ABSTRACT

Title of Thesis: NANORPOCESSORS: HARDWARE ACCELERATORS FOR EMBEDDED SYSTEMS

Lei Zong, Master of Science, 2003

Thesis directed by: Professor Bruce L. Jacob
Department of Electrical and Computer Engineering

Today's consumer market is driven by technology innovations. Many technologies that were not available a few years ago are quickly being adopted into common use. Equipment for these services requires microprocessors inside and can be regarded as embedded systems. Embedded systems are computer systems that are well hidden inside devices. At the time of design, much is known about the operating conditions and requirements. Embedded systems are designed to meet these requirements at a minimal cost. To improve efficiency and throughput, real-time operating systems (RTOSs) can be used. However, RTOSs can create overhead in systems. Using hardware accelerators can significantly reduce overhead. In this work, we survey the major overhead in embedded systems and identify and analyze some of them in detail. We then propose and discuss nanoproces-

sors, as configurable hardware accelerators, to lower this system overhead. Our simulation result shows that nanoprocessors can improve system performance at a nominal cost.

NANOPROCESSORS: CONFIGURABLE HARDWARE ACCELERATORS FOR EMBEDDED SYSTEMS

by

Lei Zong

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2003

Advisory Committee:

Professor Bruce L. Jacob, Chair
Professor Manoj Franklin
Professor Donald Yeung

©Copyright by

Lei Zong

2003

DEDICATIONS

To Luke, thanks for sharing the experience.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Bruce Jacob for his advice and help during the past couple of years. I have learnt a great deal from him. I would also like to thank everyone in the SCAL computer lab. You made it a fun experience for me.

TABLE OF CONTENTS

List of Tables vii

List of Figures viii

1 Introduction 9

1.1 Overview 9

1.2 Background Information 10

1.2.1 Modern Embedded Systems 10

1.2.2 Real Time Operating Systems 13

1.3 Nanoprocessors 16

1.4 Related Work 20

2 Overheads in Embedded Systems 24

2.1 High-Overhead Serial Input/Output(I/O) Port 24

2.2 Process Scheduling Overhead in Embedded RTOS 26

2.2.1 Task Scheduling 26

2.2.1.1 Static Scheduling 27

2.2.1.2 Dynamic Scheduling 28

2.2.2 Overhead of Dynamic Scheduling 32

3 Nanoprocessors as reconfigurable logic 34

3.1 Overall System Design 34

3.2 Reconfigurable Logic 37

4 Nanoprocessor as I/o controller 40

4.1 High-Level Functional Description 40

4.2 Design Issues 42

4.3 NanoI/O Controller Interface 45

4.4 Implementation Details 48

5 Nanoprocessor as a task scheduler 55
5.1 System Design 56
5.2 High-Level Functional Description 57
5.2.1Software Scheduler Functionality 57
5.2.2NanoScheduler Functionality 64
5.3 NanoScheduler Task Structure 67
5.4 NanoScheduler Interface 71
5.5 Implementation Details 73
6 Experimental methods 84
6.1 Performance Simulator 84
6.1.1Processor 85
6.1.2Operating System 87
6.1.2.1Processor Scheduling 88
6.1.3Benchmarks 89
6.1.3.1ADPCM 90
6.1.3.2GSM 90
6.1.3.3Patricia 91
6.1.3.4Dijkstra 91
6.1.3.5SHA 91
6.1.3.6Bitcount 92
6.1.3.7G721 92
6.1.3.8Homemade Benchmark 92
6.1.4Tasks 93
6.1.5Measurements 93
6.2 Cost Simulator 95
7 Results for nanoprocessor as i/o controller 101
7.1 System Bandwidth Analysis 101
7.2 Processor Utilization Analysis 106
7.2.1Future Optimization 110
7.3 NanoI/O Controller Cost 113
7.3.1Die Area Analysis 113

7.3.2Power Consumption Analysis....................................... 115

8 Results for nanoprocessor as scheduler.....................118

8.1 Schedulability Analysis.. 118

8.2 System Bandwidth Analysis 122

8.3 Processor Utilization Analysis 124

8.4 Die Area and Power consumption Analysis........................ 129

9 Conclusion..131

9.1 Summary 131

9.2 Future Work.. 134

Bibliography 136

LIST OF TABLES

Table 4.1:I/O Operation Overhead as a Function of the Requested Data Size 41

Table 5.1:Functionality of the OS with the nanoScheduler for handling task state transitions...64

Table 5.2:NanoScheduler Data Structure Fields and Their Explanations70

Table 6.1:Synthesis Conditions..97

Table 6.2:Switching Probability of a NAND and a NOR gate 99

Table 6.3:Power-Specific Unit Parameters 100

Table 7.1:Area Estimate for the nanoI/O Controller...............................115

Table 7.2:Power Estimates for the nanoI/O controller............................116

Table 8.1:Die Area Estimate for Hardware Scheduler............................129

Table 8.2:Power Consumption Estimate for Hardware Scheduler 130

LIST OF FIGURES

Figure 1.1: ---- Altera Excalibur EPXA 10 SoC with ARM922T microprocessor. 18

Figure 2.1: ---- two tasks with different time requirements.....29

Figure 2.2: ---- running the static priority-base scheduling 30

Figure 2.3: ---- running the EDF scheduling algorithm.....31

Figure 3.1: ---- system block diagram with 4 nanoprocessors......36

Figure 3.2: ---- Memory Map incorporating nanoprocessors 37

Figure 4.1: ---- System Operation Diagram for I/O Transactions.....41

Figure 4.2: ---- System Operation Diagram for I/O Transactions using the nanoIO Controller.....42

Figure 4.3: ---- memory layout for the nanoI/O controller 45

Figure 4.4: ---- State Diagram for the nanoIO Controller.....49

Figure 4.5: ---- IOinit maps the I/O port(s) into tables 50

Figure 4.6: ---- Function Flow Diagram for IOread 51

Figure 4.7: ---- Function Flow Diagram for IOwrite 52

Figure 4.8: ---- Functional Flow Diagram for the nanoI/O ISR 53

Figure 5.1: ---- State Transition Diagram for Tasks in mCOS.................58

Figure 5.2: ---- TCBList as a doubly-linked list 58

Figure 5.3: ---- TCBPrioTbl stores the addresses of the TCBs 59

Figure 5.4: ---- Ready List Implementation in mCOS.............................60

Figure 5.5: ---- NanoScheduler Data Structure.......................................69

Figure 5.6: ---- Address Map for NanoScheduler Control Registers.......71

Figure 5.7: ---- Function Flow Diagram for NanoCreateTask.................74

Figure 5.8: ---- Function Flow Diagram for NanoDeleteTask.................75

Figure 5.9: ---- Function Flow Diagram for NanoChangeStateWait. 76

Figure 5.10: ---- Function Flow Diagram for NanoTimeDlyTask. 77

Figure 5.11: ---- nanoScheduler's actions for task management. 78

Figure 5.12: ---- Function Flow Diagram for Resuming Delay Task. 78

Figure 5.13: ---- Function Flow Diagram for NanoTimeTick, without reorder operation 80

Figure 5.14: ---- Flow Diagram for the List Reordering Operation of the NanoTimeTick function...81

Figure 5.15: ---- Function Flow Diagram for NanoGetHighID...............82

Figure 6.1: ---- Simulation Environment Block Diagram........................85

Figure 6.2: ---- A 16 x 16 register file representation.............................98

Figure 6.3: ---- M*CORE die photo. 99

Figure 7.1. ---- Maximum System Bandwidth Comparison for 6 benchmarks.....102

Figure 7.2. ---- Processor Utilization Rate Breakdown for benchmarks .107

Figure 7.3. ---- Possible optimization to improve the processor utilization rate for single nanoI/O controller and UART.....111

Figure 7.4. ---- Configuration for a single nanoI/O controller and multiple UARTs 112

Figure 8.1. ---- Workload scheduling activity using the static-priority scheduler.....119

Figure 8.2. ---- Workload scheduling activity using the EDF scheduler .120

Figure 8.3. ---- System Bandwidth Comparison using Different Scheduling Schemes.....123

Figure 8.4. ---- Processor Utilization Comparison for different scheduling schemes.....126

CHAPTER 1

INTRODUCTION

1.1 Overview

Today's consumer market is driven by technological innovations. Many technologies that were not available a few years ago are quickly being adapted into common use. These technologies include DVD, GPS, streaming multimedia, broadband services like high-speed cable and DSL. All equipment for these services require microprocessors inside and can be regarded as embedded systems.

Embedded systems are computer systems that are well disguised and hidden inside devices. They normally perform specific, well-defined functions, and are thus designed with these functions in mind. This is in contrast to a general-purpose computer, which is designed to be highly flexible.

Because prior knowledge about performance requirements is available, embedded systems are designed to meet these requirements at a minimal cost. To improve efficiency and throughput of these systems, real-time operating systems (RTOSs) can be used. RTOSs are good for performance;

however, they can create additional work for the system. Any work that is unrelated to user applications is considered as an overhead and can incur a higher cost for the system. Overheads can be significantly reduced by using hardware accelerators. In this work, we survey the major overheads in embedded systems and identify and analyze some of them in detail. We then propose and discuss *nanoprocessors*, as reconfigurable hardware accelerators, to lower overhead. Our simulation results show that using nanoprocessors can improve system performance at a minimal cost.

1.2 Background Information

1.2.1 Modern Embedded Systems

The embedded systems market is one of the fastest growing markets of this decade. These systems have been deeply "embedded" into daily lives in many ways without us being aware of them. Unlike a desktop computer, embedded systems do not take the form of a tower chassis standing on a desk. On the contrary, they are well hidden inside many devices, such as cell phones, digital cameras, game consoles, MP3 players, PDAs, digital TVs, etc. The use of embedded systems has increased tremendously in the recent past, and the growing trend is likely to continue in the future.

Embedded systems can take on many different forms, but the components inside are somewhat similar across all systems. Most systems include

at least one microprocessor as the central processing unit (CPU), along with on-chip memory for fast access time. The size of memory can differ significantly across systems. In addition, there is normally a hardware timer, a watchdog timer, Input/Output (I/O) ports such as UARTs (Universal Asynchronous Receiver/Transmitter), and sometimes a debugging interface.

Depending on specific application requirements, other components can also be found. For instance, if an embedded system is expected to perform signal processing tasks, a multiply-and-accumulate (MAC) unit can be integrated for that purpose. Embedded systems in cell phones have cellular network communication modules included. Embedded processors that form a network are likely to have DMA controllers and network interface controllers. In addition, other peripherals such as multimedia cards, wireless adapters and infrared ports can also be found in embedded systems. Because the embedded applications are becoming more diverse, more application-specific hardware components are founded in embedded systems.

Some of the applications that run on embedded systems are real-time, meaning that if the applications do not execute within a fixed amount of time, they can result in a catastrophe. For example, in an embedded system that is used in airplanes, if an abnormal reading on a sensor is not reported

in a timely manner, it could damage the airplane and even put people in danger. Because of special requirements of this nature, most embedded systems avoid non-deterministic hardware. One good example of this is the absence of caches in most embedded systems. Caches operate on locality of data access. They can service a very high percentage of data traffic, and shorten the average access time. However, no cache can guarantee hits on all data accesses, thus it does not change the worst case access time. To meet real-time requirements, the worst case scenario must always be considered. Therefore caches are typically omitted from embedded systems. In addition this provides savings in die area and power consumption.

Power consumption is a critical design issue for embedded systems that are powered by batteries. The resources available for a system are limited by power constraints. For instance, embedded systems cannot increase system memory size at will due to the increased power consumption from memory cells. As a result, embedded system designers are generally minimalists. For example, most embedded systems use serial I/O ports instead of parallel I/O ports. The advantages of using serial ports are that they require fewer output pins, which reduces power consumption. Also they are simple to design and can transmit data over a longer distance. Even though serial ports can hardly match the bandwidth provided by parallel ports, most applications can be satisfied with serial port performance [2].

In most cases UARTs are used with serial ports. UARTs convert the parallel outgoing data from the processor into a serial bit stream and pass it to the serial ports. They also perform the serial to parallel conversion for incoming data. From the processor perspective, data communication is done in bytes.

1.2.2 Real Time Operating Systems

As the market expands and new services are added, the complexity of embedded systems grows. The modern day embedded market has expanded from 8-bit microprocessors used 30 years ago to 16-bit and 32-bit microprocessors common in today's market. The memory size of microprocessors is also on the rise to accommodate larger application code footprints. In addition, as the market becomes more competitive, the system development time becomes shorter. This combination of increasing complexity and short time to market has led designers to use third party commercial RTOSs. RTOSs offer valuable services to embedded system designers and help to save time in the production cycle. The benefits of RTOSs translate directly into profits. Thus, an increasing number of RTOSs are being used in embedded systems.

Like general operating systems (OSs), RTOSs provide users with desirable functionality such as multitasking, interprocess communication

(IPC), process scheduling, etc. RTOSs provide efficient algorithms and maximum optimization for these services.

Multitasking is arguably the most important feature of an RTOS. By allowing several tasks to run simultaneously, tasks can complete faster than sequential execution. Often a task runs for a period of time and waits for input for a period of time. While one task is waiting, another task can take advantage of the idle CPU. By interleaving tasks, an OS can increase processor utilization rate, or the amount of time a processor spends doing useful work. By efficient scheduling, an OS can reduces the overall amount of time to execute several tasks.

However, in order to keep track of several tasks at the same time for multitasking, the OS becomes more complex. Switching between tasks, also called context switching, must be done at least occasionally. A context switch is an expensive operation because all information about the current task must be saved from the CPU registers, and information about a new task must be copied into the CPU registers.

When a task gives up the CPU, an OS needs a way of finding the next available task to take over. This calls for a task scheduler. The task scheduler is at the heart of a operating system. There are two classes of scheduling algorithms: static scheduling and dynamic scheduling. Static scheduling algorithms assign task priorities at compile time. Throughout

execution, a task keeps the same priority with respect to all other tasks. For instance, a round-robin scheduler treats all tasks equally, allowing each task to execute for a fixed amount of time in a round-robin fashion. The static priority-based scheduler always executes the task with the highest priority. If a task with lower priority is running, it is interrupted to allow the higher-priority task to run first. Dynamic scheduling algorithms, on the other hand, take into consideration the runtime characteristics of the task when making decisions. Priorities with respect to all other tasks can change during execution. Earliest Deadline First (EDF), a dynamic scheduling algorithm, makes scheduling decisions based on the deadline value at the scheduling instance. Therefore, the order of execution could always be changing. The task having the smallest deadline value has the highest priority and is executed first. More about scheduling is discussed in detail in chapter 3.

However, these desirable services of an OS come at a cost. For instance, when a context switch occurs, the time spent performing the switch is in addition to the time spent executing user tasks. Therefore, context switching is considered an overhead. Depending on application loads, context switching can be a huge overhead that prohibits good performance. In this study, we investigate hardware means to reduce the overhead incurred in the RTOSs.

1.3 Nanoprocessors

There are many different peripherals existing in embedded processors. Each brand of processors has a different communication standard for on-chip peripherals. This makes migrating application code difficult across different embedded platforms. In addition, if an application has specific needs, an existing embedded system might not be able to satisfy the needs due to limited resources. In the case of a cell phone system, if the module that supports the voice encoding and decoding algorithm is not available in hardware, then it must be implemented in software. This would mean that the algorithm takes longer to execute and can shorten the battery life.

Further, software OS overhead can create performance bottlenecks for embedded systems. Other OS features can be improved. The 90-10 rule of software engineering states that 10% of the code accounts for 90% of execution time [27]. If this code segment can be identified within a system, it can be implemented in hardware for faster execution, thereby eliminating the performance bottleneck.

In this work, we propose nanoprocessors as reconfigurable hardware components to off-load some of the repetitive tasks an operating system performs. Nanoprocessors are designed to increase performance without increasing power consumption and die area significantly. They are proposed to be small finite-state machines residing close to a processor.

Depending on the application needs, the nanoprocessors can be configured for different purposes. If the application requires fast speech encoding, a nanoprocessor can be configured as a hardware voice codec. If the application requires precise time management, a nanoprocessor can be configured to be a hardware timer. Nanoprocessors are peripheral modules that can be configured to meet specific application needs.

Programmable I/O devices are reality in today's technology. Using programmable I/O devices allow designers to interface multi-standard devices. Since many I/O standards exist and each has a unique set of requirement, programmable I/O devices can simplify designs dramatically and shorten the time to market. Inspired by the programmable I/O devices, the goal of the nanoprocessors is to investigate uses for programmable logic other than I/O devices. For instance, in this work, we have identified an I/O controller and a hardware scheduler suitable for programmable logic.

Since nanoprocessors are envisioned to be on-chip reconfigurable peripherals, it is possible to implement the processor and nanoprocessor together on a on-chip FPGA. Recently many kinds of on-chip FPGAs have been made commercially available. Companies such as Altera, Xilinx, Actel, Amtel, and others now provide System-on-Chip (SoC) solutions. The on-chip FPGAs include a fixed microprocessor and programmable

logic on the same chip. The figure below shows the Altera Excalibur on-chip FPGA die photo [21]. This particular chip includes an ARM922T microprocessor and various amount of programmable logic. Using on-chip FPGAs, the nanoprocessors can be realized in the reconfigurable logic. More about reconfigurable logic is discussed in Chapter 2.

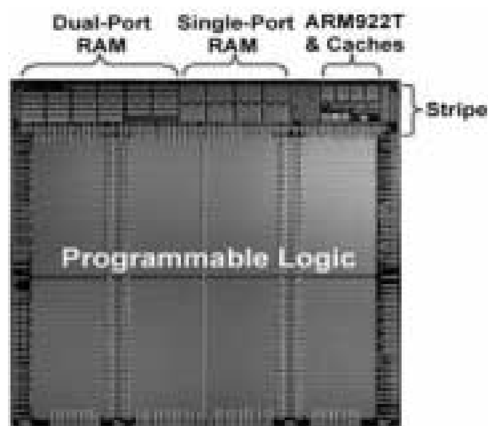


Figure 1.1: Altera Excalibur EPXA 10 SoC with ARM922T microprocessor.

There are many uses for nanoprocessors. They can be used to implement application-specific peripherals, or programmed to target system overhead encountered. In this work, we investigate moving two RTOS subsystems, which cause performance bottlenecks, viz. I/O ports and task scheduler, to nanoprocessors. If there is a large amount of data expected from external devices, a nanoprocessor can sit between an I/O port and the processor to control I/O traffic. In particular, it can save the external data into a specific location in memory. When the buffer is full, the nanoproces-

sor sends an interrupt to the CPU. Essentially, the nanoprocessor is acting as a Direct Memory Access (DMA) controller. As a result, the CPU is only interrupted once when the buffer fills up and can spend more time executing tasks.

Another performance improvement identified is the task scheduling scheme. It is apparent that dynamic scheduling has a significant advantage over static scheduling. The major hurdle in implementation is the prohibitive cost of software. Therefore, we propose a nanoprocessor as a hardware scheduler. Both static priority scheduling and dynamic EDF scheduling schemes are implemented on the scheduler to demonstrate its flexibility. Since scheduling is done in hardware, the results are returned much faster.

The rest of the document is organized as follows. Chapter 2 discusses RTOSs in detail, identifying bottlenecks for performance. Specifically, the I/O overhead and the process scheduling bottleneck are discussed. To eliminate bottlenecks, nanoprocessors are proposed. Chapter 3 introduces the nanoprocessors from a high-level perspective, discussing possible implementations using existing technologies. Chapter 4 and 5 introduce two nanoprocessors, nanoI/O controller and nanoScheduler, targeting the two bottlenecks of an OS. Their designs are explained in detail and their interaction with other system blocks is analyzed as well. Chapter 6 gives an overview of the simulation environment along with the different bench-

marks used as workloads. Chapter 7 and 8 present and analyze results for the two nanoprocessors. Effects on performance, as well as on die area and power consumption are evaluated. Finally, Chapter 9 summarizes the study and indicates future work direction.

1.4 Related Work

Others have investigated moving the OS to hardware entirely or moving partial OS to hardware for efficiency.

A hardware RTOS, called Real-Time Unit, has been developed [11]. It provides services such as scheduling, IPC, time management, etc. Since the RTU is a hardware component, it decreases the system overhead and can improve predictability and response time. In addition, the RTU reduces the memory footprint of the OS. However, moving the entire OS to hardware makes the system less flexible. Extensions to the OS are difficult to add. New chips must be made if extensions are added to the system. For existing embedded systems, major hardware changes are required in order to use the RTU. On the other hand, the nanoprocessors offer the fast execution time in hardware for selected OS services while providing the users the flexibility of choosing which service to perform in hardware. The nanoprocessors allow most of the OS services to remain in software and avoid the upgradability issue.

