) the timeout queue according to the next time execution time. Please refer to the test-bed chapter for a detailed description. By this way the OS will be able to foresee the possible system load in the near future. In fact, the timeout queue only provides the information of which task is going to be executed at which moment, but has no clue of how long each task is going to take, which is necessary to calculate the system load.

The group scheduling algorithm uses a system timer to keep track of the history execution time of each task, and predicts the next execution time based on these history data. Combining the prediction execution time of each task and the timeout queue information, the group scheduling algorithm will calculate the next step processor speed. This speed will be used to execute all the tasks in the timeout queue. The next prediction will not be made until all the predicted tasks get executed.

Please see Figure 5.5 for a simple example. In this example, at time t0, there are three tasks in the timeout queue, with their own timeout value and predicted execution time value. The algorithm predicts the future system load

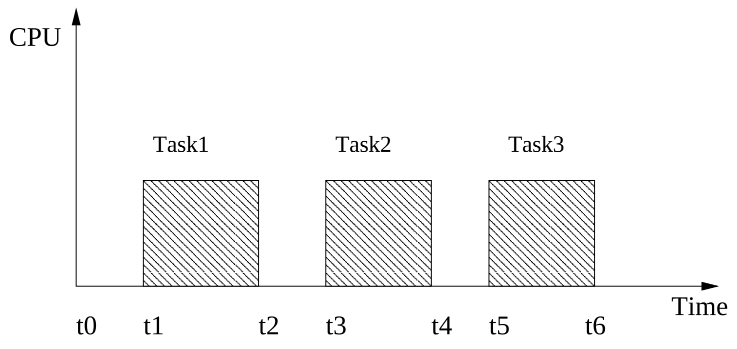


Figure 5.5: At moment t0, the system has 3 tasks to schedule, with known starting moments

by calculating the ratio of free time to busy time during a time period of t0 to t6, and then sets the processor speed to the corresponding speed.In this example, at moment t0, the algorithm will set the processor speed to

$$Speed = FullSpeed \cdot (\Delta t_{21} + \Delta t_{43} + \Delta t_{65})/\Delta t_{60} \tag{5.5}$$

The system will run at this speed until task 3 finishes and then another prediction and speed-setting will be made.

The execution time of a task is predicted by using the sliding windows algorithm, i.e.

$$t_{next} = (kt_{past} + t_{current})/(k+1) \tag{5.6}$$

When the system first starts, it runs at full speed. The prediction can only be done when all the tasks have ran at least once.

This algorithm has several features that other algorithms do not have. First, the prediction is not done exactly periodically, instead, it speed-sets several tasks each time. This makes sure that speed-setting is not done too often. Usually it is not desirable to do more than one time speed-setting during the execution of one