Furthermore, many researchers have looked into moving the process scheduler to hardware to support a more complex scheduling algorithm. In the field of high-speed networks, scheduling at switching nodes is an important issue since it directly affect the network throughput. Specifically, several priority queue (PQ) architectures exists for finding the highest priority packet to route. These architectures are binary tree of comparators, FIFO priority, shift register, and systolic array[14]. In particular, The shift register is the closest implementation to the nanoScheduler, but there are still differences. The two implementations differ in that the shift register method finds the appropriate location for the new entry in 1 cycle by broadcasting to all entries. It then shifts part of the list to insert a new entry. This implementation results in fast access time, however, because simulataneous operations must be performed for a large list, the power consumption will be high. In general, studies done for network switches do not sacrifice performance for power considerations. In contast, the nanoprocessor study views the power consumption as important as performance gains.

[13] studied a configurable hardware scheduler for real-time systems. The hardware scheduler supports Priority Queue, EDF, and Rate Monotonic scheduling algorithms. The main distinction from nanoScheduler is that their scheduler keeps track of multiple queues. In addition, operations are performed under a costant overhead since a comparator is used for

every element in the ready list. This will cause a dramatic increase in die area and power consumption. The power consumption estimate is not provided with the study, therefore, no comparisons can be made.

Another study done by [15] proposes a reconfigurable co-processor that implements application-specific functionality. The processor is proposed to be implemented on FPGAs. The processor executes custom-instructions, and require a new assembler to convert user applications into custom-instructions. The advantage of the processor is that it provides the maximum flexibility to the users. The nanoprocessors investigates operation sharing on a coarse granularity. An operation is defined as a candidate of a nanoprocessor and that operation is entirely moved to hardware. Custom instructions, on the contrary, shares operations on a finer granularity.

Some work has been done to evaluate the power consumption of RTOS functions. Particularly, the study done by [12] shows measurements for processing time and energy consumption for various RTOS functions. The simulator used is an instruction-level simulation of the Fujitsu SPARClite processor. The study shows that the particular application programming style can affect the power consumption in the system. In the nanoprocesssor study, we assume that the applications are already optimized for low power consumption. The goal of the study is to identify

problematic operations in the OS that are inefficient and to move the operations into hardware to save on power consumption.

CHAPTER 2

OVERHEADS IN EMBEDDED SYSTEMS

Real time operating systems provide numerous services for today's embedded processing needs. These desirable features come at a cost, creating overhead in the system. In this section, we identify two major overheads and discuss the nature of the problems in detail.

2.1 High-Overhead Serial Input/Output(I/O) Port

Frequently, embedded systems are used in portable devices, such as cell phones, game consoles, measurement equipment like voltage meters, etc. These devices operate on batteries, thus often use serial I/O ports (along with UARTs) to save on power consumption. However, I/O ports[1] can create unnecessary overhead. Consider the following example. When an external device wants to communicate to the CPU, it sends input data through an I/O port. A UART converts a serial bit stream on the port side into a byte stream on the processor side. This reduces potential OS overhead by a factor of eight. However, there is still much room for improvement, as data communications are often in packets that are much longer

1.UARTs and serial I/O ports together are referred as I/O ports.

than a single byte. With the present arrangement, every time a byte of data is available at the I/O port, the CPU is interrupted to save the byte into a buffer and then it resumes its original operation. This interruption is expensive since a context switch is involved. Thus the overhead of interrupt handling can be significantly since it includes an interrupt for each byte of data.

It is analogous to a merchant who ordered several shipments from different vendors. The shipments can arrive at any time and notices for shipments are sent to the merchant. If there is no storage available at the dock, the merchant must go and pick up the shipments as they come in. If the distance from the merchant and the dock is far, it is easy to imagine a grumpy merchant complaining about all the driving involved, not to mention expense for gas and wear on his automobile. If he can rent a storage room at the dock, shipments can be automatically stored as they come in, and would only require the merchant one trip to pick them all up. It is easy to see that the merchant would prefer the storage room provided the room is not expensive to rent and his shipments are not easily spoiled. The time spent on driving is analogous to the time spent in CPU context switching, which can be better utilized.

2.2 Process Scheduling Overhead in Embedded RTOS

Many embedded applications that are also real-time applications, have strict time requirements. Therefore, a RTOS that manages these tasks[2], should behave in a predictable and deterministic manner[4]. The predictability requirement also applies to the task scheduler.

2.2.1 Task Scheduling

The task scheduler determines which task to run and when. Just like a predictable program, a deterministic task scheduler should return the result within a constant amount of time. Without any prior information, it is difficult to know the next task to run within a fixed timeframe. Consequently, many RTOSs impose certain limitations on tasks to achieve the constant process scheduling time requirement. For example, MicroC/OS-II (μCOS) requires the maximum number of tasks declared to be less than 64. In addition, each task must have its own static priority level for a total of 64 task priorities[3]. With this requirement, a static priority-based scheduler always runs the task with the highest priority. This way, by knowing exactly how

---

2.Tasks, user applications, and processes all indicate the programs running on a system. Therefore, they are used interchangeably, and indicate the workload of a system.
3. In reality, most of embedded systems do not have a large number of tasks defined. Thus, the task limit is not a performance bottleneck.

many tasks exist in a system at any given moment, we can calculate the current running task.

However, static priority scheduling algorithm does not take the task runtime characteristics into consideration, and thus may not achieve the best possible schedule. Other scheduling algorithms that use task runtime characteristics can achieve a better task schedule. However, these scheduling algorithms can incur a higher overhead. In the rest of the chapter, we first discuss in detail how static scheduling achieves a constant overhead in µCOS, then we discuss the advantages of dynamic scheduling, in particular the Earliest Deadline First (EDF) algorithm.

2.2.1.1 Static Scheduling

For optimization, µCOS uses the static priority-based scheduling algorithm mentioned above with a 8 x 8 lookup table. Tasks are grouped into 8 groups and each group consists of 8 tasks. The ready bits of the tasks are stored into the 8 x 8 table, where row numbers indicate group identifiers (IDs), and column numbers indicate task IDs within a group. The algorithm is performed in constant time; first find the row that contains the highest priority ready task and then the column for that task in the table. Both steps involve a deterministic number of instructions which is fixed at compile time. The 8 x 8 table is optimized for 8-bit processors. The same

table structure can be easily extended to a 16 x 16 table for 16-bit processors and systems needing 256 tasks or priority levels. Thus, the table structure is not only simple and efficient for static priority-based scheduling; it is also scalable.

2.2.1.2 Dynamic Scheduling

Static scheduling is simple and efficient, but can fail to produce a schedule even when a feasible schedule exists. Consider the following case where a static priority scheme would fail. Before describing the situation, there are some definitions that need to be presented. *Period* is the amount of time between two consecutive instances of task executions. *Runtime* is the amount of time a task takes to complete one instance of task execution. *Deadline* is the time when a task should complete execution. A task is *late* if it has not completed execution after the *deadline* has passed. A feasible schedule allows the requests from all tasks to be fulfilled before their

respective deadlines. With these terms in mind, below is a sketch of a simple situation where static priority scheme would fail.

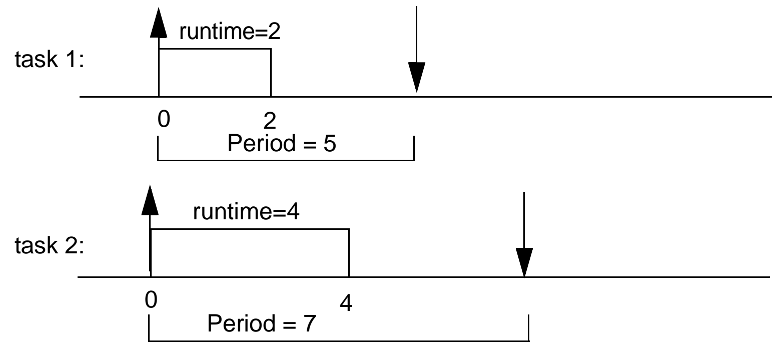


Figure 2.1: two tasks with different time requirements.

For these two tasks, task 1 has a shorter period with a short runtime and task 2 has a longer period with longer runtime. Assume that the deadline criteria has been relaxed. That is, as long as the task finishes execution somewhere within its period, the OS assumes the operation is legitimate. The up arrow indicates where the period starts and the down arrow where the period ends. For simplicity, these two tasks are created at the same time[4]. Under the static priority scheme, typically the task with the smallest period is assigned the highest priority. Once the priorities have been determined, they are fixed with respect to all other tasks for execution. In this case, task 1 is scheduled to execute first. It is not interrupted until completion. When task 1 is finished, task 2 can start at time 2. At time 5, task 1

4.Different creation time may move the delay detection to a different period, but does not change the result.

becomes available to run when its second period starts. It takes over since it is the task with a higher priority. When task 1 finishes at time 7, task 2 resumes execution. However, since task 2 has a deadline of 7, the task is late since its execution is not complete at time 7. The following graph illustrates the scheduling problem. The solid arrow indicates task 1 deadline, the dotted arrow indicate task 2 deadline.

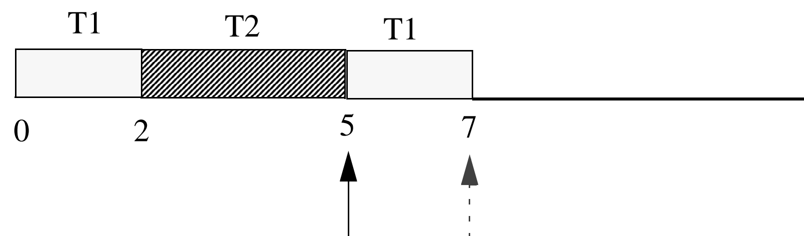


Figure 2.2: running the static priority-base scheduling.

Note that the problem remains even if task 2 were given the higher priority. In this case, task 2 executes first to time 4. Task 1 starts execution then, but fails to finish before the end of its period.

On the other hand, this problem can be entirely avoided if a dynamic scheduling scheme is used. For example, the EDF algorithm figures out the priorities for tasks at runtime depending on their deadline values. As the name suggests, the task with the smallest deadline has the highest priority. In the previous example, the deadline is the same as the period. When both tasks are initialized at time 0, the EDF algorithm decides that task 1 has a

higher priority since its period is 5. After task 1 finishes executing at time 2, task 2 starts execution. At time 5, the period of task 1 expires and starts a new period. Therefore, priorities need to be re-evaluated since the run-time characteristic of task 1 has changed. At this point in time, task 2 has deadline at time 7 and task 1 has deadline at time 10, and thus task 2 continues execution. The figure below shows that at every deadline (indicated by arrows), the priorities are re-evaluated. For instance right before time 15, task 2 is the current task. However, task 1 has a deadline at time 20 and task 2 has a deadline at time 21. Thus task 1 takes the higher priority and executes. As the graph shows, the EDF scheduler can successfully schedule

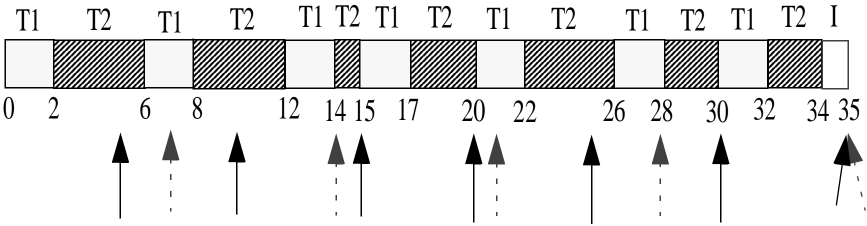


Figure 2.3: running the EDF scheduling algorithm.

both tasks. At time 35, the two deadlines coincide and task 1 has a higher priority since it has a shorter deadline. The graph repeats itself at every 35 time units. In fact, numerous studies have shown that dynamic scheduling algorithms such as EDF are optimal[5] compared to the static priority-based

5.An optimal scheduling algorithm can achieve the best possible schedule under the given circumstances, if such schedule exists. That is, an optimal scheduling algorithm can achieve a processor utilization of 100%.

scheduling algorithms[3]. In the case of two tasks scheduled by the static priority scheme, the best utilization rate[6] is about 83%. In general with a large number of tasks, the static-priority scheme can achieve utilization rate of 69%. However, with the EDF algorithm, it is possible to achieve utilization rate of 100%[3].

2.2.2 Overhead of Dynamic Scheduling

Even though an EDF algorithm can achieve higher schedulability, it is not commonly used in the real-time embedded operating systems. This is because an EDF scheduler is more expensive to maintain. To be able to schedule tasks efficiently, more information about the tasks needs to be available to the scheduler. Therefore, the storage area requirement increases for each task in the system. In addition, the algorithm used in EDF is more complex than a static priority scheduling scheme. Thus dynamic scheduling algorithms would take longer to execute, and might not completed within a constant timeframe. Specifically, EDF sorts the system tasks using deadline values which can range freely, making it difficult to achieve a constant overhead. Hence, it is extremely hard to match the predictable system performance obtained from using static scheduling.

6.Utilization rate is computed as (RUNTIME/PERIOD * 100%).

In summary, using a dynamic scheduling algorithm may increase the processor utilization rate, however, the complexity of a dynamic algorithm may also increase the amount of system overhead.

CHAPTER 3

NANOPROCESSORS AS RECONFIGURABLE LOGIC

Having identified overheads in the RTOSs, we propose nanoprocessors as configurable hardware accelerators to reduce the overheads. In this chapter, we illustrate the overall system design incorporating the nanoprocessors and possible physical implementation using on-chip FPGAs.

3.1 Overall System Design

Embedded processors offer on-chip memory in the form of flash memory and/or SRAM, interrupt controllers, on-chip peripherals such as hardware timers, watchdog timers and I/O ports. Like on-chip peripherals, nanoprocessors are located on the same die as the processor. A processor can support a number of nanoprocessors depending on application needs. Nanoprocessors can have many uses in a system. For instance, if the system expects large amount of external data, a couple of nanoprocessors can be configured as I/O controllers. A system that supports signal processing unit can configure one nanoprocessor as a MAC unit. In addition, nanoprocessors could also be programmed as peripheral or memory controllers.

The purpose of nanoprocessors is to provide flexibility and convenience to users.

The nanoprocessors are designed to be architecture independent, i.e., they should work with most (if not all) microprocessors irrespective of microarchitectures (or processor pipeline architectures) and Instruction Set Architectures (ISAs). This does not mean that changes can not be made to the processors; extensions to the ISA can be added by introducing extra instructions. However, no ISA extensions are used in this work. Alternatively, the CPU controls the nanoprocessors through memory-mapped ports. In addition, no change is made to the processor microarchitecture. This is a desirable feature, making the nanoprocessor concept universal. As long as the interface to the system is kept the same, the processor itself can be changed without affecting the entire system.

To incorporate nanoprocessors, the system architecture is slightly modified. A second memory bus is added to connect the nanoprocessors and the main memory. All nanoprocessors share this bus. In addition, all nanoprocessors are connected to the peripheral bus to access the interrupt

controller. Each nanoprocessor can have a different interrupt. The following system block diagram illustrates how nanoprocessors are envisioned.

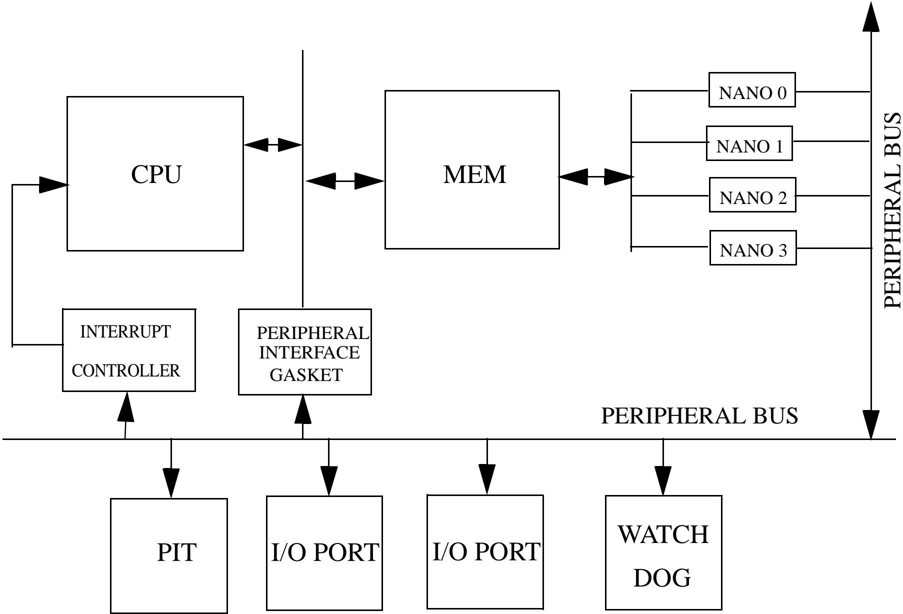


Figure 3.1: system block diagram with 4 nanoprocessors.

The interrupt controller multiplexes all peripheral interrupts to the processor. Each peripheral uses a different interrupt. The interrupt controller can support up to 32 distinct interrupts. The peripheral interface gasket is a simple controller that allows the processor to communicate with the peripherals. All on-chip peripherals and I/O ports are connected to the peripheral bus.

A memory segment is allocated for each nanoprocessor. Within each memory range, there is at least one control register dedicated for communication between a nanoprocessor and the CPU. When the CPU wants to

send a request to a nanoprocessor, the CPU writes to the appropriate control register. The nanoprocessor polls on the register for new work. When the nanoprocessor completes execution for the request, it can either return the results directly or sends an interrupt to inform the CPU. The following diagram shows a memory map incorporating nanoprocessors. Even though the memory ranges for the nanoprocessors appear to be contiguous in the memory map, they might not reside in the main memory. It is possible that the memory resides inside the devices.

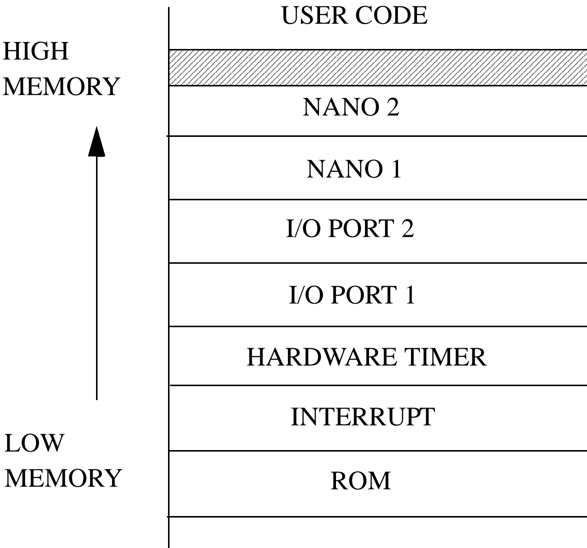


Figure 3.2: Memory Map incorporating nanoprocessors.

3.2 Reconfigurable Logic

As mentioned in chapter 1, nanoprocessors can be implemented as on-chip FPGAs. On-chip FPGAs are often included in Systems-on-Chips

(SoCs). SoCs are embedded systems that are on single chips. They normally include a microprocessor, memory, programmable logic, I/O, and software. The microprocessors used in SoCs can vary, ranging from 32-bit processors to 8-bit microcontroller. FPGAs are good to use because they are relatively easy and straight forward to program. Integrating FPGAs on chip allows the reconfigurable hardware to access main memory with minimum latencies. In many SoCs, the on-chip memory is dual-ported; allowing access from the processor and the programmable logic simultaneously. Certain on-chip FPGAs also include dedicated memory to service the programmable logic in addition to the dual-ported memory[20]. In all, incorporating on-chip FPGAs with a SoC provides users with both performance and flexibility.

The SoC not only has a good performance, but also maintain a low power consumption. Studies have shown that power consumption from using on-chip FPGAs is comparable to that from using ICs only [22]. For several benchmarks, software critical-loops, which are the code that is responsible for a large portion of execution time, are moved to on-chip FPGAs. As a result, the program executions achieve speedups ranging from 1.5 to 3.0. In addition, the measured power consumptions for an ASIC-only chip and a combined ASIC and FPGA chip are very similar. Consequently, the system can achieve dramatic energy savings by using

on-chip FPGAs due to shortened execution time. This study shows that on-chip FPGAs allow systems to exploit hardware parallelism and still maintain a low power consumption level.Therefore, on-chip FPGAs are good candidates for implementing nanoprocessors.

In the following chapters, we identify a couple of operating system performance bottlenecks, provide solutions using nanoprocessors, and present results from our simulation.

CHAPTER 4

NANOPROCESSOR AS I/O CONTROLLER

The overhead due to the RTOS which were identified in chapter 2 are undesirable as they take away processor time from doing application-related work. However, the services provided by RTOSs can increase the system performance and flexibility, thus making them desirable in embedded systems. The purpose of nanoprocessors is to reduce the overhead incurred in providing these services. In this chapter, we discuss the detailed implementation of a nanoprocessor as I/O controller. Similar to a DMA controller, the I/O controller manages data traffic between the I/O ports and the memory.

4.1 High-Level Functional Description

Embedded systems support I/O transactions such as reading and writing to an I/O port. The interface of these transactions exists in the form of system library calls. When such transactions are invoked, the CPU is involved in transferring data bytes between the memory and the I/O port. In the case of reading from an I/O port, the CPU gets a data byte from the I/O

port and saves it to the memory. This operation is repeated as many times as the requested data size. The figure below illustrates the transaction.

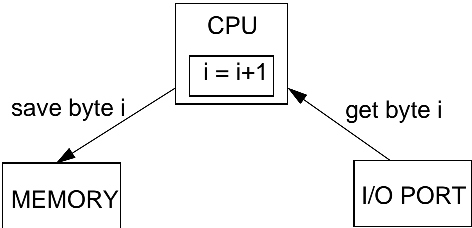


Figure 4.1: System Operation Diagram for I/O Transactions.

If the request packet size is large, this operation can be inefficient. For instance, on our simulator, the time spent moving data between an I/O port and the memory can double as the requested data size doubles. The table below shows the operation time in cycles for requests of different size.

Request Size in bytes	Operation Time in cycles
64	4932
128	7677
256	16260

Table 4.1: I/O Operation Overhead as a Function of the Requested Data Size.

To reduce the overhead incurred in I/O transactions, we propose using a nanoprocessor as an I/O controller, i.e. the nanoI/O controller. It is located between the main memory and the I/O ports, directing traffic between the two. The CPU issues the nanoI/O controller a command at the beginning of the transaction. Upon transaction completion, the nanoI/O

controller causes an interrupt and informs the CPU about the end of operation. The system operation becomes the following:

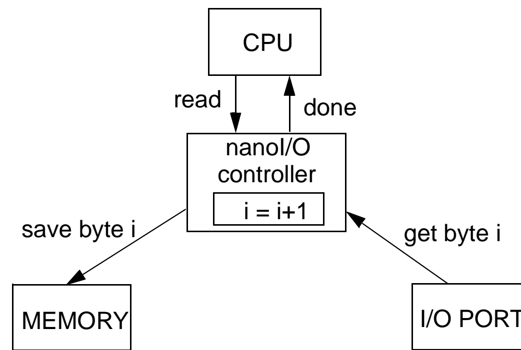


Figure 4.2: System Operation Diagram for I/O Transactions using the nanoIO Controller.

While the nanoI/O controller moves the I/O data, the CPU is freed to operate on other tasks. Our results show that, for tasks that perform much I/O data communication, the benefit of nanoI/O controller is significant.

4.2 Design Issues

In this section, we give a specific example of how I/O transactions work in the system without the nanoI/O controller. The example below demonstrates a request to read from an I/O port.

```
{
    for (i = 0; i < requested data size; i++)
        Disable IRQ;
        *(user buffer + i) = *(I/O port);
        Enable IRQ;
}
```

The I/O read function is implemented as a loop of moving one data byte at a time, from the memory-mapped I/O port to the user declared

buffer in the main memory. The interrupt masking calls (Disable/Enable IRQ) are used to guarantee data correctness. In the case of several tasks sharing an I/O port, data from different streams may become incoherent due to interrupts. Since no virtual memory is used in embedded systems, the memory management is left to users. It is possible that a user task is interrupted while moving data from the I/O port to the user buffer and in turn receives corrupted data. To avoid this situation, interrupts are disabled for data byte during data movement, as indicated by the IRQ commands in the pseudo-code.

An alternative solution is to disable interrupts before entering the loop, as illustrated below in the pseudo-code:

```
{
    Disable IRQ;
    for (i = 0; i < requested data size; i++)
        *(user buffer + i) = *(I/O port);
    Enable IRQ;
}
```

Interrupts are turned off only once under a constant overhead. However, the amount of time the OS spends in non-preemptive mode grows linearly with the amount of data communication. This can potentially disable preemption if the data size is large. Also, disabling interrupts for a long time can result in increased interrupt latency. Both complications are not desirable in RTOSs. For instance, higher priority tasks can not use the system if a lower priority task is occupying the CPU to access I/O data. In this

case, the OS can not context switch to a higher priority task if interrupts are disabled during I/O operations. It would create a priority inversion problem.

The first implementation is more desirable than the second implementation in the sense that interrupts are disabled for a shorter amount of time. However, the high overhead incurred in manipulating interrupts can counteract this advantage. On the other hand, the nanoI/O controller performs the I/O transaction in hardware and lowers the interrupt overhead, making the first implementation feasible. Thus, I/O transactions are implemented using the first approach in this work.

4.3 NanoI/O Controller Interface

The following figure shows the memory configuration of the nano-processor registers.

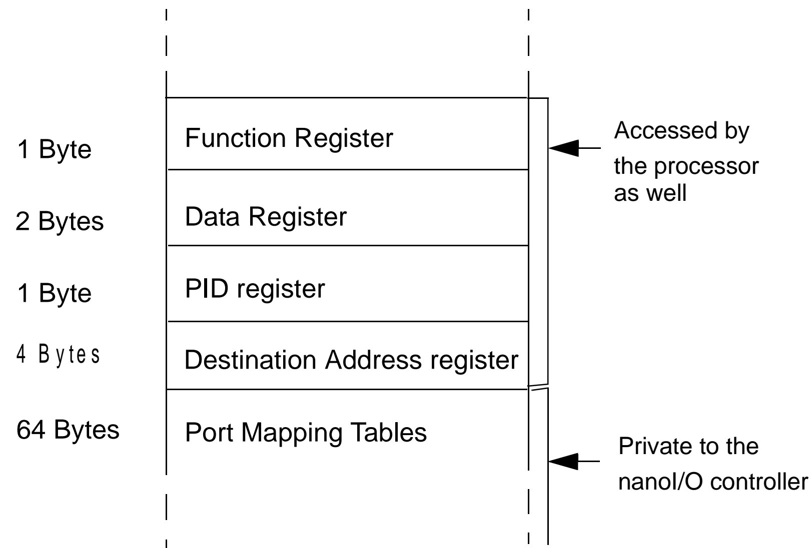


Figure 4.3: memory layout for the nanoI/O controller.

- Function Register is a 1-byte register used for passing function request to the nanoI/O controller. The request states originated from the processor can include IOread, IOwrite, IOgetInt, IOputInt, and IOinit. IOread and IOwrite are requests to read from and write to a specific I/O port. IOgetInt and IOputInt are special read and write functions where the requested data length is fixed at an integer size. These functions are designed for fast data access with a minimum overhead. IOinit registers the task to the nanoI/O controller at the first instance of I/O communication request. Two request states, IOread_done and IOwrite_done, are allowed

only if they are originated from the nanoI/O controller. They are the states that signals the CPU the requested I/O transaction is completed. When the nanoI/O controller is not in any of the above states, it is in the Idle state. These states are discussed in more details in the next section.

- Data register is a 2-byte register holding the request length in the cases of IOread and IOwrite. The maximum allowed length is a 16-bit integer.

- PID register is a 1-byte register holding the task priority for the current requesting task.

- Destination Address register is a 4-byte register holding the 32-bit memory address of the user buffer. The address is the destination in the cases of IOread and IOgetInt, or the source in the cases of IOwrite and IOputInt.

- The input and output port mapping tables are used to store task-to-port mappings. Each table is 32 bytes. The tables are indexed by the task ids, containing 64 entries. Thus each entry is 4-bit long, capable of supporting 16 I/O ports. In addition, a task can have different input and output ports.

The nanoI/O controller takes a total of 72 bytes. Out of the total storage requirement, 64 bytes are for mapping tables. The size of the mapping

tables can be changed to support a different number of I/O ports. The effect on die area is quantified in chapter 7.

When the OS issues the nanoI/O controller to read from the I/O port, the software writes the command into the control registers shown above. Instead of the loop implementation shown in section 4.2, the I/O read function call becomes:

```
{
    Disable IRQ;
    *(NANO_FUNC) = IO_READ;
    *(DEST) = user buffer;
    *(DATA) = requested data size;
    *(PID) = task id;
    Enable IRQ;
    block_self();
}
```

The above pseudo-code shows that the requesting task asks the nanoI/O controller to complete an IOread in hardware. The address of the user buffer, the requested data length, and the calling task id are also provided. The calling task is then blocked waiting for the controller to finish execution. During the waiting period, the CPU can execute other tasks. When the transaction finishes, the nanoI/O controller raises an interrupt to wake up the process that has been waiting for the transaction. The process then exits the I/O read call and returns to user code.

This I/O transaction now operates within a near constant time. The time taken to issue the command is constant. The amount of time waiting

for the nanoI/O controller can vary depending on the amount of data requested. However, since the reading is handled in hardware, the growth in time is much slower compared to that for the original software loop implementation. Thus the total execution time for the I/O read using the nanoI/O controller is near constant.

Even though the nanoI/O controller is introduced in hardware, the interfaces these functions provide to the users are not changed for portability reasons. In another word, a user application that calls IOread would not know about the existence and the usage of the nanoI/O controller. The details about the command issued to the nanoI/O controller are hidden within the system call. Each I/O transaction the nanoI/O controller supports is discussed in detail in the next section.

4.4 Implementation Details

The nanoI/O controller supports the following I/O transactions, IOread, IOwrite, IOgetInt, IOputInt. IOgetInt and IOputInt are special cases of IOread and IOwrite, where the requested data size is an integer. Since the size is small, the cost of blocking the task outweighs the benefit of the performing the function in hardware. Thus for these two transactions the CPU doesn't block the calling task and waits for the result to return. In addition, when the task requests I/O transaction for the first time, a special

function (IOinit) is called to set up the task-to-port mapping tables for the calling task. The state diagram below illustrates the operations among all states supported by the nanoI/O controller.

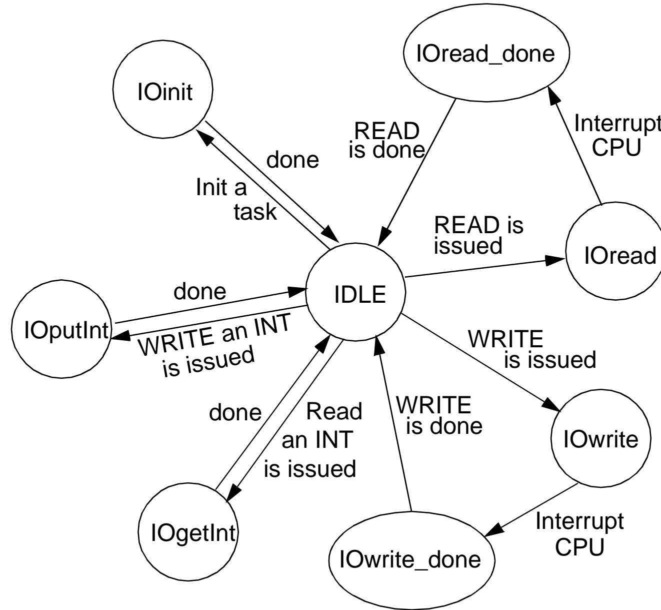


Figure 4.4: State Diagram for the nanoIO Controller.

Each state is described in detail below:

IOinit: Associates an I/O port with the calling task. There are two tables managed by IOinit. One is for input ports and the other is for output ports. The maximum number of tasks supported is 64 as defined in the OS. Therefore, each table has 64 entries. Each entry in the tables is an I/O port number. The input port and the output port for the same task do not have to be the same. An IOinit request is accompanied by the desired I/O ports and the task id. The nanoI/O controller uses the id to index into the two tables

and saves the I/O port at the appropriate entries. Because the operation is relatively short and is performed in hardware, the current task does not block and waits for the operation to finish.

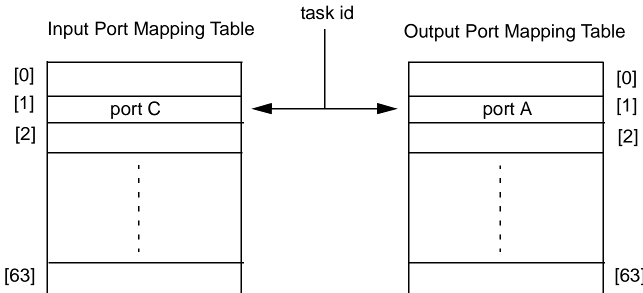


Figure 4.5: IOinit maps the I/O port(s) into tables.

IOread: Moves input data from the input port to the main memory. The current task issues the request and provides the necessary data such as the length to read, the destination buffer address in memory, and the task id. The task is then blocked, and another ready task can be executed. The nanoI/O controller uses passed-in task id to index into the mapping tables for the appropriate input port. When the data of requested size is moved, the function register is changed to IOread_done (see below) and an inter-

rupt is raised. This informs the CPU that service is required to the nanoI/O controller.

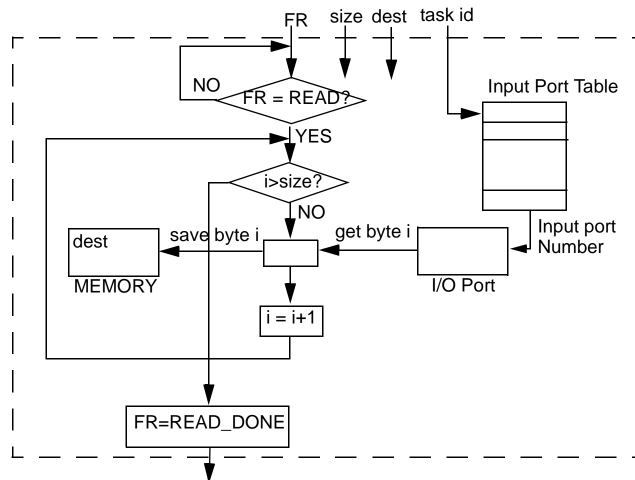


Figure 4.6: Function Flow Diagram for IOread.

IOwrite: Moves a list of items, each of a fixed size, from the main memory to the output port. The output port is looked up from the output mapping table using the task id. Operation is similar to IOread, where the requesting task is blocked until the operation is finished. Upon completion,

the function register state is changed to IOwrite_done and the same interrupt is raised in a similar manner mentioned above.

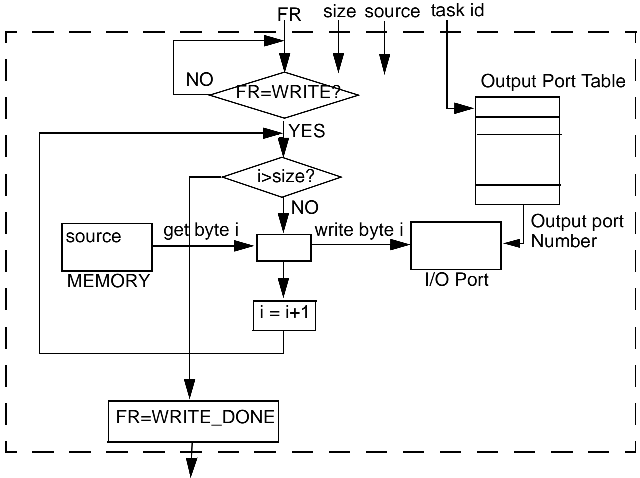


Figure 4.7: Function Flow Diagram for IOwrite.

IOgetInt: Moves a 32-bit integer from the input port to the main memory. The input port is looked up in a similar manner as in IOread. This function is used in some benchmarks. Because each function call only allows one 32-bit element to be returned, the function does not block the current task in order to save on context switching cost. The function flow diagram is similar to that of the IOread, except only one 32-bit integer is read from the input port into the memory.

IOputInt: Moves a 32-bit integer from the main memory to the output port. Similar to IOgetInt, the function does not block the calling task and waits for the result to return. The function flow diagram is similar to

that of the IOwrite, except only one 32-bit integer is written from the memory to the output port.

IOread_done: Signals the CPU that IOread function is complete. The ISR that services the nanoI/O controller interrupt recognizes this state as an indication to wake up the task that has been waiting for the I/O read transaction.

IOwrite_done: Signals the CPU that IOwrite function is complete. The ISR that services the nanoI/O controller interrupt recognizes this state as an indication to wake up the task that has been waiting for the I/O write transaction.

The nanoI/O controller has one ISR routine in software. When the nanoI/O interrupt occurs, the CPU executes this ISR. The routine looks at the function register for the state of the controller and releases the appropriate task.

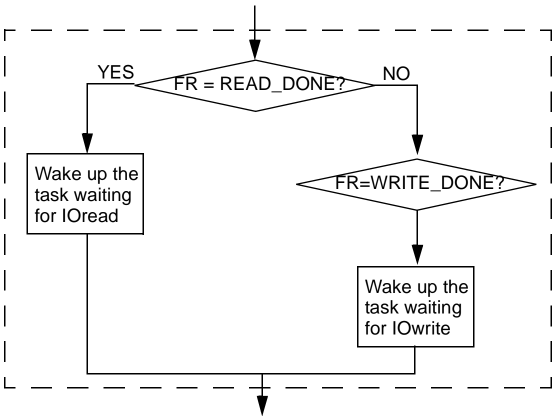


Figure 4.8: Functional Flow Diagram for the nanoI/O ISR.

To summarize, the nanoI/O controller uses 72 bytes of memory to manage I/O communications for all tasks. When a task makes an I/O request that is longer than 32-bit, the task is blocked while waiting for the nanoI/O controller to perform the operation. When the nanoI/O controller finishes the request, it interrupts the processor to wake up the calling task. By using the nanoI/O controller, the CPU remains free and is able to execute applications. This results in a reduced system overhead and increases system efficiency.

CHAPTER 5

NANOPROCESSOR AS A TASK SCHEDULER

As discussed in chapter 2, the static priority-based scheduling scheme in software offers predictability and simplicity, but sacrifices high processor utilization rate. A static scheme cannot schedule certain tasks with different runtime characteristics. On the other hand, a dynamic scheduling scheme, such as Earliest Deadline First (EDF), can achieve the feasible schedule. However, the algorithm itself is more complex than a static priority-based one and may not achieve a constant overhead. Thus, to implement such a scheduling algorithm in software would mean much more overhead for the operating system. To reduce the overhead created by an EDF scheduler, we propose a nanoprocessor as a hardware task scheduler (nanoScheduler). The goal is to implement an optimal scheduling algorithm in hardware while keeping the cost of such implementation reasonable. Hardware implementation is more attractive because it can achieve better cycle time and extract more parallelism from the algorithm. The nanoScheduler off-loads the scheduling functionality from software to hardware. The proposed hardware scheduler implements both static prior-

ity-based and EDF algorithms. The software can decide which scheduler to use based on the prior knowledge about the workload. The major benefit is that a system can increase processor utilization rate by supporting an optimal scheduling algorithm as an advanced feature. In this chapter, we first discuss the overall system design using a nanoScheduler. Then we analyze the architectural design of such a scheduler.

5.1 System Design

Similar to the nanoI/O controller described in chapter 4, the nanoScheduler is also treated as an on-chip peripheral. It accesses the main memory through the added memory bus. When a context switch occurs, instead of executing the software scheduler, the OS issues a request to the nanoScheduler and waits for the result. In addition, the nanoScheduler also uses memory-mapped ports for communication with the processor.

To lower the complexity of the nanoScheduler, only scheduling elements in the software are off-loaded to the hardware scheduler. Other functionality such as event management is left in software. To be specific, when a task is blocked on a semaphore, the state update is done in hardware. However, the semaphore wait queue is still managed in software.

5.2 High-Level Functional Description

This section discusses the functionality of the nanoScheduler at a high level of abstraction. For comparison purpose, we first discuss the software task scheduler in the original operating system. Specifically, we itemize the steps involved in managing tasks for scheduling. Then, we discuss the OS functionality in the presence of the nanoScheduler. By off loading the scheduler into hardware, the OS is responsible for less work. We then compare the OS functionality in the presence of the nanoScheduler with that of the software scheduler.

5.2.1 Software Scheduler Functionality

The task scheduler is at the heart of an operating system that supports multitasking. It affects all aspects of operations. The task state diagram shown below illustrates the possible state transitions in µCOS. At any

given moment, a task can be in any of the five states. The OS must keep track of the state transitions accurately to make a scheduling decision.

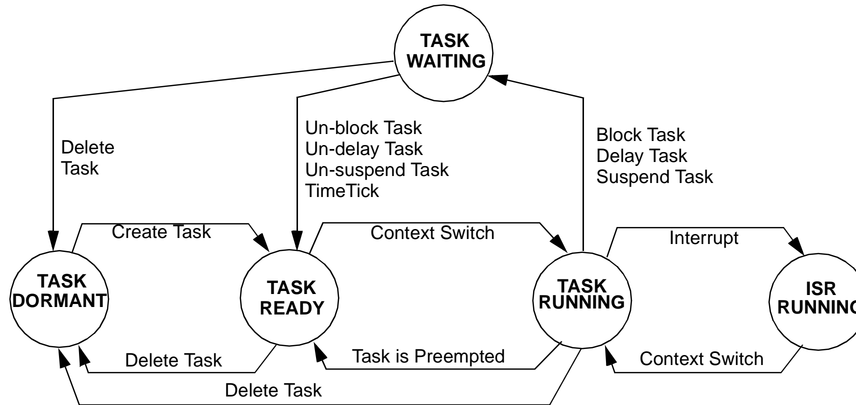


Figure 5.1: State Transition Diagram for Tasks in µCOS.