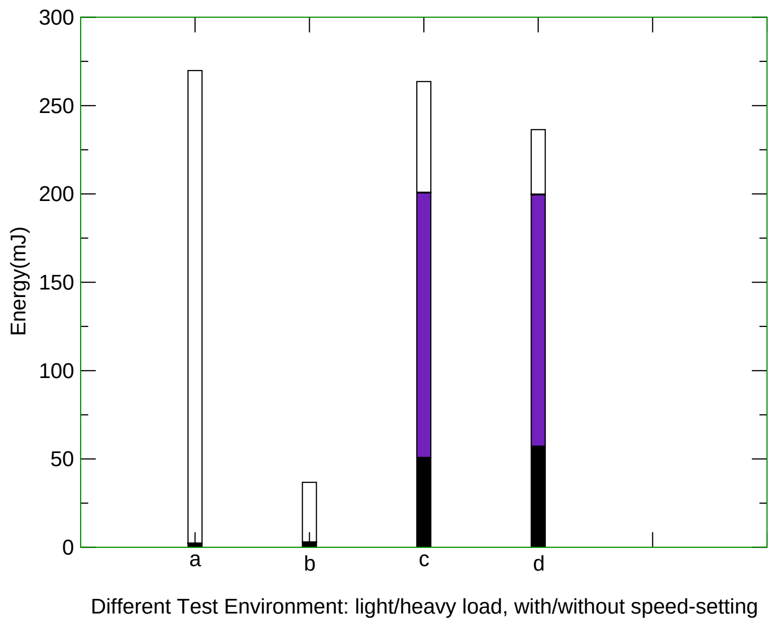


Figure 5.6: Power consumption results with and without speed-setting. (a) is low load and no speed-setting. (b) is low load and speed-setting. (c) is heavy load and no speed-setting. (d) is heavy load and speed-setting

task[11]. Second, this algorithm uses the information that the operating system provides, and this will make the algorithm more accurate.

With no doubt, this algorithm has its own limitations. It needs the OS to provide some information about the tasks. Some OSs simply do not have this information. For example, running a periodic task in $\mu C/OS$ by using semaphores, the OS will not be able to know the future execution time of the task.

### 5.4.4 Experiment Results

Figure 5.6 is the energy consumption graph with and without using speed-setting algorithm, under light load and heavy load, respectively. Obviously when the system is in light load, the processor has plenty of idle time, so that its speed is set to minimum to lower the power consumption. As a result, the energy consumption is much lower than without doing speed-setting. Please note that because the user application has very short execution time, even when the processor is slowed down, it still does not eat much of the processor resource. Therefore, there is still much idle space.

When the system load goes heavy, it is a different story. Because of heavy load, the processor can not be slowed down very much. Therefore, the energy saving is not as dramatic as in light load. In this case, the system tries to maintain a minimum processor idle time. Because of irregularity of processor time use, there is still a small amount of idle time.

With the energy saving graph shown, it is necessary to take a look at the performance impact brought by the speed-setting. Figure 5.7 is the jitter graph with and without speed-setting, with light load and heavy load respectively. Figure5.8 is the delay graph in the above situations. We can see that the jitter graph is not greatly changed by the introduction of speed-setting. That means most tasks can still make their time on the beat, as they do without speed-setting. However, the delay graph shows a big difference before and after speed-setting. This is mainly because the response time is directly related to the processor speed. While slowing down the processor speed by the factor of 10 will give us dramatic power saving, we need to pay the price of 10 times slower interrupt response time.