The OS maintains four structures for tracking tasks.

- A doubly-linked list of Task Control Blocks (TCBs) called `TCB-List`. A TCB is created for each existing task, containing all task information such as priority, delay value, task state, stack pointer, stack size, etc. The newly created TCB block is inserted at the head of the doubly-linked `TCBList`.

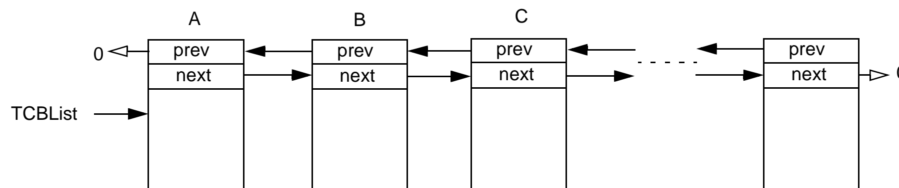


Figure 5.2: TCBList as a doubly-linked list.

- A table of pointers to the TCB blocks called `TCBPrioTbl`. The table is indexed by the task priority. Thus there are 63 entries in the `TCBP-`

rioTbl. When a task is made the current task, the table is referenced to retrieve the TCB block of the task.

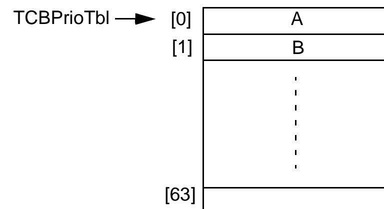


Figure 5.3: TCBPrioTbl stores the addresses of the TCBs.

- A 1-byte register OSRdyGroup to store ready list groups. All tasks in μCOS are grouped by priorities (eight tasks per group) in OSRdyGrp. The group number is obtained by shifting the task priority 3 positions to the right.

- A 64-bit table OSRdyTbl to find ready tasks. Each of the 64 tasks in μCOS is represented by a single bit in this table. The bit location indicates the task priority. For instance, bit 26 represents the task with priority 26. The table is structured as a 8-element array, where each element is a 1-byte register representing the eight tasks in each group. When a task is ready to run, it sets the corresponding bit in both OSRdyTbl and OSRdyGrp. For instance, to make task 26 (11010 in binary) ready to run, the OS first sets bit 3(= 11010 >> 3) in OSRdyGrp. This bit also corresponds to the row position in OSRdyTbl. The column position in the table is the last 3 bits of the task prority (= 11010 & 111). Thus the OS sets

`OSRdyTbl`[3][2] to 1. The diagram below illustrates the relationship between `OSRdyGrp` and `OSRdyTbl`.

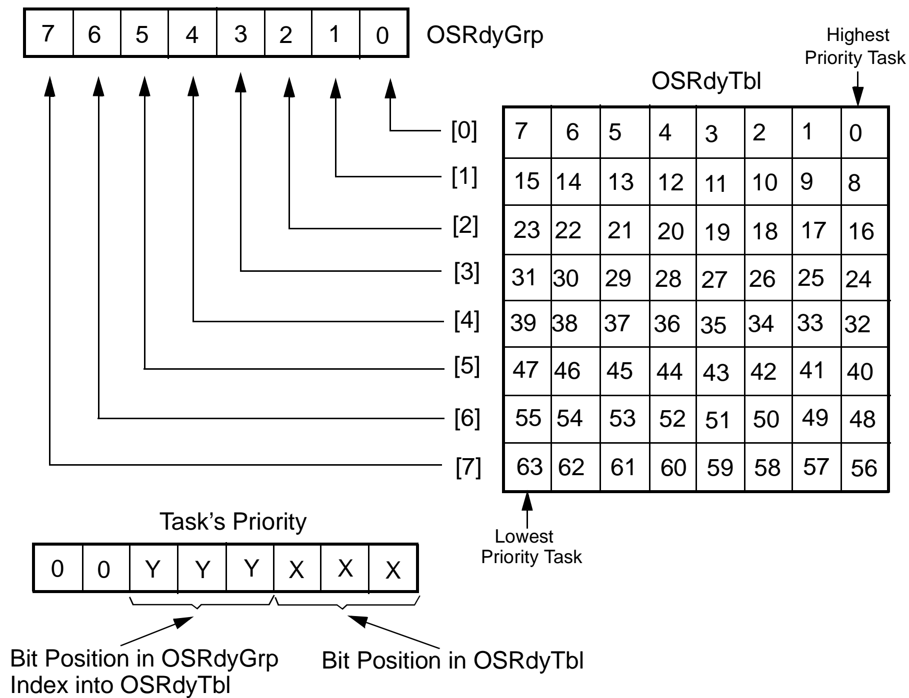


Figure 5.4: Ready List Implementation in μCOS.

To transition from one task state to another, the OS needs to modify the above structures. For instance, when a task is created (transition from the Dormant state to the Ready state), the OS must perform the following steps:

- Create a new TCB and initialize the TCB parameters.
- Insert the TCB to the head of the `TCBList`.
- Store the TCB address to the `TCBPrioTbl` at the location indicated by the task priority.

- Mark the task as ready to run in `OSRdyGrp` and `OSRdyTbl`.
- Schedule the task.

To transition from any other states to the Dormant state, the task delete operation is performed. It reverses the create operation. The OS must follow the steps below:

- Clear the task bits in `OSRdyGrp` and `OSRdyTbl`.
- Remove the task TCB from the `TCBPrioTbl`.
- Remove the task TCB from `TCBList`.
- Clear the TCB and mark it as free.
- Schedule the task.

All of the above steps have constant overheads. The `TCBList` is inserted always from the head. Since the TCB address for a particular task can be looked up in `TCBPrioTbl`, the doubly-linked list manipulation also has a constant overhead. Task scheduling involves looking up the highest priority task in the `OSRdyTbl`, which has a constant overhead as well.

Transition from the Ready state to the Waiting state involves suspending, delaying, or blocking a task. These operations perform similar steps:

- Look up the task TCB by its priority in `TCBPrioTbl`.
- Mark the state as suspended, delayed, or blocked depending on the operation by modifying the appropriate TCB fields.

- Clear the `OSRdyGrp` and `OSRdyTbl` bits for the task.
- When the task is blocked on an event, the TCB is added to the event's waiting list.
- Except the situation where a task is suspending a lower priority task, the OS invokes the task scheduler to run a ready task.

Transitions that involve context switches and interrupts also have constant overheads. During a context switch, the information in the CPU registers is saved to the current task stack and the new task information is copied into the CPU registers. To service an interrupt, the CPU registers are saved before user ISR routine is invoked. After user ISR routine finishes, the OS invokes the task scheduler. If the interrupted task has the highest priority, its state is restored into the CPU registers and execution resumes. Thus context switch and interrupt handling have constant overhead as well.

Transition from the Waiting state to the Ready state is split into two parts. The first part includes the resuming functions for blocking, suspending, and delaying functions. The OS performs the steps below:

- Look up the TCB in `TCBPrioTbl` by the task priority.
- Clear the appropriate TCB state fields depending on the operation.
- When a task is unblocked by an event, the task is removed from the event's waiting list.

- Set the corresponding bits in `OSRdyGrp` and `OSRdyTbl` if the task is ready to run.
- Invoke the task scheduler.

As before, these steps all involve constant overheads.

The other part of the transition from Waiting state to Ready state includes the timer tick services. μCOS requires a periodic timer for time keeping. The timer granularity can be changed. The timer is updated by the timer ISR which is invoked at the set granularity. The timer ISR serves two purposes:

- Traverse the list of existing TCBs (`TCBList`) and decrement each non-zero delay field. If a delay field reaches zero, the corresponding task is made ready to run by setting the bits in `OSRdyGrp` and `OSRdyTbl`.
- Increment the global timer variable.

The first purpose of the timer ISR imposes a linear overhead on the system. The overhead is proportional to the number of tasks existing in the system. This is undesirable since the amount of time to service the timer ISR can vary dramatically. If there are many tasks existing in the system, the long period of time required for the timer ISR can cause other high priority tasks to miss deadlines.

With the exception of the timer tick operation, all other mentioned operations have constant overheads. However, under a heavy workload, the timer tick operation can dominate the total OS overhead since it is proportional to the workload. Thus nanoScheduler is introduced to off-load part of the OS functionality into hardware. Performing some OS functions in hardware reduces the total overhead incurred in the OS.

5.2.2 NanoScheduler Functionality

The table below illustrates the partial functionality left in the software. The rest of the functionality is done in the nanoScheduler, which is discussed in detail in the next section.

Functions	OS Functionality in the presence of the nanoScheduler	OS Functionality Moved to the nanoScheduler
Task Creation	• create TCB • store the TCB address to TCBPrioTbl	• insert the task to a TCBList • mark the task as ready to run • insert the task to a ready list
Task Deletion	• clear TCBPrioTbl entry • clear the task TCB	• remove the task from the TCBList • remove the task from the ready list
Task Blocking	• add the TCB to the event's waiting list	• look up the task from the TCBList • mark the task state as blocked • remove from the ready list
Task Unblocking	• remove the TCB from the event's waiting list	• look up the task from the TCBList • mark the task state as ready • insert the task to the ready list
Task Suspending	• none	• look up the task from the TCBList • set the task suspend state • remove the task from the ready list

Table 5.1: Functionality of the OS with the nanoScheduler for handling task state transitions.

Functions	OS Functionality in the presence of the nanoScheduler	OS Functionality Moved to the nanoScheduler
Task Un-suspending	• none	• look up the task from the TCBList • clear the task suspend state • insert the task to the ready list
Task Delaying	• none	• look up the task from the TCBList • set the task delay state • set the delay amount • remove the task from the ready list
Task Un-delaying	• none	• look up the task from the TCBList • clear the task delay state • clear the delay amount • insert the task to ready list
Task Scheduling	• none	• traverse the ready list to find the highest priority task to run
Timer Tick	• update the global timer variable	• traverse the TCBList to decrement each non-zero delay fields. I f a delay field reaches zero, the corresponding task is made ready to run
Context Switch	• same as original OS	• none
Interrupt Handling	• same as original OS	• none

Table 5.1: Functionality of the OS with the nanoScheduler for handling task state transitions.

Table 5.1 shows that with the nanoScheduler, the OS performs much less operations. Since context switching, interrupt handling, and event management are left in the software, the OS still needs to keep the TCBs for existing tasks. In addition, the `TCBPrioTbl` is still needed to lookup a TCB address by the task priority. However, since the doubly-linked `TCB-List` is used to service timer tick operation, it is eliminated in the software. Instead, the nanoScheduler performs the timer tick operation in

hardware. Similarly, `OSRdyGrp` and `OSRdyTbl` are also eliminated from the software since task scheduling is now moved to the nanoScheduler.

Using the nanoScheduler greatly simplifies the OS. The most significant saving comes from the timer tick operation. Using the nanoScheduler reduces the linear overhead to a small constant overhead. The saving is amplified when there is a large number of tasks running on the system. To be specific, performing the timer tick operation entirely in software incurs an overhead of:

100 cycles + 120 cycles * # of tasks.

It's easy to see that for 8 tasks in the system, the overhead can exceed over 1000 cycles. Furthermore, this overhead is incurred for each timer tick. As the granularity of the timer tick reduces, the overhead can burden the system significantly. In contrast, using the nanoScheduler keeps the timer tick operation under a constant overhead, which is independent of the number of tasks in the system. This makes the system scalable and can support a higher-resolution timer.

In addition, many operations are entirely eliminated, i.e., suspending and delaying a task. For other operations, using the nanoScheduler reduces the software overhead, i.e., blocking and unblocking a task. Although invoking the nanoScheduler adds a small overhead to the system, as our simulation results show in chapter 8, the overhead increase due to the

nanoScheduler is insignificant compared to the saving achieved in the timer tick operation.

With the potential benefits of the nanoScheduler, we examine the task structure of the nanoScheduler in the next section.

5.3 NanoScheduler Task Structure

In order to schedule for all tasks created, the nanoScheduler must have buffer storage for all tasks. In the case of μCOS, 64 task priorities are allowed. The OS uses the lowest task priority for idle task. Therefore, a total of 63 tasks are allowed for user applications. Since the idle task is scheduled by default and the task states are never changed, the nanoScheduler only needs to allocate space for 63 tasks. The task information is stored in the main memory as part of the dedicated nanoScheduler memory. In addition to the task array, the nanoScheduler also keeps two 1-byte registers for the total number of existing tasks and the array index for the first task element. These two registers are referred to as ListSize and ListHead in the remainder of the chapter.

The EDF algorithm is implemented as a singly-linked list in the nanoScheduler. An element in the list is a data structure keeping track of state information for one task. Thus there are a total of 63 elements in the list. Creation in list takes O(N) if N is the size of the list. The linear growth

is due to the list traversal required in linked list. Searching is a constant time (O(1)) operation if the list only consists of ready tasks. This suggests that the nanoScheduler needs to manage multiple lists for tasks with other states, i.e., blocked, suspended, delayed, etc. Having multiple linked lists can greatly complicate the implementation. Alternatively, the nanoScheduler can be implemented in Content Addressable Memory (CAM); achieving constant time for list operations. However, this would require a comparator for each element in the list, which complicates the hardware. Further, searching is done in parallel in all elements of CAM, thus power consumption would increase as well. On the other hand, if the nanoScheduler keeps a single list and adds a status field to distinguish task states, the searching of the list can not be performed in constant time anymore. It is possible that tasks at the front of the list are blocked and the last element in the list is the only task that is ready to run. Thus the worst case searching time becomes O(N) as well. In reality, this occurs rarely and searching should finish in a relatively short amount of time. However, because a RTOS needs to guarantee a deterministic behavior for time-critical applications, the worst case performance should be assumed at all times.

Based on these considerations, the nanoScheduler implements a single linked-list. Even though the linked-list operation times can be linear, the overhead is not significant as long as the number of tasks in the system

is small. Since most embedded systems support only a modest number of tasks, the linked list implementation does not create performance bottlenecks, as shown in the performance analysis in chapter 7.

For each task, the nanoScheduler needs to know the task's *period*, whether it is a *periodic* task, the *deadline* of the task, the *priority* level, and some state information (*ready*, *suspend*, *delay*). Therefore, all of these fields are present in the data structure for a task. Furthermore, since the nanoScheduler is implemented as a singly-linked list, field like *next* task pointer is also needed. The figure below illustrates the structure of the linked list along with the two auxiliary registers ListHead and ListSize.

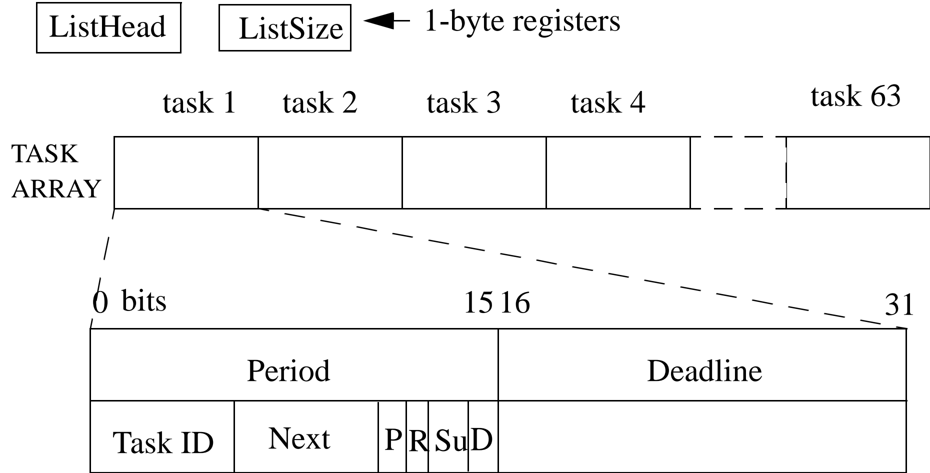


Figure 5.5: NanoScheduler Data Structure.

The table below explains the fields, their length, and their functionality.

Field Name	Length (bit)	Functionality
Period	16	Specify a task period.
Periodic (P)	1	Indicate whether a task is periodic or not.
Deadline	16	Specify a task deadline. It is initially set to Period, and is decremented on every timer interrupt. When it reaches 0, it is reset to Period to indicate the beginning of a new period.
Task ID	6	Specify the id of the calling task. Since there are at most 63 tasks in the list, a task ID requires 6 bits.
Ready (R)	1	When set, indicate that the calling task is ready to be scheduled. When cleared, indicate that the calling task is blocked, i.e., by a semaphore call.
Suspend (S)	1	When set, indicate that the calling task is suspended. A task can be suspended while under states other than ready.
Delay (D)	1	When set, indicate that the calling task is delayed. The delay amount is specified in the deadline field, and is decremented at every timer interrupt until reaches 0. Upon reaching 0, the scheduler clears the delay flag. A task can be delayed while under states other than ready.
Next	6	Store the task id of the next task in list. Since the list is sorted by deadline values, the next task is the one with the next smallest deadline values. When any task's period restarts, the next fields of all task need to be reordered to reflect the change.

Table 5.2: NanoScheduler Data Structure Fields and Their Explanations.

In all, each data element requires 48 bits or 6 bytes. The array of 63 element is statically allocated, making the storage area to be 2976 bits or 372 bytes. Include ListHead and ListSize registers, the total storage area is 374 bytes. The array structure and the two auxiliary registers are for nanoScheduler use only and should not be accessed by other components in the system. Thus they reside in the dedicated memory section as illustrated in figure 5.2.

5.4 NanoScheduler Interface

The processor communicates with the nanoScheduler through memory-mapped ports. The address map for these ports are given below.

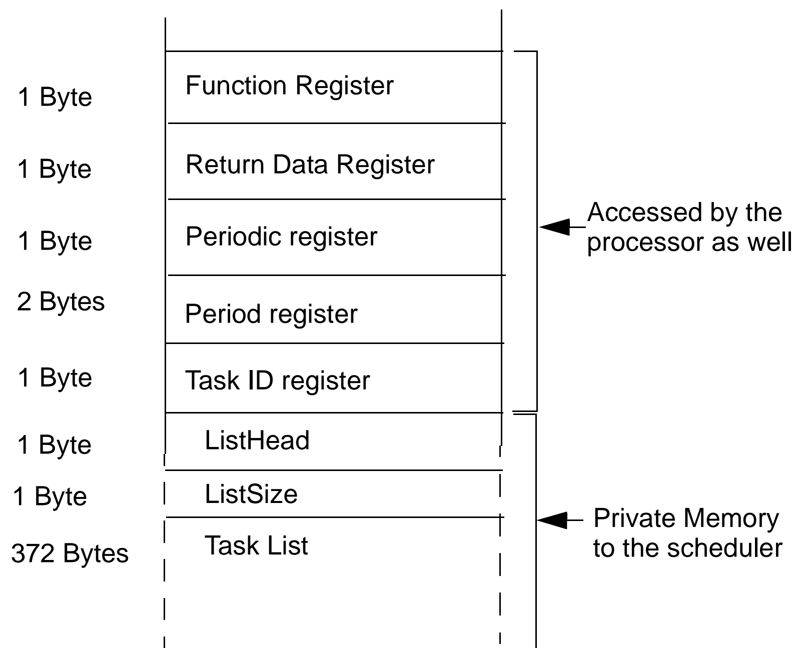


Figure 5.6: Address Map for NanoScheduler Control Registers.

In the case of creating a new periodic task that has a period of 144 ms and a task id of 5, the processor would write to these registers with the task parameter values. To be specific, the processor would include the following code:

```
{
    /* inform the nanoScheduler to create a task */
    *Periodic Register = 1;
    *Period Register = 144;
    *Task ID Register = 5;
    *Function Register = NanoCreateTask;
    while (*Function Register!=0) {};

    /* leftover OS functionality */
    Create_TCB();
    Store_TCB_to_Tbl(TCBPrioTbl);
    Context_Switch;
}
```


The above code indicates that the processor writes the task parameters to the nanoScheduler registers and tells the nanoScheduler which function to execute through the Function Register (FR). After the command is issued, the processor waits for the nanoScheduler to respond. The nanoScheduler clears the FR when finishes. In the case when the nanoScheduler needs to return values, the Data Register is used. The processor reads the Data Register after detecting that the nanoScheduler finished execution.

5.5 Implementation Details

The scheduler supports several functions that manipulate task states and the linked list. These functions are described below.

- NanoCreateTask(periodic, period, pid): When a task is created, it is initialized in the next available data structure in the array (which is indicated by the ListSize register) with *ready* set, *suspend* and *delay* state cleared, *periodic* set to periodic, *deadline* and *period* set to period, and *task-id* set to pid. All existing tasks, besides the newly created task, are reordered by *deadlines* to reflect the new element in the list and their *next* fields changed if necessary. The operation essentially inserts the newly created task into its appropriate position in the linked list. If there are multiple tasks having the same *deadline*, they are ordered by the existing order in the list. In other words, the newly created task is inserted to be the last task within the task group that have the same *deadline*. This function is triggered when FR is 1. It takes one cycle to compare to one task in the list. This insertion takes O(N) time where N is the number of tasks. So the max-

imum insertion time is 63 cycles. The function flow diagram below illustrates the functionality.

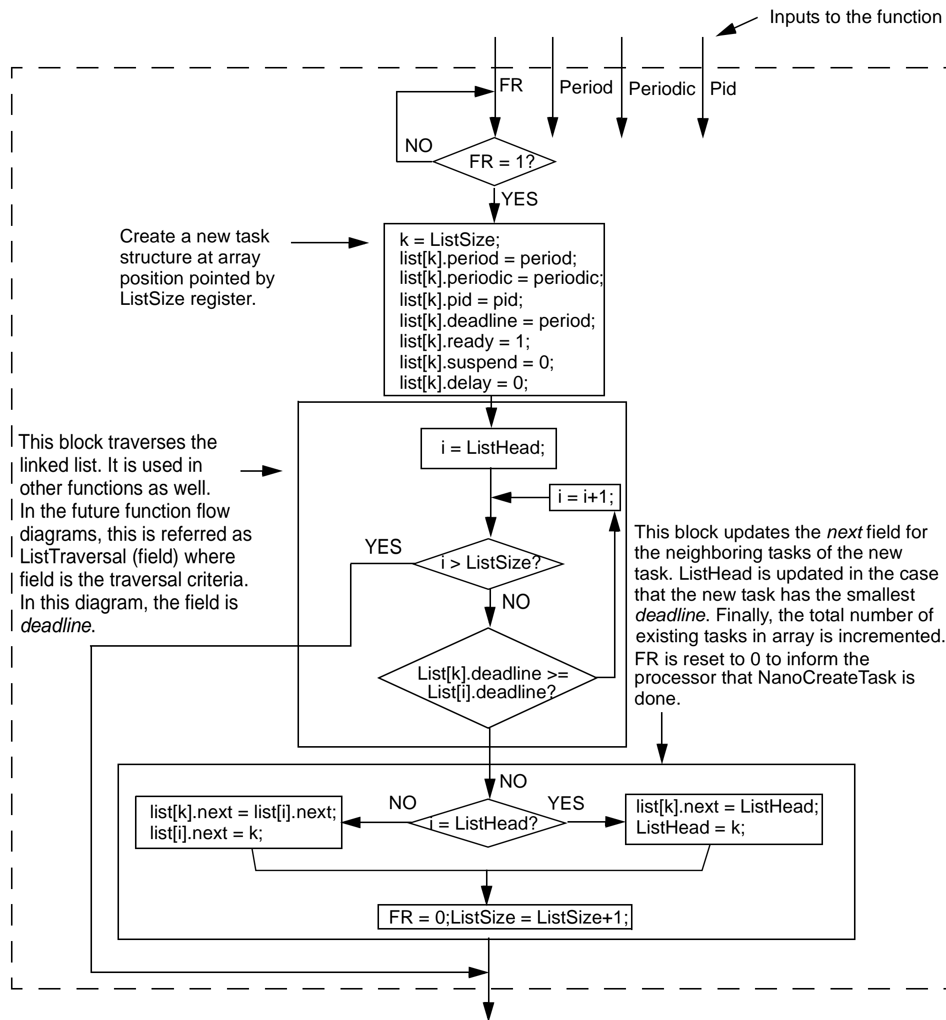


Figure 5.7: Function Flow Diagram for NanoCreateTask.

- NanoDeleteTask(pid): Similarly, when a task is deleted, the OS provides the task id to the nanoScheduler. The scheduler first traverses the array to find the current position of the task. Then the *next* field of the element who precedes the task is modified to skip the current location. Once a

task is deleted, it can not be restarted without creating a brand new task with a different task id. This function is triggered when FR is 10. The deletion operation also takes O(N) time due the list traversal. The following flow diagram illustrates the functionality:

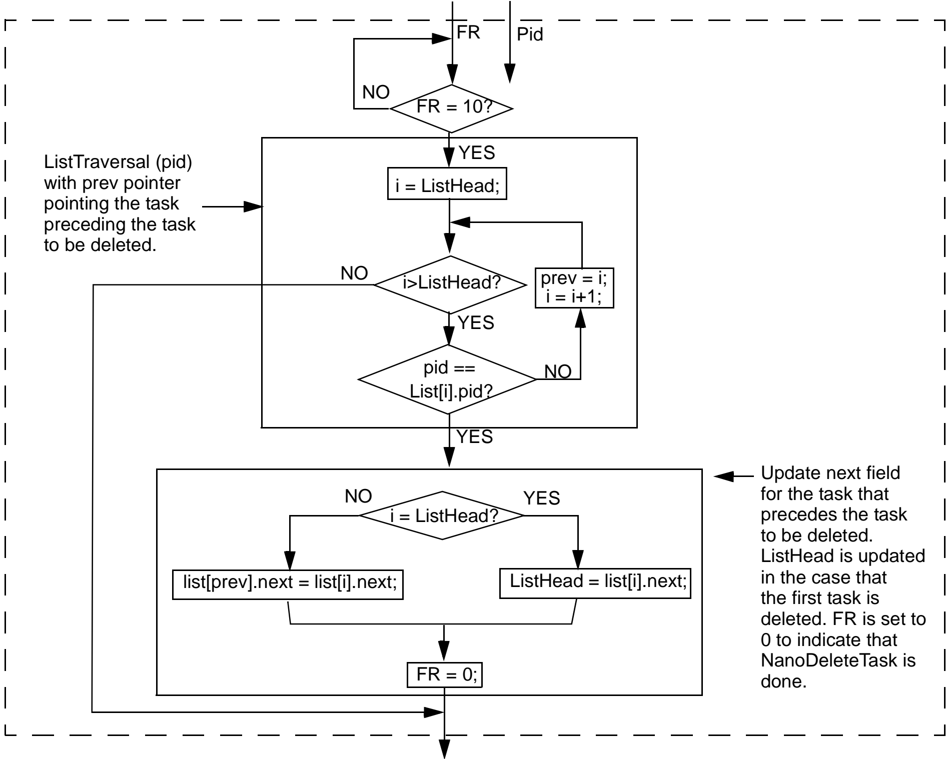


Figure 5.8: Function Flow Diagram for NanoDeleteTask.

When a task is placed on wait queue, or suspended, or delayed, the state information for that task needs to be updated. The OS provides the id of the task since it is the only unique parameter to distinguish the current task from all other tasks. It then calls the appropriate nanoprocessor function to perform the operation. The list is traversed from the first task until

the matching task is found. Thus, changing state information takes O(N) time as well. The specific functions the nanoprocessor supports are:

- NanoChangeStateWait (pid): changes the task with id *pid* from ready state to wait state. This function is triggered when FR is 3.

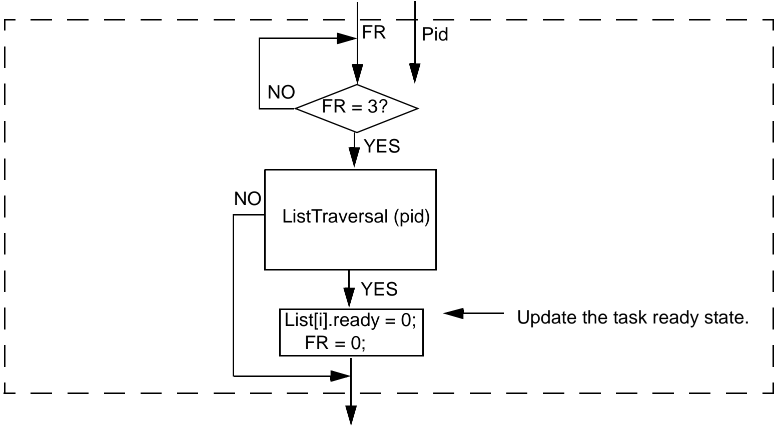


Figure 5.9: Function Flow Diagram for NanoChangeStateWait.

- NanoChangeStateReady(pid): changes the task with id *pid* from wait state to ready state. This function is triggered when FR is 4. The function flow diagram is similar to figure 5.5, except the *ready* field is set.

- NanoSuspendTask(pid): suspends the task with id *pid* by setting the *suspend* flag. This function is triggered when FR is 5. The function flow diagram is similar to figure 5.5, except the *suspend* field is set.

- NanoUnsuspendTask(pid): un-suspends the task with id *pid* by clearing the *suspend* flag. This function is triggered when FR is 6. The function flow diagram is similar to figure 5.5, except the *suspend* field is cleared.

• NanoTimeDlyTask(pid, delay): delays the task with id *pid* by delay amount and sets the *delay* flag. The delay amount is specified in terms of OS timer ticks. The value is stored in the *deadline* field. This function is triggered when FR is 7.

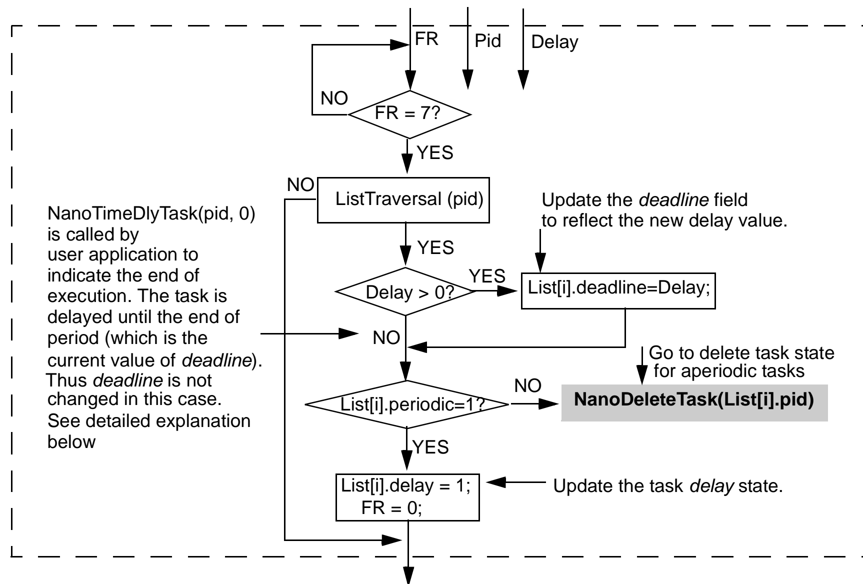


Figure 5.10: Function Flow Diagram for NanoTimeDlyTask.

When a task finishes execution, the *deadline* value indicates the remainder of the current *period*. To prevent the task from executing again in the same period, the user task calls NanoTimeDlyTask(pid, 0) to set the *delay* flag after execution, with *deadline* set to the remainder of the current period. This way, the scheduler can guarantee that the task is not scheduled

till the beginning of next period. The following graph illustrates the actions taken by the scheduler.

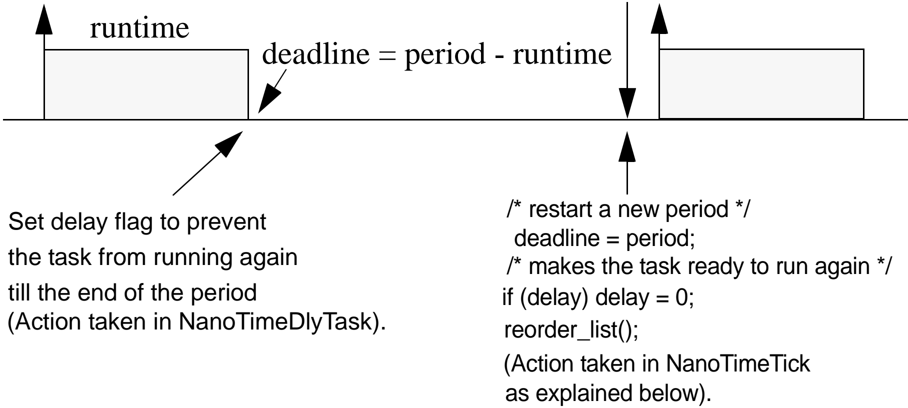


Figure 5.11: nanoScheduler's actions for task management.

When an aperiodic task tries to set the delay flag after execution, the task is deleted rather than delayed. Thus the nanoScheduler is transferred to the NanoDeleteTask state.

- NanoResumeTimeDlyTask(pid): un-delays the task with id *pid* by clearing the delay amount and the *delay* flag. This function is triggered when FR is 8.

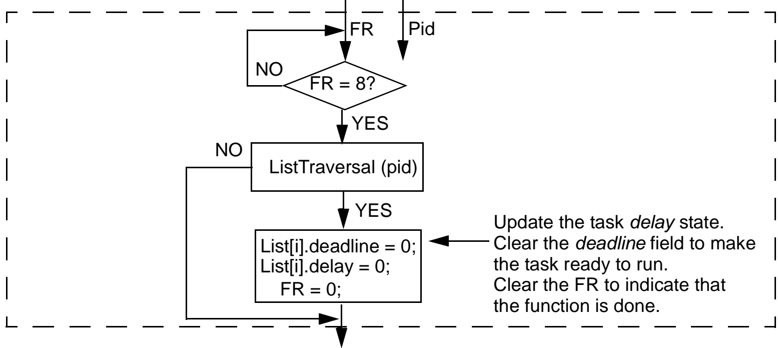


Figure 5.12: Function Flow Diagram for Resuming Delay Task.

- NanoTimeTick(): A timer tick is the unit of time in an OS. Often, an OS has a dedicated register that keeps track of time. Without the nanoScheduler, updating the timer register would require a special interrupt (the timer interrupt) to occur at each timer tick granularity. An ISR is executed to service the timer interrupt. Inside of the ISR, global timer register is updated along with the list of tasks. All non-zero delay fields are decremented in software.

When the nanoScheduler is used, the timer ISR makes a call to the scheduler instead on every execution. The nanoTimeTick function traverses the entire task list to decrement non-zero *deadline* fields in hardware. When a *deadline* becomes 0, it is reset to the task *period* to indicate the beginning of a new period. If the *delay* flag was set, it is cleared to make the task ready to run again. However, since the deadline values have

changed for some tasks, the list must be reordered, which is indicated by the “REORDER=TRUE” statement shown below.

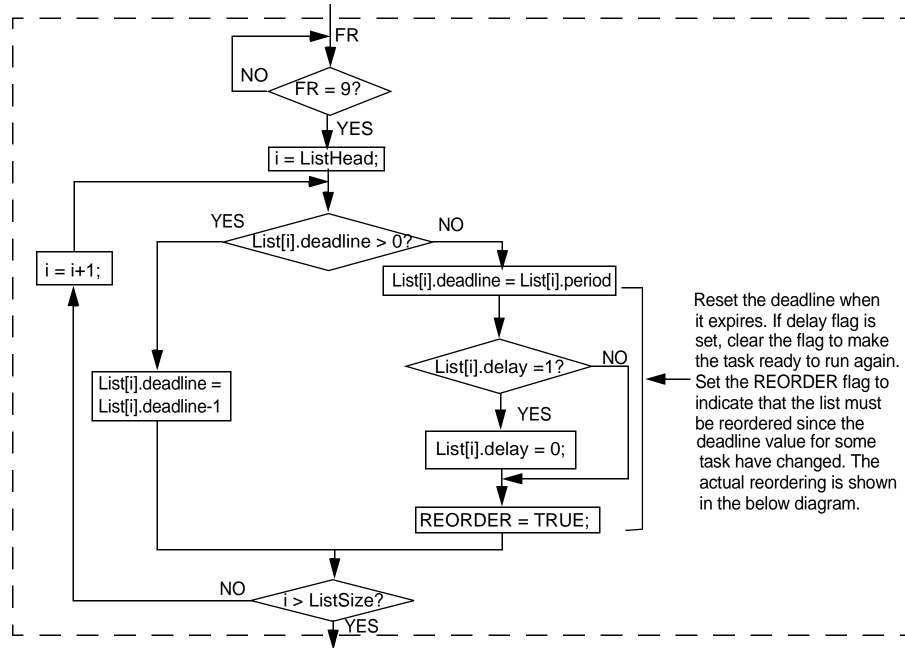


Figure 5.13: Function Flow Diagram for NanoTimeTick, without reorder operation.

Since the list is ordered by the task *deadlines*, the tasks at the head of the list always expires first. Thus, after renewing the *deadlines*, the reordering operation pushes the tasks at the head of the list back. The renewed tasks have their *periods* equal to their *deadline*s. In the reordering operation shown below in figure 5.10, while the head task is renewed (List[ListHead].period = List[ListHead].deadline), the list is traversed to find an appropriate position for the renewed task. Block 1 performs the list traversal; the new task should be inserted between prev and curr pointers.

After the new position is found, the operation checks that the new position isn't the same as the original position (List[prev].deadline = List[List-Head].deadline is not true). Block 2 performs the insertions for all tasks that are instances of the same benchmark (these tasks would be renewed together since they have the same deadline). The list traversal and the task insertion are done as many times as there are renewed benchmarks.

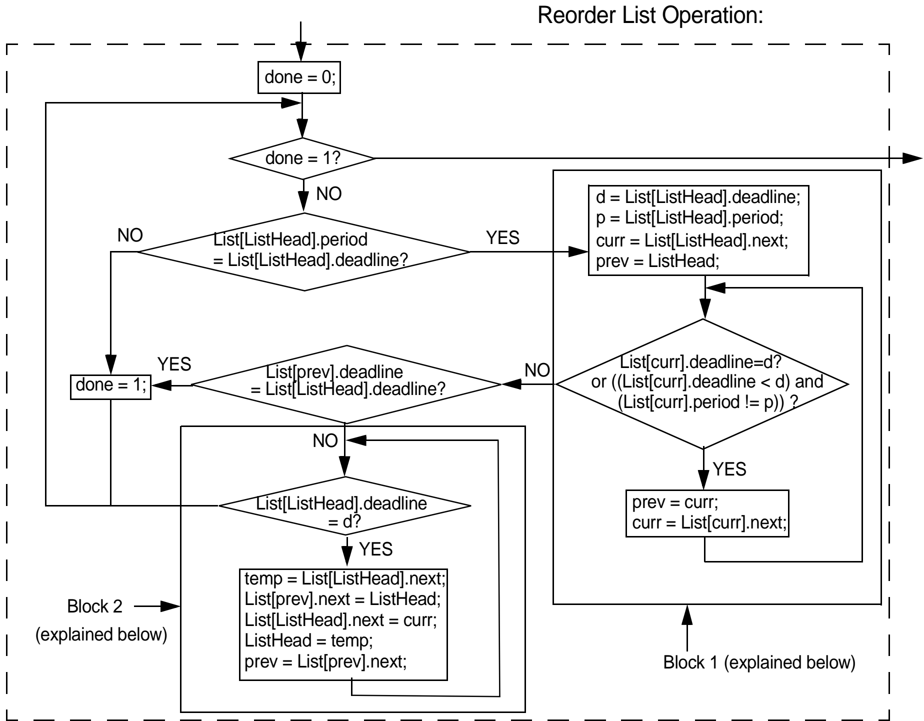


Figure 5.14: Flow Diagram for the List Reordering Operation of the NanoTimeTick function.

- NanoGetHighID(): traverses the linked list to find the first task that is ready to run. Since the list is sorted by the *deadline* values, the task in the

front of the list has the smallest *deadline*. The function returns the task id to the OS. This function is invoked when FR is 2.