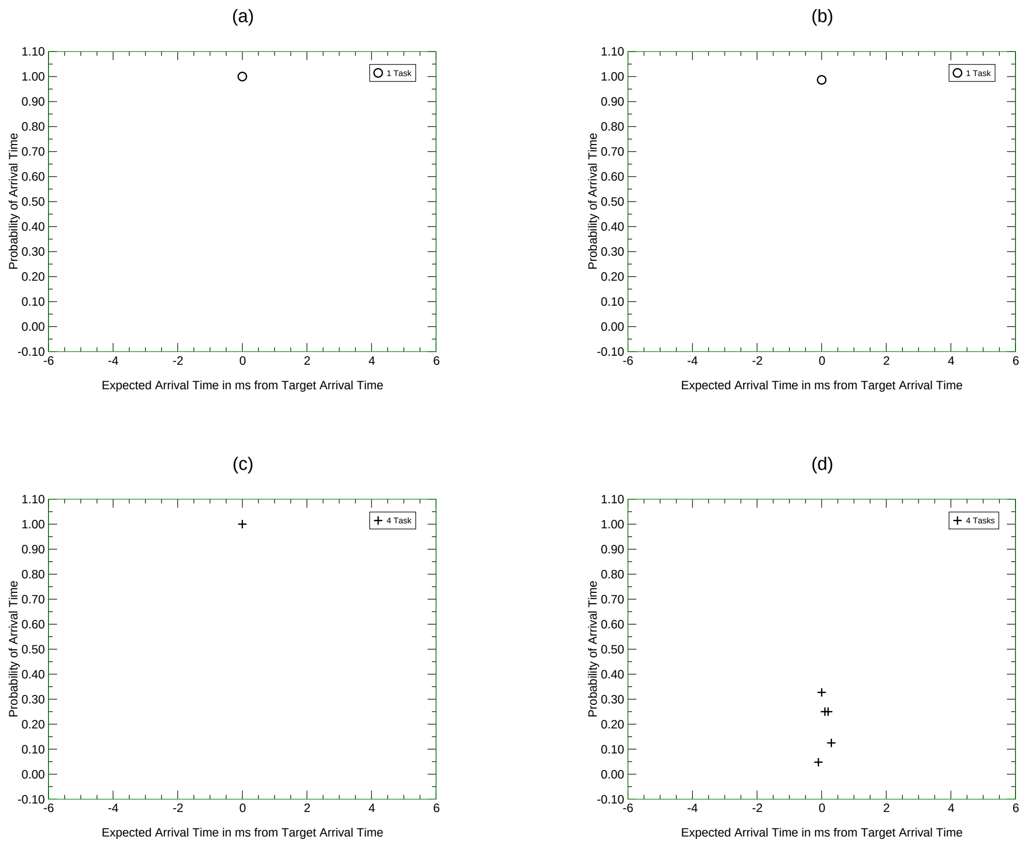


Figure 5.7: Jitter results of NOS with and without speed-setting. (a) is light load and no speed-setting. (b) is light load and speed-setting. (c) is heavy load and no speed-setting. (d) is heavy load and speed-setting.