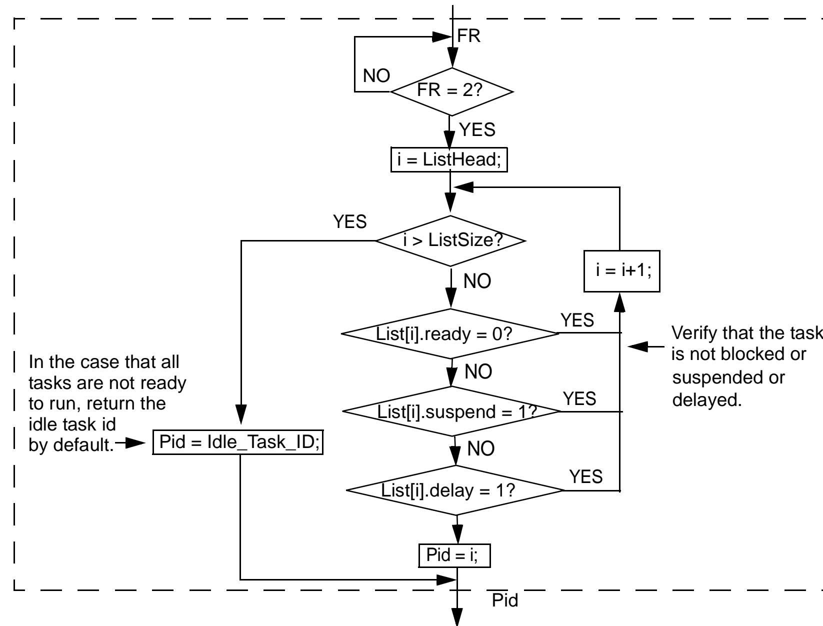


Figure 5.15: Function Flow Diagram for NanoGetHighID.

Finally, when an OS switches contexts, it executes the task with the highest priority. In EDF, the highest priority task is the task having the smallest *deadline* value and is ready to run. Since the list is sorted by *deadline*, the hardware traverses the list starting with the head element. As long as an element is not ready to run, the element pointed by the *next* field is examined. The worst case is that all 62 tasks at the front of the list are not ready to run and the last task is the only ready task. This case rarely happens, however, it must be considered. Thus, the searching time also takes O(N) or 63 cycles for worst case scenario.

Even though the linked list operations take O(N) times to complete, the overhead of this operation is still low. This is because the OS has an upper bound on the number of tasks supported. Typically, embedded RTOSs support a limited number of tasks. Using the nanoScheduler would create an overhead of 63 cycles in all list operations for the worst case. Smaller overhead can be achieved when the number of tasks in the system is low.

CHAPTER 6

EXPERIMENTAL METHODS

In order to accurately characterize the effect of nanoprocessors on real-time embedded processors, a test system is setup in a realistic manner to capture the performance behavior. The test system closely resembles real-world systems used for developing commercial embedded hardware products. To analyze costs effectively, another test system is also setup to find estimates of die area increase and additional power consumption for the nanoprocessors.

6.1 Performance Simulator

The performance simulator used in this study is the SimBed simulator[24, 25]. SimBed is a cycle-accurate simulator developed to run unmodified RTOSs. The executable of the RTOS coupled with user applications is read into the processor simulator. The entire simulation environment including user applications, RTOS, and processor simulator executes on the workstations in System and Computer Architecture laboratory at the University of Maryland. These workstations are Pentium-III dual processor

machines running the Mandrake Linux operating system. The test environment setup is illustrated as below:

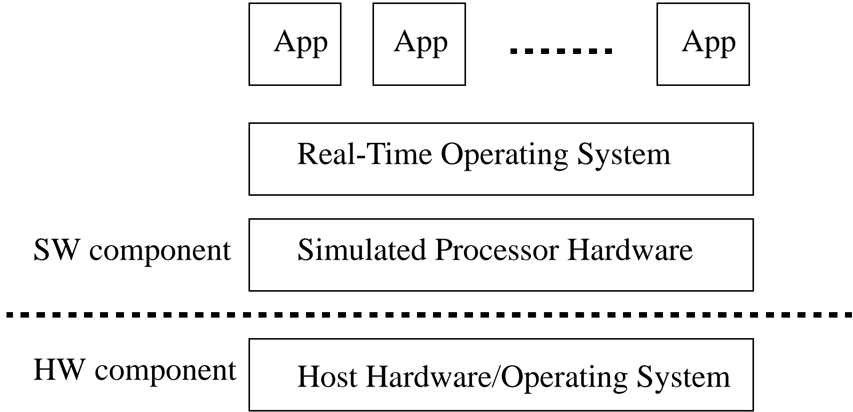


Figure 6.1: Simulation Environment Block Diagram.

6.1.1 Processor

The particular processor the simulator models in the test environment is the Motorola's M*CORE processor[6][7]. The M*CORE processor is a 32-bit general-purpose microcontroller designed by Motorola to target devices that are battery-powered, thus the processor is designed from the ground up with low power consumption in mind.

The processor is an integer processor conforming to the Reduced Instruction Set Computer (RISC) architecture. It offers a 4-stage pipeline with fetch, decode, execute, and register writeback stages. Both user mode and supervisor mode are supported. Instruction length is fixed at 16 bits and most of the instructions can be executed in both modes. A few instruc-

tions are privileged, being accessed only in supervisor mode. Most of the instructions are one cycle operations. Branch instructions and memory instructions are executed in two cycles. Memory instructions that involve multiple registers take multiple processor cycles to complete.

The processor has 32 internal registers, 16 of them being general-purpose, accessible in both operating modes, and the rest being the shadow registers used only in supervisor mode to lower the overhead for interrupt exception processing. In addition, the processor supports one fast interrupt and up to 32 unique interrupt sources.

For the test environment, the processor is set to run at 20MHz. The speed of the processor is not impressive for today's processor world, however, since the M*CORE is a power conscious processor designed specifically for embedded applications, performance isn't the single most important issue and can be sacrificed to save on power consumption.

To support multiple interrupt sources and operating systems, extra hardware is required in addition to the CPU. The simulation environment implements an interrupt handler. Interrupts are vectored to the processor through dedicated registers. A hardware timer unit is also included. It generates a timer interrupt at a predetermined period informing the operating system to increment the global timer counter. This is crucial to the OS since

some of its features, such as the ability to delay tasks, depends on an accurate global timer.

The simulation system also simulates on-chip memory. In the current setup, the memory is fixed at 12 MBytes. The memory bus is a 32-bit bus. This memory space is used by both the OS and user applications. Each internal peripheral has a dedicated range of memory. The size of the memory is large for a typical embedded system. This is because no external memory is simulated. For simplicity, the RTOS and the user applications are assumed to be loaded on chip. The large memory space is needed for the OS and user applications.

6.1.2 Operating System

The real-time operating system chosen for the test environment is the Micro-C/OS II(μCOS) or MicroController Operating System version 2, developed by Jean Labrosse[5]. Specifically, μCOS is a portable, scalable, preemptive, real-time multitasking kernel. It supports up to 62 user applications. It provides several interprocess communication methods such as semaphores, message queues, and mailboxes. It also provides task management such as task creation, deletion, priority-level changes, task state changes, etc. In addition, μCOS provides time management and fixed-size memory block management. Moreover, μCOS also comes with hardware-

dependent code for context switching and interrupt handling. μCOS has been used in many commercial embedded products, i.e., cell phones, self service stations, and credit card processing units[8].

To understand the proposed hardware, the communication aspect of the OS is discussed in detail. Any external devices that wish to communicate with the processor must go through I/O ports. For programming transparency, I/O ports are mapped to particular memory addresses. To access the I/O ports, load and store instructions are used. The system can distinguish between different ports from the memory address provided. The nanoprocessors are treated as memory-mapped on-chip peripherals.

6.1.2.1 Processor Scheduling

The OS does not have a sense of a task period. The OS simply asks for a task to run when it is idle. No check is performed to see if the same task has been running twice in a row, or if another task hasn't run for a while. Therefore, the period of a task must be enforced through a user program. The user program starts off the task with the highest priority. When the task finishes execution, it goes on a wait queue. The user program increments a counter for that task at every time tick and wakes up the task if the counter reaches the task period. This user program, which is a task

itself, has the highest priority in the system (priority of 0) to ensure that it is executed at every time tick. This creates unnecessary overhead for the OS.

6.1.3 Benchmarks

The benchmarks used in this study are taken from MediaBench suite[9] and MiBench suite[10]. The MediaBench suite primarily consists of voice compression/decompression algorithms used commonly in cellular networks. MiBench suite is a superset of MediaBench and other commercially representative embedded applications. The goal of MiBench is to cover a diverse application domain and represent embedded applications from different markets. The suite is modeled after the benchmark suite developed by the EDN Embedded Microprocessor Benchmark Consortium (EEMBC). The MiBench breaks down to six categories: automotive and industrial control, consumer devices, networking, security, office automation, and telecommunications. The telecommunications category overlaps with MediaBench. Because M*CORE is a integer processor, we choose only the integer benchmarks for this study. We were able to transform some of the floating point benchmarks into integer applications without loss of generality.

6.1.3.1 ADPCM

ADPCM, or Adaptive Differential Pulse Code Modulation, is a speech compression and decompression algorithm used frequently for audio coding. It offers intelligibility at a high compression rate. A common implementation of the ADPCM takes 16-bit PCM samples and compress them into 4-bit samples, achieving a compression rate of 4:1. There are two components to the benchmark. The compression component takes an audio file with PCM samples and produce a compressed ADPCM output file of the same audio file. The decompression component does the reverse.

6.1.3.2 GSM

GSM is the defined standard protocol for cellular networks in Europe. GSM has a codec which compress and decompress audio speech files. In the benchmark used in this study, GSM compresses frames of 160 13-bit samples into 260 bits achieving a compression rate of 8:1. The quality of the algorithm is good enough for reliable speaker recognition which is used in many automated telephone systems. Like ADPCM, GSM benchmark also takes PCM audio files for compression and produce GSM files for transmission over network. Decompression component does the reverse.

6.1.3.3 Patricia

Patricia is a benchmark which falls in the networking category. A Patricia trie is a data structure used in place of a full tree with sparse leaves. If a branch only has one leave node, it is collapsed upwards in the trie to reduce traversal time. Often, a trie is suitable to represent routing tables in a network. The input file into the benchmark is a list of mock IP addresses simulating traffic to a highly active web server. The benchmark builds a patricia-trie structure using these addresses and outputs short messages depending on whether an address is found in the trie.

6.1.3.4 Dijkstra

Dijkstra is a networking algorithm which computes the shortest path for a given set of nodes. The benchmark construct a large graph in the form of adjacency matrix and compute the shortest path between every pairs of nodes repeatedly. The algorithm executes in $O(n^2)$ time. The input to the benchmark is a cost matrix defining the inter-nodes cost. The output of the file is a number of shortest paths between nodes.

6.1.3.5 SHA

Sha, or secure hashing algorithm, is used often in generating digital sigatures. It’s used in the famous MD4 and MD5 hashing functions. Sha

produces a 160-bit (20 bytes) message digest for a given input. The input file is a large ASCII text file from a random article found online.

6.1.3.6 Bitcount

bitcount is a benchmark in the automotive and industrial control category, testing the bit manipulation of a processor. The benchmark counts bits in an array of integers using 5 different methods. The methods are optimized 1-bit per loop counter, recursive bit count by nibbles or half-bytes, non-recursive bit count by nibbles using a table look-up, non-recursive bit count by bytes using a table look-up, and shift and count bits. The input data is a list of numbers with equal numbers of 1's and 0's.

6.1.3.7 G721

G.721 is an international telecommunications standard for digital coding of analog signals. The standard was defined by the International Telegraph and Telephone Consultative Committee (CCITT), which is now part of the International Telecommunications Union (ITU). The compression and decompression algorithms in G.721 are used in this study.

6.1.3.8 Homemade Benchmark

This benchmark is used for studying the impact on system performance from using the hardware scheduler. Each iteration of the benchmark

increments a counter 185 times. The benchmark has a well-defined runtime of approximately 0.2 ms. This simple benchmark is needed because it provides a fixed runtime value with no variation. This characteristics makes testing schedulers easier.

6.1.4 Tasks

Some of the benchmarks include multiple components. In the simulation environment, each component is treated as an individual task. For example, the ADPCM codec has encoder and decoder components. Each component by itself is a stand-alone task. In addition, the OS can execute multiple copies of a task depending on the configuration. The total number of tasks in the system is defined to be the workload of the system. To get a uniform workload, multiple instances of the same task are created. This workload represents a system that provides a well-defined service to multiple clients. To get a mixed workload, various tasks with different periods can be used. In addition, the system can vary the number of instance of each task to introduce diversity in the workload.

6.1.5 Measurements

To evaluate the impact of the nanoprocessors on the system, we measure performance such as system bandwidth and processor utilization rate.

System bandwidth is the maximum number of tasks a system can handle when the processor is executing 100% of the time. It is a measure of system throughput. Under a fixed workload, we also look into the processor utilization rate. The chosen workload is the maximum sustained workload for the original system setup without the nanoprocessors. The processor utilization rate is the breakdown of the processing time that is used in executing applications and OS overhead. It is a measure of system efficiency. The processor utilization rate for the OS overhead can be broken into several categories, such as process scheduling, interrupts, context switching, etc. Our results show only the breakdown of the OS overhead including idle time. The rest of the processing time is application related.

Two system setups are used; one for the original system, one for the system using nanoprocessors. The nanoprocessors are tested one at a time. To make a fair comparison, all test parameters are kept the same on the two setups. These include the types of tasks, task periods, and simulation time. We vary the number of tasks to test the system limits. To measure the processor utilization rate, we evaluate both system setups executing the same workload within the same simulation time. The processor utilization is broken up into categories and the results compared.

To measure the system cost, the cost simulation environment is described below.

6.2 Cost Simulator

Because nanoprocessors are envisioned as on-chip peripherals, they are modeled as circuitry for cost estimates. To achieve accuracy, the Cadence and Synopsys toolsets are used. First the nanoprocessors are modeled in the Verilog Hardware Description Language (HDL) to verify the correctness of the design. Then they are synthesized into gate-level schematics for area and power estimates.

Verilog is a high-level language that are used to model hardware components. Data types include those that represent native hardware units such as registers, wires, buffers, etc. In many ways, Verilog resembles C, providing users with conditional statements, array structures, loop statements, and even some limited file operations. In other ways, Verilog code mimics real hardware. There are distinctions between the positive edge and negative edge of a signal, edge-triggered vs. level-triggered logic, and the important concept of non-blocking assignments. Non-blocking assignments allow multiple statements to be evaluated simultaneously within the same clock period.

To measure the impact of nanoprocessors, the nanoprocessors that are modeled in C are also implemented in Verilog. The functional correctness of the Verilog code is verified through a test driver that tests each nanoprocessor's functionality thoroughly. Files are used to represent the external

data arriving at an I/O port. Input files are read into a piece of memory at initialization. During the execution, the nanoprocessor code can directly access the memory that contains the input files. This direct data access takes 1 cycle for each word, thus simulating the special bus connecting between the memory and the nanoprocessors.

To provide a reference for the area and power estimates, an M*CORE simulator has also been built. The 4-stage pipelined simulator is cycle accurate. It includes the register file, an ALU, and vectored interrupt handling. Memory is not included in the simulator. Inside the ALU, a divider for signed and unsigned divides is built to reflect the accurate hardware cost[1]. This simulator is not used for performance analysis. It is used to provide a reference point in area and power estimates.

After the Verilog simulators are built, we use the Synopsys toolset to synthesize our HDL design. Synopsys tools allow designers to make the first step from a high-level design to realizing it in hardware. The toolset translates a high-level HDL program into gate-level schematics and optimizes the design based on area, timing, and power requirements. Since these requirements are dependent on the fabrication technology used, a technology library is provided. In this work, a 0.25 µm standard cell library

1.A multiplier is not built because the technology library used for synthesis can provide a generic multiplier.

from the Taiwan Semiconductor Manufacturing Company (TSMC) is used. The library provides detailed area and delay information for numerous gates used in circuitry building. Further, we configure the Synopsys toolset to use parameters closely resemble our M*CORE simulation system. These parameters for synthesis is outlined below in table. With the library and user Verilog code fed into Synopsys, the software optimizes the user design based on configuration. To get a consistent result, the same optimization parameters are used for the M*CORE and the nanoprocessors.

Parameter	**Value**
Technology Library Used	TSMC Standard Cell Library
Clock Frequency	20MHz
Operation Condition	Typical
Wire Load Model	Conservative
Max Fan-out for Nets	4

Table 6.1:Synthesis Conditions

The area estimates are obtained from the technology library information. It is the sum of all cells generated by the Synopsys synthesis toolset. There is no unit for area estimates. It is specific to the Synopsys toolset and the technology dependent library. The total area is the sum of combinational logic and non-combinational logic. The non-combinational logic includes the Flip-Flops used for storage. The combinational logic includes

the gates used to represent the logic. In addition, Synopsys provides other information such as total cell counts, etc.

For reference purpose, we obtained the following information from simulation. First, we obtained that a NAND gate or a NOR gate has an area of 17. Second, to find the Register-Bit-Equivalent (RBE) value, we have built a 16 x 16 register file illustrated below.

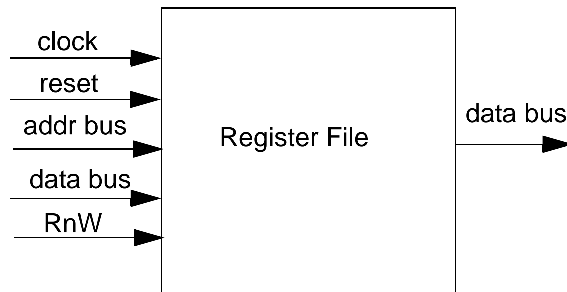


Figure 6.2: A 16 x 16 register file representation.

The register file takes a total area of 55952 in Synopsys. Since the module contains only 256 bits of memory and the necessary logic involved to access the memory, the total area is divided by 256 to get the RBE value of 218. Third, from the die photo of a M*CORE processor shown below, we estimated that the CPU takes about 15% - 20% of the total die area with a small memory size. The ratio is used later in area calculation. This ratio is

in smaller in our simulation since the amount of memory simulated is more than that shown on the die photo.

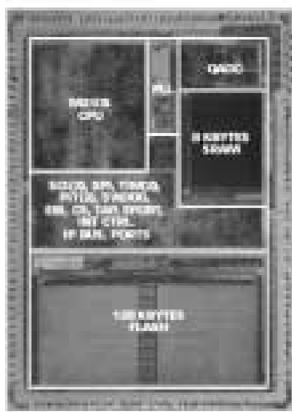


Figure 6.3: M*CORE die photo.

The power estimates depend heavily on the switching activity within the hardware components. For a rough estimate, Synopsys uses a pre-defined algorithm to compute a hypothetical switching activity. It assumes that all input pins to the hardware components have a 0.5 probability of switching. The probabilities are cascaded down the logic until the output pins. Since the circuits are built with NAND and NOR logics, the table below shows the probability of switch for a NAND gate and a NOR gate.

	NAND Gate		NOR Gate	
Switching Activity	0 -> 1	1 -> 0	0 -> 1	1 -> 0
Probability	0.75	0.25	0.25	0.75

Table 6.2:Switching Probability of a NAND and a NOR gate

Power consumption is computed based on this switching activity model. The power estimate given is broken up into dynamic power and cell leakage power. Cell internal power and net switching power make up the dynamic power estimate. Cell internal power represents the power consumed in gates and latches. Net switching power represents the power consumed in wires when data is in flight. The sum of the two types is the total power consumed for execution. When the hardware component is idle, the cell leakage power accounts for the power consumed to maintain the gates and latches. The Synopsys environment parameters for power-specific unit is listed in the table below.

Parameter	**Value**
Global Voltage	2.5V
Capacitance Unit	1 pf
Time Unit	1 ns
Dynamic Power Unit	1 mW
Cell Leakage Unit	1 nW

Table 6.3:Power-Specific Unit Parameters

CHAPTER 7

RESULTS FOR NANOPROCESSOR AS I/O CONTROLLER

In this section, the performance impact of adding the nanoI/O controller is examined in detail. In particular, the system bandwidth and processor utilization rate are discussed as the primary measures for performance. The area and power consumption of the nanoI/O controller are also evaluated. Further tradeoffs between cost and functionality are discussed.

7.1 System Bandwidth Analysis

The system bandwidth is affected dramatically by using the nanoI/O controller. The graph below illustrates the performance difference for six different benchmarks. The µCOS and multiple instances of a benchmark are compiled together and executed by the M*CORE simulator. Every instance of the benchmark is an individual task. The number of tasks is varied to see the impact of different workloads on the OS. Because the software static scheduler assigns unique priority levels to tasks, the OS executes them in a predetermined order. When one task finishes execution or is interrupted, the OS context switches to another task.

The graph below shows the system bandwidth with and without nanoI/O controller.