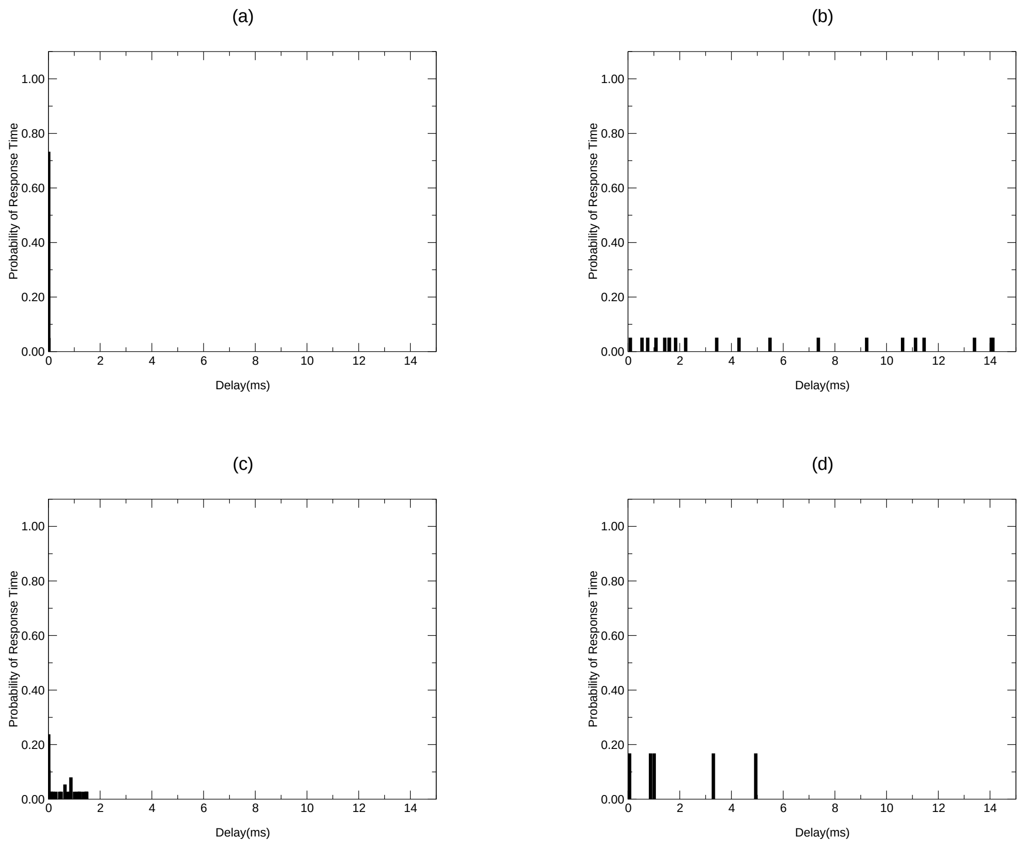


Figure 5.8: Interrupt response delay of NOS with and without speed-setting. (a) is light load and no speed-setting. (b) is light load and speed-setting. (c) is heavy load and no speed-setting. (d) is heavy load and speed-setting.

## 5.5 Metrics of Evaluating Embedded Systems

There is a basic question on evaluating embedded systems: the metrics. If algorithm A can bring down the energy consumption by a factor of 2, and increases the response time by a factor of 1.5, while algorithm B can lower the energy consumption by a factor of 3 and increase the response time by a factor of 2, which one is better? The answer depends on which metric one is going to use. If one does not care about response time, algorithm B will be a better choice. However, if one cares about both of them, the winner could be either of them, depending on what the evaluation metric is.

We suggest a generic way to evaluate such systems. Our metric is

$$PenaltyScore = Energy^{m} Performance^{n} \tag{5.7}$$

Where the PenaltyScore is the comprehensive number that reflects the overall performance of the system.

As we have discussed before, the performance of an embedded system is divided into two parts in this thesis: jitter and response delay. The bigger the jitter, the worse the performance. So is response delay. From equation 5.7 we know that the bigger the Penalty Score is, the worse the system is. That is how we got the name for Penalty Score.

The response delay can be accurately represented by the average value of the delay data , i.e. the mean of the delay data. The jitters can be represented by the standard deviation of the jitter data. Please note that by measuring only the standard deviation of the jitter, we are ignoring the mean, which is the average lateness of all tasks. The assumption is that the system is on-time on average, i.e., the average jitter is equal to 0. This is to measure the predictability of the

|  | Excellent(1) | Good(2) | Okay(3) | Bad(4) | Really bad(5) |
|---|---|---|---|---|---|
| Jitter | 0-0.02 | 0.02-0.05 | 0.05-0.1 | 0.1-0.2 | Above 0.2 |
| Delay | 0-0.5 | 0.5-1.2 | 1.2-3 | 3-6 | Above 6 |

Table 5.1: The grouping standard for Jitter and Delay

system.

Since the jitter numbers are standard deviations and the delays numbers are means, and they have different ranges and different changing rates , it is not disirable to just multiply them together to form the metric. For example, for the system with light load and no speed-setting, the delay number is 0.007. With speed-setting, the delay number is 5.74. If we simply use $Energy \times Delay$ as the metrics, the speed-setting needs to lower the energy consumption to $0.007/5.74 = 0.0012$ of original energy consumption to be able to compete. This is not very reasonable. I borrowed the idea of non-linear mapping from fuzzy logic[25] to divide both jitter and delay into 5 categories: 1 is very good, 3 is okay, 5 is very bad. And 2 and 4 are between for middle cases. The border numbers for this specific system are set as in Table 5.1. The numbers in Table 5.1 are set according to the experimental numbers and experience we had. They can be changed to adapt different situations.

If we simply take the response delay to represent the Performance part, when m=1, n=0, equation 5.7 is an energy-only standard. When m=1,n=1, it is an energy-delay standard [12].

Now let's compare the system performance before and after speed-setting and voltage scaling by using four different standards that origin from equation 5.7.