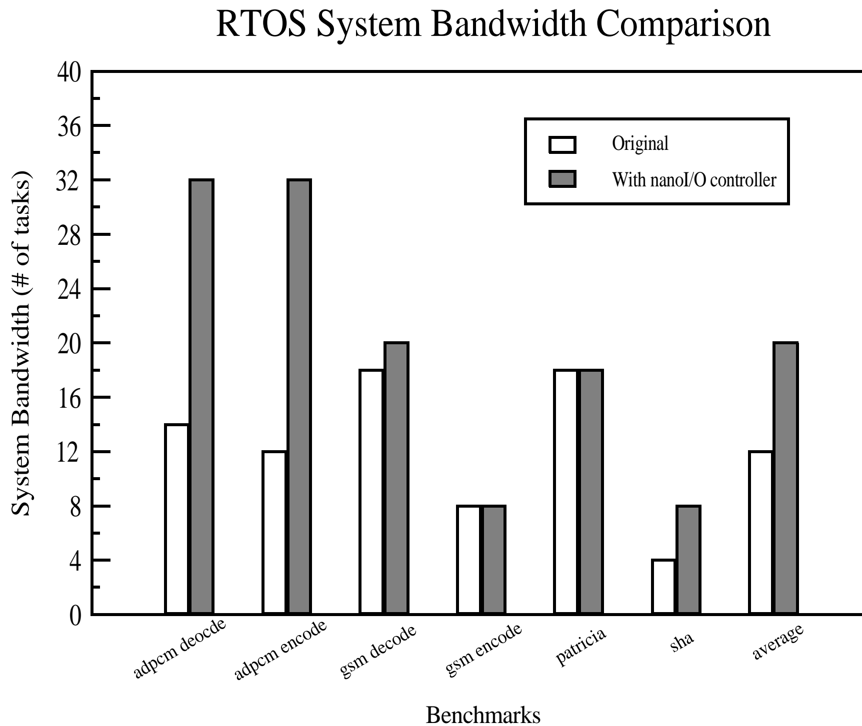


Figure 7.1. Maximum System Bandwidth Comparison for 6 benchmarks. The first bar of each benchmark shows the maximum sustained workload for the original system setup. The second bar shows the maximum sustained workload for the system with the nanoI/O controller.

The horizontal axis represents the six benchmarks and their average. The vertical axis represents the maximum number of tasks the system can sustain. Without the controller, the OS can handle less than 14 tasks of ADPCM DECODE. When there are 14 or more tasks, the OS starts to miss task deadlines and drops some tasks from execution schedule. This is the point at which the system overloads. It indicates that the system is working

at the maximum rate and fails to complete all tasks within allowed time. When the nanoI/O controller is used, the system can now handle up to 32 tasks. When a task makes an I/O request, it is put onto a wait queue while the nanoI/O controller fulfills the request. Since the CPU doesn't have to service the request anymore, the CPU is freed and can execute another task.

Similarly, the graph shows that ADPCM ENCODE and SHA both have large amount of I/O operations. In the case of ADPCM ENCODE, every iteration of the benchmark reads in 295KB PCM samples and produces a 73KB ADPCM samples at the output port. ADPCM DECODE performs the reverse computation, reading in the ADPCM samples and produces the PCM samples. Thus, the encoder and decoder have the same amount of IO communications even though the computation portions differ. Likewise, every iteration of the SHA algorithm reads a 300KB input and produces a 20 bytes output message. Therefore, ADPCM codec and SHA are regarded as communication intensive benchmarks.

On average, communication intensive benchmarks have fairly large request length. The cost of the nanoI/O controller can be amortized over the large data size. An IOread or IOwrite request adds 2 additional context switches to the system; one for blocking the current task and one for resuming the current task. The overhead is constant for the functions. The

overall cost can be reduced by fewer I/O requests with larger access size per request. In the case of ADPCM ENCODE, the request sizes are 2000 bytes for IOread and 500 bytes for IOwrite. ADPCM DECODE requests 500 bytes for IOread and 2000 bytes for IOwrite. Consequently, they benefit the most from the nanoI/O controller. When the system runs ADPCM ENCODE with the nanoI/O controller, the sustained workload increased from 12 tasks to 32 tasks, which represents an increase of 167% in system bandwidth. Similarly with ADPCM DECODE, the system bandwidth increased by 129%.

In the case of SHA, a single IOread request asks for a buffer of size 4096 bytes, which is the largest request size of all benchmarks. However, the system bandwidth only increased by 100% with SHA. This is because only one nanoI/O controller is simulated. When the nanoI/O controller services a request, any additional requests from the system must wait until the controller finishes the current request. Therefore, using one nanoI/O controller limits the performance gain in this case. Performance can be improved by using more nanoI/O controllers. However, when multiple controllers are used, synchronization among memory accesses must be coordinated.

The graph also shows that less communication intensive benchmarks do not benefit as much from the nanoI/O controller. Tasks like the GSM

ENCODE and PATRICIA do not see a difference in system bandwidth. This is because the overhead created by more context switches outweighs the benefit of fast and direct I/O operations. These tasks require the system to perform more computation than communication, thus making these benchmarks to be computation-intensive benchmarks. All communication requests are of small size, average only about 10 bytes of data for each request. The amount of time saved for accessing the requested data is not enough to balance the additional time spent in context switching.

To improve the performance, I/O operations with small data size do not block the calling task. Since the access size is small, it is better for performance reasons that the system stays idle and waits for the result. Thus, functions such as IOgetInt and IOputInt do not block the calling task for a 4-byte data.

Similarly optimization can be done in IOread and IOwrite by specifying a threshold for data size. If the total access size is under the threshold, the requesting task simply sits idle and waits for the result; otherwise, the requesting task blocks on the I/O request. The threshold can be defined at system initialization to provide flexibility to the system.

On average, the nanoI/O controller improves the system bandwidth from handling 12 tasks to 20 tasks, which is an increase of 67%. To under-

stand the impact more thoroughly, a breakdown of processor usage is analyzed next.

7.2 Processor Utilization Analysis

The processor usage is broken into categories. The categories of interests are process scheduling, interrupt handling, I/O communication, and interprocess communication (IPC). Process scheduling includes OS functions that are responsible for finding the next ready-to-run task and context switching to it. Interrupt handling includes functions that process and manipulate interrupts, such as ISRs and interrupt masking routines. As the IOread pseudo-code in chapter 4 shows, disable and enable IRQ are interrupt manipulation functions for critical sections. I/O communications includes functions for accessing and storing I/O data. IPC includes functions that manage semaphores, message queues, and message boxes. These categories are of interests because the nanoI/O controller may impact the

execution time spent in these categories. The Idle time represents the amount of time the system is available for more work.

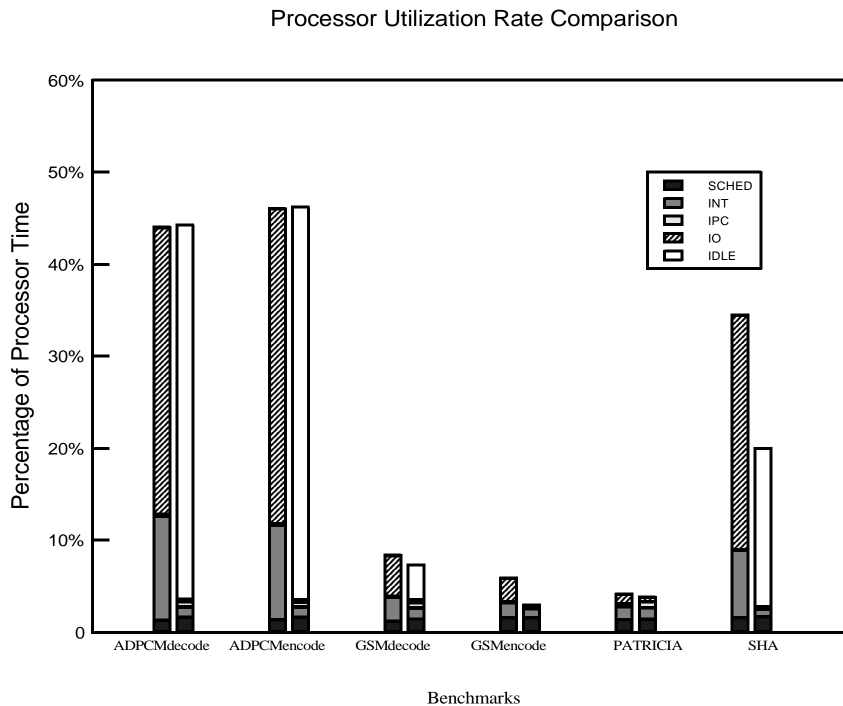


Figure 7.2. Processor Utilization Rate Breakdown for six benchmarks. The first bar for each benchmark is for the original system setup running the maximum sustained workload. The second bar is for the system with the nanoI/O controller running the same workload.

The graph above illustrates the processor usage breakdowns for all benchmarks. The workload used is the maximum sustained workload for the original setup, that is, 14 tasks for ADPCM DECODE, 12 tasks for ADPCM ENCODE, etc. The first bar for each benchmark represents the processor utilization rate for the original system without the nanoI/O con-

troller. The second bar is the same workload running with the nanoI/O controller. All other test parameters are kept the same.

The system executes other functions that don't fall under the above categories as well. For instance, the time for 14 task initialization takes about 30% of total execution time. However, since the nanoI/O controller has no impact on the initialization time, it is omitted from the comparison graph. In addition, time spent in user applications is unaffected by the nanoI/O controller as well. Therefore, the total amount of processor time does not sum to 100%.

For ADPCM DECODE, under original setup without the nanoI/O controller, the system is overloaded at 14 tasks. It means that the processor can not complete execution for the 14 tasks within the periods of some of the tasks. Consequently, the processor is never idle in this case. The most significant amount of time is used in I/O communications which accounts for almost 31% of total execution time. Along with I/O communication, interrupt also takes a great deal of processing time, over 11% of total execution time. As shown in the pseudo-code of IOread, every byte of data read is accompanied by a pair of interrupt disable and enable functions. Under the situation where multiple tasks share an I/O port, this operation ensures that the data being moved by one task is not corrupted by another. A similar situation arises in IOwrite as well and the same solution to dis-

able interrupt is used when data is being written out to a port. Therefore, the total number of interrupt masking calls is proportional to the total number of bytes communicated.

When nanoI/O controller is used, both the I/O overhead and the interrupt overhead are eliminated. Interrupt masking functions are called only once per request, and the overhead of setting up the controller becomes constant. Even though the contoller's response time to request is still proportional to the data size, the operation is performed in hardware and can achieve much smaller delay. Thus, the data size has a minimal impact on the growth of the response time. As a result, in the case of ADPCM DECODE, the processor is idle over 40% of the time with the use of nanoI/O controller. The percentage of idle time is approximately the sum of I/O communication and interrupt overhead in the original setup. Consequently, the processor can support 18 more tasks as shown earlier.

For GSM ENCODE and PATRICIA, the effect of nanoI/O controller is hardly felt. As mentioned in section 6.1, these benchmarks are computation-intensive. The I/O operations are of frequent access and have small access size. This characteristic accumulates the cost of context switching and outweighs the benefit of using nanoI/O controller. The graph confirms the reasoning. I/O and interrupt are of a small percentage of total processor usage, roughly 5% of total execution time combined for GSM ENCODE.

Using nanoI/O controller reduces the time down to about 2%. However, more time is spent in context switching, process scheduling, and IPC management. Thus the benefit and cost of using the nanoI/O controller cancel each other out in this case. Out of all benchmarks, PATRICIA has the worst utilization saving. This is because the benchmarks calls IOgetInt and IOputInt for integer inputs. Since these IO functions still block the processor, little saving is achieved in utilization rate.

From the graph, it's shown that benchmarks that are communication-intensive, like ADPCM codec and SHA, benefits a great deal from the nanoI/O controller. On the other end of the spectrum, benchmarks like GSM codec and PATRICIA are computation-intensive and don't see a great performance boost. To improve utilization rate, these benchmarks can combine some I/O operations to achieve a larger request size, thus amortizing the cost. In any case, it is important to notice that at no time the nanoI/O controller lowers the system performance.

7.2.1 Future Optimization

For simplicity reasons, only one nanoI/O controller is simulated and one I/O port is shared among all processors. If the controller is servicing a request when another one comes in, the later request must wait until the earlier request finishes and then takes control of the nanoI/O controller.

When this occurs, the waiting task is not swapped out the processor and the CPU time is wasted. To get better performance, it's possible to build a request queue in the nanoI/O controller to record the request. The task can then be blocked waiting for its request to be fulfilled. The diagram below illustrates the idea of the request queue. This will save more processor time from being wasted. However, the cost and complexity of the nanoI/O controller will increase due to the management of the queue structure.

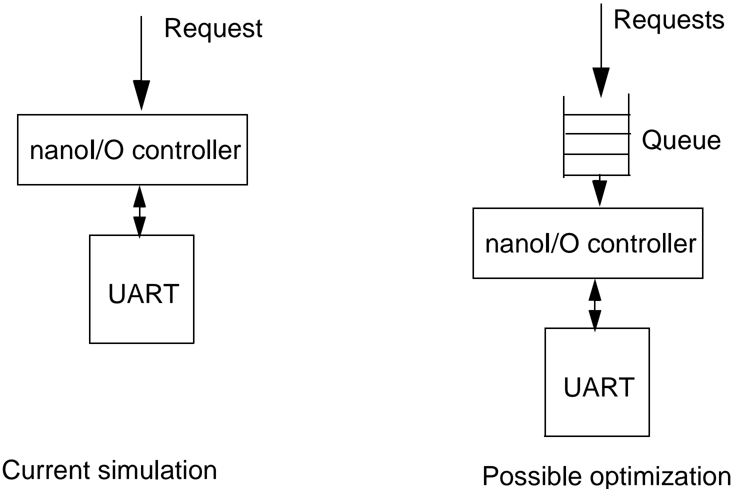


Figure 7.3. Possible optimization to improve the processor utilization rate for single nanoI/O controller and UART.

Another possible optimization is to have dedicated I/O ports. Without the nanoI/O controller, interrupt masking functions are necessary in the case where several tasks share an I/O port. For every byte transferred, atomicity must be guaranteed in order to ensure data correctness. However, if each task has a designated I/O port (as illustrated below), the interrupt

masking functions can be moved to the outside of the loop (as discussed in section 4.2). Thus, it would save execution time since time spent in interrupt masking is constant. However, it is still beneficial to use the nanoI/O controller since the amount of time spent in I/O operations is not affected by the number of I/O ports. In addition, the queue structure can also be use here to further improve performance.

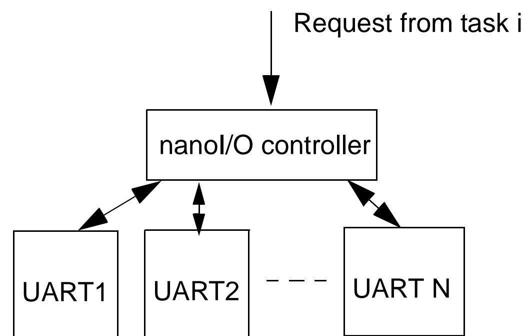


Figure 7.4. Configuration for a single nanoI/O controller and multiple UARTs.

To achieve better processor utilization rate, multiple nanoI/O controllers can be used. This eliminates the need to block when one nanoI/O controller is busy. Therefore, as many requests as there are nanoI/O controllers can be satisfied simultaneously. Each nanoI/O controller can control one or more I/O ports. However, since multiple memory accesses can be generated concurrently, memory bus synchronization must be done to ensure data coherency.

7.3 NanoI/O Controller Cost

Cost considerations include die area and power consumption. Die area directly impacts the manufacturing cost. The additional power consumed by the nanoI/O controller puts more pressure on batteries and can shorten battery life. In addition, the nanoI/O controller can generate more heat and complicate the cooling mechanism for devices. These concerns are as important as performance for embedded system designers. In this section, die area and power consumption issues are addressed. Heating characteristics is omitted since the study would require setups beyond the current simulation environment.

7.3.1 Die Area Analysis

With 72 bytes of total required storage, the nanoI/O controller has a reported area of 222947. There is no unit for the area. It is technology dependent and can be compared with other hardware component synthesized by the same toolset. However, to get an idea of how much area the controller requires, several references are obtained. First, a single NOR or NAND gate has an area of 17. Second, a RBE has an area of 218. In addition, the relative size of the M*CORE CPU is 899096. Thus, the controller is about 24.8% of the CPU size. To get a sense of the amount of logic involved in implementation, we estimate the number of logic gates used.

The area information provided by Synopsys tools includes area information for gate cells only. This is the amount of area ASIC gates occupy using the specificied technology. Thus gate count is obtained by dividing the gate cell area by the area of a NAND gate. Since the nanoprocessors are targeted for FPGA technology and FPGA vendors often provide sizes in units of system gates, we would like to find the relationship between ASIC gates and FPGA gates. Even though the gate conversion is vendor-specific, the general rule of thumb is that FPGA system gates are 2 to 3 times of that in ASIC gates [28]. Further, the port-mapping table sizes can be changed to measure effects on area. With 64 bytes for both tables, the nanoI/O controller is capable of supporting up to 16 ports. In addition, the input and output ports for one task do not have to be the same. If the system do not support 16 ports, the table sizes can be reduced. For example, the M*CORE system studied here support only 2 I/O ports. This would only require 1 bit per table entry for both 64-entry tables. This adds to a total of 16 bytes. In typical embedded systems, 4 to 8 I/O ports are supported. The table below summarizes the area cost for different table sizes. The area measures obtained in Synopsys are given in the first column. The M*CORE area is listed for comparison. The Synopsys estimates are translated into RBE counts in second column. To compare to the CPU, the ratios of the area are given. To provide a estimate for the logic complexity, a gate count estimate

is provided. Furthermore, to get a sense of the needed size of an FPGA to implement a nanoI/O controller, estimates in FPGA system cells are provided assuming 2 FPGA system cells are equivalent to 1 ASIC gate.

Hardware Component	**Synopsys Area**	**RBE counts**	**Relative Area to CPU**	**Gate Count**	**FGPA System Cell Estimate**
MCORE CPU	899096	4124		36224	72448
nanoI/O controller 16 ports	222947	1023	24.8%	6079	12158
nanoI/O controller 8 ports	191002	876	21.2%	5153	10406
nanoI/O controller 4 ports	162749	747	18.1%	4446	8892
nanoI/O controller 2 ports	131230	602	14.6%	3546	7092

Table 7.1: Area Estimate for the nanoI/O Controller

7.3.2 Power Consumption Analysis

The power consumption estimates are also obtained for the above nanoI/O controller configurations. The amount of power consumed is directly related to the switching activities in the circuit. Synopsys provides a switching activity input using a probability model, discussed in chapter 6. The table below lists the power estimates for the nanoI/O controllers configured in Table 2.

Hardware Component	Dynamic Power (mW)	Cell Leakage Power (nW)
M*CORE	25.642	2354.77
nanoI/O controller 16 ports	3.0669	416.56
nanoI/O controller 8 ports	2.5377	359.58
nanoI/O controller 4 ports	2.0228	302.58
nanoI/O controller 2 ports	1.4996	246.75

Table 7.2: Power Estimates for the nanoI/O controller

Even though the nanoI/O controller adds 12% of the CPU dynamic power, tasks can execute faster with the nanoI/O controller since the communication component and the computation component of a task can be execute in parallel. This implies that the total execution time for a fixed number of task can be shortened. Thus, shorter execution time can translate into savings in energy consumption.

The nanoI/O controller can improve the system performance by offloading the communication requests from user applications. For communication-intensive benchmarks, the nanoI/O controller can increase the system bandwidth dramatically. The nanoI/O controller is affordable as well. A nanoI/O controller that supports 16 I/O ports increases the total die area by approximately 10%. For systems that support fewer ports, the impact of the nanoI/O controller on die area is even smaller. In addition, the nanoI/O

controller does not increase the system power consumption significantly and can help saving total energy consumption.

CHAPTER 8

RESULTS FOR NANOPROCESSOR AS SCHEDULER

This section presents results showing the effect of the nanoScheduler on the system performance and cost. First schedulability is studied to show that the EDF algorithm is capable of producing a feasible schedule when the static priority scheduler fails to do so. Then the performance impact on the system is discussed by analyzing the breakdown of processor overhead. Lastly, die area cost and power consumption estimate are presented.

8.1 Schedulability Analysis

The hardware scheduler supports both a static priority scheduling scheme and an Earliest Deadline First (EDF) scheduling scheme. The performance analysis is done on four setups: static-priority based scheduling in software, static-priority in hardware, EDF scheduling in software, and EDF in hardware. The tasks and their periods are chosen such that the static-priority scheduling scheme would fail and the EDF scheduling scheme would succeed. Therefore, the workload is chosen to be a dummy task running with a period of 10 ms mixed with the same task running at a period of 17 ms. The dummy task increments a counter in a loop. The runt-

ime of the task is fixed at 45% of the period. For the two tasks that run at 10 ms and 17 ms, the runtimes are at 4.5 ms and 7.65 ms. The below diagram shows why the static priority scheme would fail.