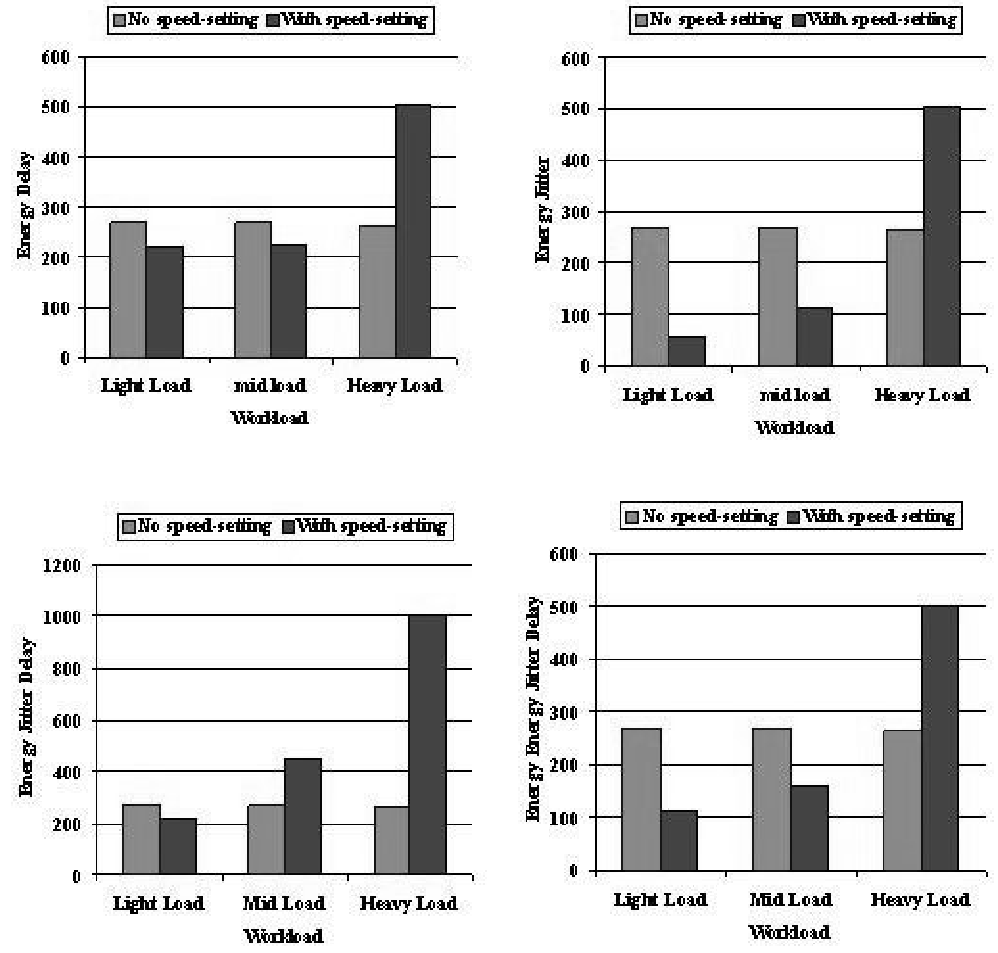


Figure 5.9: Overall performance of NOS with different evaluation metrics.

| Workload/speed-setting | Energy | Delay/Category | Jitter/Category |
|---|---|---|---|
| Light/No | 268.76 | 0/1 | 0.007/1 |
| Light/Yes | 55.62 | 0/1 | 5.74/4 |
| Middle/No | 269.1 | 0/1 | 0.04/1 |
| Middle/Yes | 56.4 | 0.03/2 | 4.4/4 |
| Heavy/No | 263.4 | 0/1 | 0.484/1 |
| Heavy/Yes | 251.57 | 0.02/2 | 0.942/2 |

Table 5.2: The performance and energy data of NOS under different workload

Definations of the four metrics:

$$EnergyDelay : PenaltyScore = Energy \times Delay \tag{5.8}$$

$$EnergyJitter : PenaltyScore = Energy \times Jitter \tag{5.9}$$

$$EnergyJitterDelay : PenaltyScore = Energy \times Jitter \times Delay \tag{5.10}$$

$$EnergyEnergyJitterDelay : PenaltyScore = \sqrt{Energy^2 \times Jitter \times Delay} \tag{5.11}$$

The X axis of the graph is the system load with and without the speed-setting. We collected the data under light, middle and heavy loads. The light load is running 1 IPC task with 16ms period. The middle load is running 2 FIR tasks with 16ms period. The heavy load is running 4 FIR tasks with 2ms period.

Table 5.2 shows the data we collected. Obviously, with different evaluation metrics, we have different results regarding whether it is good to do speed-setting. With energy jitter metrics, doing speed setting is not very promising. Only when the system had light load has the speed-setting done a little better. With both middle and heavy load, it was outperformed by the original system. However, if we use the other three metrics, the with-speed-setting system outperformed the original system. Only under heavy load has the system been outperformed by the original system.

From Figure 5.9 we can get two points:

1. Speed-setting itself is worth a try when the system load is not very high. With heavy load, the system does not have much free CPU time to offer, and therefore, the introduction of speed-setting might impact the performance of the system much more than the gain of the energy consumption.

2. The metrics of evaulating an embedded system will determine whether an algorithm is good. Although the energy delay standard is now the main one being used, my personal point of view is that there is never "the" standard that meets every person's requirements. Some systems need to stress more on performance than energy, and some systems more on energy than performance. People should be able to tailor their own standard based on the specific case of their own embedded systems.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

In this paper, the construction of a full-featured embedded system test-bed was discussed. This test-bed is based on a cycle-accurate Motorola MCORE processor emulator that is equipped with functions like speed-setting and voltage scaling , peripherals like interrupt controller, timer, flash memory and display, and features like memory bus monitoring and power consumption modeling. This test-bed will provide a great research base for embedded processor architecture research and embedded operating systems research.

A widely-used public-domain preemptive real time operating system $\mu C/OS$ was ported to the test-bed, and different kinds of benchmarks were used to test the performance of this OS against the performance of a simple home-made non-preemptive multi-tasking scheduler NOS. Task scheduling jitters and interrupt response delay were used as the criteria of the performance test. With light load, both operating systems had low jitters rates. With heavy load, NOS schedules tasks by a fixed late period all the time, while $\mu C/OS$ schedules part of the tasks on time, and the rest of the tasks later. Periodic background noise couples with the foreground tasks, and the impact is very obvious when the frequency is close. The interrupt response delay for $\mu C/OS$ is fixed all the time, as expected from a preemptive operating system, while the response delay of NOS heavily depends on the system load. When lightly loaded, NOS has shorter response time than $\mu C/OS$ because of its simplicity and non-preemptivity. When heavily load, NOS has response time evenly distributed on the time line, but $\mu$Cos is still short.

Power consumption breakdown data were achieved for both NOS and $\mu C/OS$. When the load is light, NOS kernel consumes less energy than $\mu C/OS$. When heavily load, $\mu C/OS$ consumes less energy than NOS. The kernel power consumption of $\mu C/OS$ is less dependent on the system load than NOS.

Group Scheduling algorithm of speed-setting is discussed. This algorithm takes advantage of the task information provided by NOS to do the speed-setting. The speed-setting is done dynamically according to the system load. More than 80% of the energy can be saved when the system load is light, but only 5-10=% is saved when load is heavy. Also, performance of the system in terms of task execution jitter and interrupt response delay are discussed for speed-setting. It turned out that the jitter performance is not heavily influenced by the speed-setting, but the delay performance is. A generic metric of evaluating the overall system performance in terms of both computing performance and energy consumption is proposed.

## 6.2 Future Work

More work on speed-setting and voltage scaling needs to be done, including comparing the group scheduling algorithm with other algorithms that have been proposed by other fellow researchers.

Also serious embedded system benchmarks like GSM encoding/decoding algorithms, MPEG decoding algorithms, etc. would be very helpful in benchmarking the real time operating systems.

Another interesting research is to benchmark different architectures, like a DSP processor and MCORE RISC processor, by running the same operating system and benchmark-applications.

# BIBLIOGRAPHY

[1] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *The 27th Annual International Symposium on Computer architecture*, 2000.

[2] Rata Yu Chen, Mary Jane Irwin, and Raminder S. Bajwa. An architectural level power estimator. *Proceedings of the 1998 Power Driven Microarchitecture Workshop*, 1998.

[3] Christopher M. Collins. An evaluation of embedded software behavior using full-system software emulation. Master's thesis, University of Maryland, College Park, 2000.

[4] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Power analysis of embedded operating systems. *37th Design Automation Conference*, June 2000.

[5] D.Kalinsky. A survey of task schedulers. *Embedded Systems Conference*, September 1999.

[6] Processor Emporium(UK). Intel pentium 4 to ship in october. *http://www.baznet.freeserve.co.uk/x86n67.htm*, 2000.

[7] Jerry Frenkil. Issues and directions in low power design tools: An industrial perspective. *Proceedings of the 1997 international symposium on Low power electronics and design*, 1997.

[8] J. Ganssle. Conspiracy theory, take 2. *The Embedded Muse newslesster"*, March, 22 2000.

[9] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. *Proceedings of Internactional Conference on Mobile Computing and Networking*, 1995.

[10] Dirk Grunwald, Philip Levis, and Keith I. Farkas. Policies for dynamic clock scheduling. *4th Symposium on Operating System Design and Implementation*, October 2000.

[11] Inki Hong, Darko Kirovski, Gang Qu, and Mani B. Srivastava. Power optimization of variable voltage core-based systems. *Proceedings of the 35th annual conference on Design automation conference*, 1998.

[12] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. *IEEE Symposium on Low Power Electronics*, 1994.

[13] Motorola Inc. *MMC2001 Reference Manual.* Motorola Inc., 1998.

[14] Zilog Inc. Z80185/z80195 preliminary datasheet. Technical report, Zilog Inc., 2000.

[15] Brian Kurkoski. Design embedded systems for low power. *http://www.edtn.com/embapps/emba002.htm*, 2001.

[16] Jean J. Labrosse. *MicroC/OS-II, The Real-Time Kernel.* R&D Books, 1998.

[17] Steve Leibson. The shift from speed to power dissipation. *Microprocessor Report*, October 2000.

[18] J.W.S. Liu and et al. Perts: A prototyping environment for real-time systems. *Proceedings IEEE Real-Time Systems Symposium*, 1993.

[19] M.J.Bach. *The Design of the UNIX Operating System.* Prentice-Hall Inc., 1986.

[20] Alex Pike, Steve Jacobs, and Mike Sickmiller. Thermal simulation of a unique ultra-thin semiconductor packaging architecture for improved power dissipation. *The 33rd International Symposium on Microelectronics*, August 2000.

[21] Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. Designing the low-power mcore architecture. *Proceedings IEEE Power Driven Imcroarchitecture Workshop*, 1998.

[22] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Energy-efficient design of battery-powered embedded systems. *Proceedings 1999 International Symposium on Low Power Electronics and Design*, 1999.

[23] D.B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, December 1997.

[24] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Poweranalysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, December 1994.

[25] Texas A&M University. Introduction to fuzzy logic. *http://www.cs.tamu.edu/research/CFL/fuzzy.html*, 1997.

[26] N. Vijaykrishnan, M.Kandemir, M.J.Irwin, H.S.Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. *The 27th Annual International Symposium on Computer architecture*, June 2000.

[27] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. *Proceedings of the First Symposium on Operating Systems Design and Implementation.*, 1994.