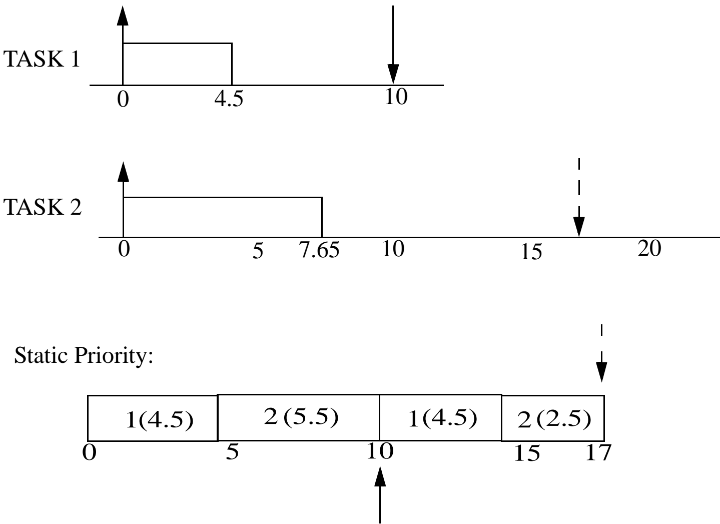


Figure 8.1. Workload scheduling activity using the static-priority scheduler.

Under static priority scheduling, task 1 has a smaller period, thus becomes the higher priority task. When task 1 runs to completion, it turns the OS to task 2. At time 10, task 1 starts a new period and becomes ready to run again. Since it has a higher priority, task 1 runs to task completion until time 14.5, at which time task 2 resumes execution. When the deadline of task 2 expires at time 17, the system notices that task 2 has only executed for 7 ms and has not completed its runtime, and thus is late.

In the same situation, the EDF scheduling algorithm can schedule the two tasks successfully. Like with static priority algorithm, the task 1 starts

execution at time 0. However, when task 1 is released at 10, the EDF scheduler compares the deadline of the two tasks. Task 1 has a deadline at time 20, while task 2 has deadline at time 17. The EDF scheduler decides that task 2 should have the higher priority and continues executing until task completion at approximately time 13. The graph below illustrates the

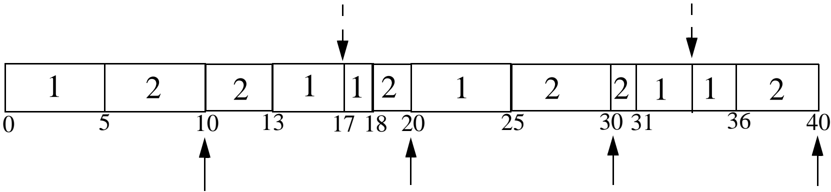
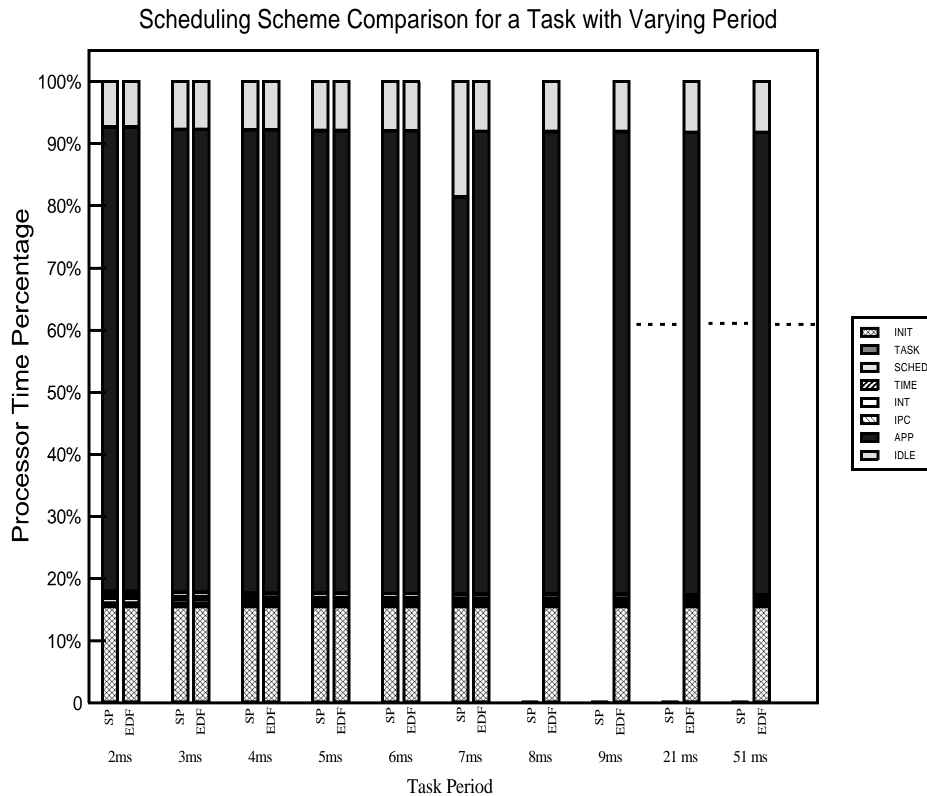


Figure 8.2. Workload scheduling activity using the EDF scheduler.

EDF scheduling activities. The output obtained from our simulation environment also verifies that the EDF algorithm achieves the valid schedule shown above.

The next result shows the schedulability study using the two tasks mentioned above. We fix one task at 10 ms period, and vary the period of the other task to test when the static priority scheduling scheme fails.



The graph shows that varying the period of a task does not change the percentage of processor utilization. This is because the runtime is proportional to the task period. When the varying task reaches a period of 7 ms, the static priority scheme fails to schedule the two tasks. This is indicated by the drop in user application time. On the other hand, the EDF algorithm never fails to schedule the two tasks with different periods. As the varying

period increases, the downtime of the task also increases. This makes scheduling easier. Our result indicates that the system idle time at a larger period is slightly longer than the idle time at a smaller period. This implies that the scheduling time converges to a lower bound and thus the EDF algorithm will not fail in this situation.

8.2 System Bandwidth Analysis

The EDF algorithm is an optimal algorithm, capable of achieving a feasible schedule if such a schedule exists. Consequently, by scheduling more tasks than a static priority algorithm, the EDF scheduler can help increase the system bandwidth. Our next simulation result verifies this by running a benchmark similar to the one mentioned above. We study all four scheduling schemes; they are static priority scheduling in software, static priority scheduling in hardware, EDF scheduling in software, and EDF scheduling in hardware. The benchmark increments a counter inside of a loop. The benchmark is cloned to multiple tasks. However, the runtime of each task is fixed at 0.2 ms. This benchmark has a small runtime to allow execution at a high frequency. Further, the runtime is well-defined with little variations. This simplifies the performance evaluation process. The first 10 tasks are running at a period of 3 ms, next 10 tasks are running at a period of 4 ms, and the rest is running at a period of 5 ms. Varying the task

periods introduces diversity into the workload and makes the workload representative of real-world applications. By choosing a benchmark that has a short runtime and composing a workload that consists of frequent tasks, we were able to execute the scheduler repeatedly. Thus the frequent scheduling magnifies the impact of scheduling schemes on system performance. The result is shown below.

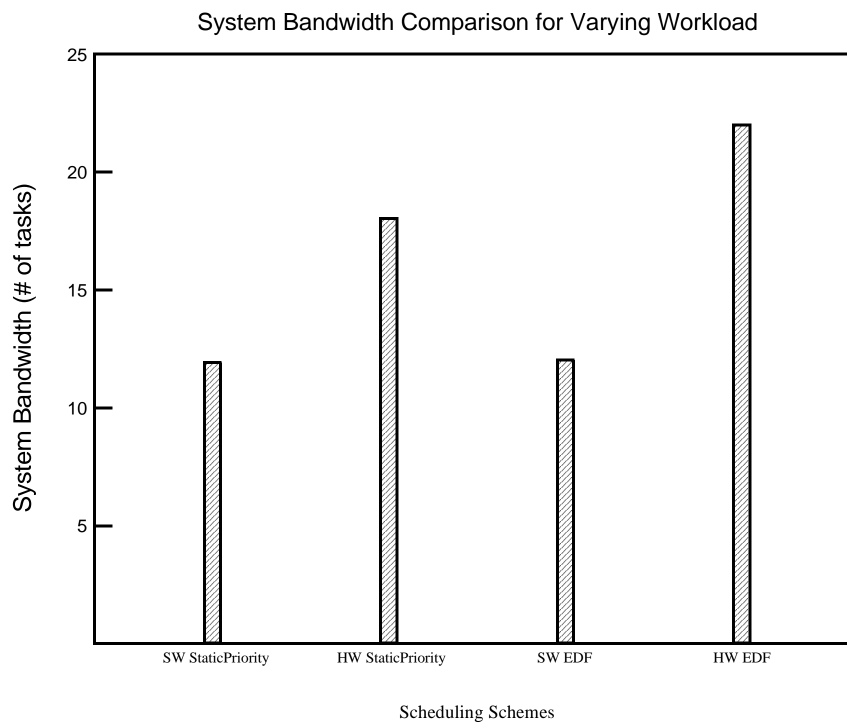


Figure 8.3. System Bandwidth Comparison using Different Scheduling Schemes.

The result shows that a system with software scheduler can handle up to 12 tasks. Any additional tasks would overload the system and are either executed late or dropped from the schedule. Note that this does not imply

that the scheduler fails to schedule the tasks properly. The system is overloaded due to the high overhead software schedulers create. As a result, the system is overloaded before the software schedulers actually fail. On the other hand, a system running the hardware static priority scheduler can handle up to 18 tasks. The static priority scheduler can not manage more than 18 tasks. In comparison, the EDF algorithm is able to take advantage of the runtime characteristics of the tasks and schedule up to 21 tasks. As a result, the algorithm can break up the processor time more efficiently and produce an optimal schedule. The hardware static priority scheduler is able to increase the system bandwidth by 50%. The dynamic EDF algorithm can increase the bandwidth by an additional 19%, achieving a total increase of 75% of system bandwidth compared to the software schedulers. To understand the performance gain thoroughly, we discuss the processor utilization breakdowns in the next section.

8.3 Processor Utilization Analysis

Using the workload described, we obtained processor utilization breakdown to demonstrate that the hardware EDF implementation can improve system performance significantly. The types of overhead shown are system initialization, task management, process scheduling, time management, interrupt, IPC, user application, and system idle time. Schedul-

ing, interrupt, and IPC are the same categories discussed in chapter 7. System initialization includes functions that create tasks, set up stack spaces, and prepare the system for execution. Task management includes functions for creating and deleting tasks and changing task states. Time management includes functions that maintain the software-kept timer, traverse the list of tasks to decrement delay fields, and retrieve and set time for users. User application is the amount of the time the system spent executing application code. Finally, system idle time is the amount of time the system is available for more work.

The figure below shows the processor utilization breakdown.

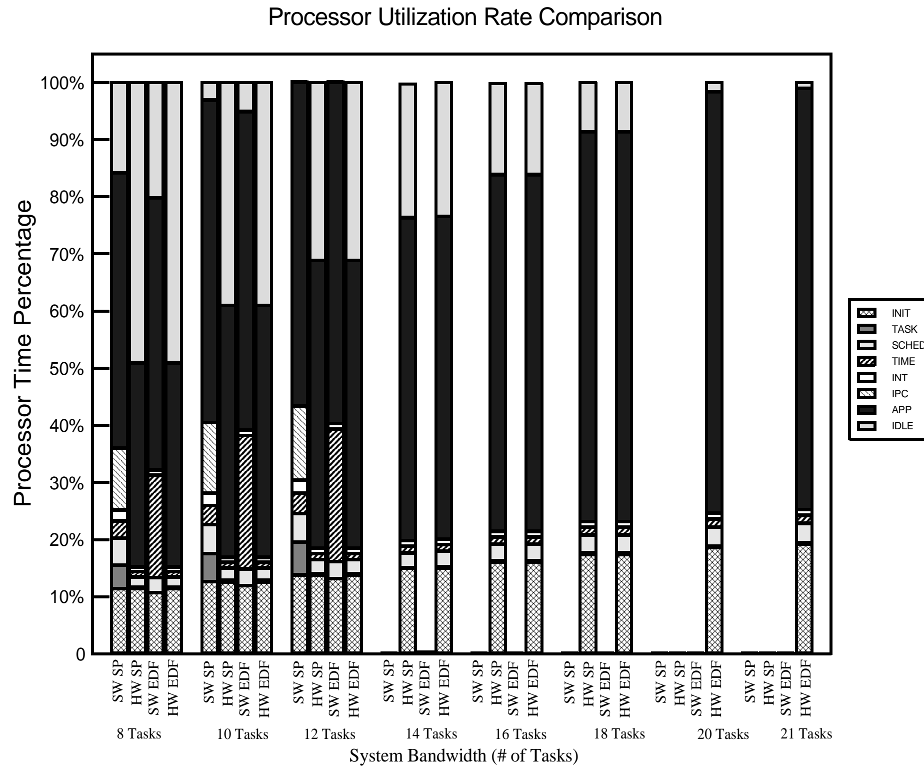


Figure 8.4. Processor Utilization Comparison for different scheduling schemes.

The horizontal axis represents the workload. Each group of bars represents software static priority scheme, hardware static priority scheme, software EDF scheme, and hardware EDF scheme. Each bar is segmented into 8 categories showing the percentage of processing time each category takes. Beyond 12 tasks, the systems that execute software schedulers are overloaded due to the high system overhead. Thus only the processor utilization breakdowns for hardware schedulers are shown after 16 tasks. Similarly, the processor utilization breakdown for the hardware static-priority

schedulers is not shown after 18 tasks since the scheduler fails to achieve a feasible schedule beyond 18 tasks.

For a given workload, the system initialization time and the user application time are not affected by the scheduling schemes. As the number of tasks increases, both initialization and user application times grow linearly.

The overheads from the software schemes are on average higher than the hardware schemes by at least 20%. In the case of software static priority scheduler (SW SP) executing 8 tasks, the system spends approximately 25% of total execution time in IPC, interrupt, time management, scheduling, and task management. This is due to the fact that a high priority task must be execute to enforce the periods of all user tasks. Recall in chapter 6, we discuss the special task that is execute every timer tick. The task increments a counter for every user task. If the counter matches the specified period of a user task, the user task is made ready to run. After the user task executes, it is blocked waiting for the next period. This special task that manages user tasks is needed because μCOS does not have a sense of periodic tasks. As the workload increases, the overhead due to the special task grows linearly. When the system executes 12 tasks, the overhead is nearly 30%.

In the case of the software EDF algorithm, the special task is not needed. This eliminates the overhead associated with the task. However, the time management has increased significantly. The software EDF (SW EDF) keeps track of two linked lists. One is for ready tasks and the other is for other tasks that are blocked, delayed, or suspended. The deadline values are absolute time instead of relative deadlines. Both lists are sorted according to the absolute deadline values. On every timer tick, the blocked list is traversed to check if any task becomes available to run. If such a task exists, it is taken off the blocked list and put into the ready list. When task state is changed, the task is taken off the ready list and put into the blocked list. The management of the two lists in software contributes to the overhead seen in time management. As the number of tasks increases in the system, the overhead for time management also increases linearly, reaching over 25% for a workload of 12 tasks.

For the hardware schedulers, the system overhead is insignificant, contributing less than 7% of the total execution time. Furthermore, the system overhead stays almost constant as the workload increases. When the workload reaches 21 tasks, the overhead has grown from 4% to 7% of total execution time.

Our results show that the hardware schedulers are good because they reduces the system overhead associated with time management, IPC, inter-

rupt and task context switching. The EDF algorithm is better than static priority algorithm because more tasks can be executed, thus improving system bandwidth. As a result, the proposed nanoScheduler, as a hardware EDF scheduler, has the best performance out of the four schedulers studied.

8.4 Die Area and Power consumption Analysis

As discussed in chapter 5, the 63-task scheduler takes a total storage area of 372 bytes. The majority of the total storage area is for storing task state information. This can be stored in the main memory. Inside of the nanoScheduler, only 2 data structures are needed. They are used for comparison operations in list traversal. The table below lists the area estimate for a 64 task scheduler. The Synopsys area estimates are provided in the first column and the RBE counts in the second column. To make a comparison, area estimate of the M*CORE processor is also included. The gate count is provided to estimate the logic complexity. The FPGA system cell estimate is provided to show the required size of an FPGA needed to implement a nanoScheduler.

Hardware Component	Synopsys Area	RBE counts	Relative Area to CPU	Gate Count	FPGA System Cell Estimate
M*CORE	899096	4124		36224	72448
64 task scheduler	49870	229	6%	1668	3336

Table 8.1: Die Area Estimate for Hardware Scheduler

The table below provides the power consumption estimates for the 64-task schedulers. The M*CORE power consumption is provided for reference.

Hardware Component	Dynamic Power (mW)	Cell Leakage Power (nW)
M*CORE	25.642	2354.77
64 task scheduler	0.411	93

Table 8.2: Power Consumption Estimate for Hardware Scheduler

Since the task scheduler only contains 2 data structures, it has a very small impact on both die area and power consumption.

CHAPTER 9

CONCLUSION

9.1 Summary

In this work, reconfigurable hardware accelerators, nanoprocessors, are proposed to specifically assist the processor in reducing overhead incurred by embedded systems. Nanoprocessors can be programmed to meet application specific requirements. They can be realized using Systems-on-chip with on-chip FPGAs. In this work, we demonstrate two uses for nanoprocessors, to increase system bandwidth and processor utilization rate.

Two system performance bottlenecks are identified. In particular, high-frequency I/O ports require the CPU to save every data byte that comes in from an I/O port. This operation can incur a significant overhead. Another performance bottleneck comes from the process scheduler. The scheduler returns the next available task for execution. As one of its responsibilities, the scheduler must examine every task in the system and update delay information on every timer interrupt. This operation also

incurs a significant overhead. Furthermore, the overhead increases linearly with respect to the number of tasks.

As a result, nanoprocessors are proposed, as hardware accelerators, to off-load some of the OS functionality to hardware. In this work, one specific function defined is the I/O controller. An I/O controller is similar to an DMA controller, managing communication traffic from the I/O ports to the main memory. The I/O controller operates in parallel as the CPU. Another identified nanoprocessor function is the processor scheduling. Since static scheduling can not achieve 100% processor utilization rate, an EDF algorithm scheduler which can achieve full processor utilization is implemented in hardware.

The nanoprocessors are treated as on-chip peripherals. There is a special memory bus dedicated for the nanoprocessors use. The nanoprocessors also have access to the bus that connects the interrupt controller. The nanoI/O controller uses an interrupt to inform the CPU about the results of I/O requests. The nanoScheduler does not use the interrupt and returns results immediately while the processor is waiting.

From our simulation enviornment, nanoprocessors show a good performance increase. For the nanoI/O controller, both system bandwidth and the processor utilization rate increases dramatically for the chosen benchmarks. For the nanoScheduler, the processor utilization rate is increased

because the EDF algorithm can produce a better schedule than the static priority-based algorithm. Further, the operating system overhead is also reduced by using the hardware scheduler.

Cost of the nanoprocessors are also evaluated. Specifically, die area increase and power consumption estimates are studied. A simulation enviornment is setup using Verilog HDL. Synthesizable programs are obtained using the Synopsys toolset. Area and power estimates are obtained from the synthesis results. Area estimates show that the nanoI/O controller increases the die area by 10% if the system supports up to 16 ports. For a smaller number of ports, the nanoI/O controller area is even smaller, only reaches 5% of the total die area for a 2-port system. The nanoScheduler keeps track of state information for all system tasks. Since the task information are stored in the main memory, very little memory is required on the nanoScheduler. Thus, the nanoScheduler logic increases the total die area only by 1.1%. Together, the nanoprocessors increase the total die area by 6%.

Power consumption estimates show that the nanoI/O controller that supports 16 ports uses about 3 mW. This is not a significant increase considering that the M*CORE alone consumes 25 mW without any memory. The nanoScheduler consumes 411 uW, which is negligible. However, since the memory is now dual-ported, power consumption for the entire will

increase. However, other studies have shown that the Systems-on-Chip do not require significant amount of extra power. In additon, programmable hardware accelerator can help to reduce the total execution time dramatically and thus save on energy consumption.

To conclude, the nanoprocessors as reconfigurable hardware accelerators, are designed to off-load high-overhead tasks from the CPU into hardware. Based on our simulation study, using nanoprocessors as an I/O controller and as a hardware scheduler can increase system bandwidth and processor utilization rate, and does so while minimizing the die area and power consumption increase.

9.2 Future Work

In this work, the simulation enviornment is setup and particular implementations of the nanoprocessors are completed. However, the implementations can be optimized. For the nanoI/O controller, chapter 7 outlines several optimizations that could be done to improve system performance. For the nanoScheduler, implementations other than the linked-list can be explored and compared. Detailed power estimate models can be used for the nanoprocessors. The current power estimates are obtained from a hypothetic switching activity level. More realistic switching should be used in obtaining accurate power consumption. In addition, other areas

of the operating system overhead should be investigated and more use of the nanoprocessors can be explored.

BIBLIOGRAPHY

[1] Taiwan Semiconductor Manufacturing Company Ltd, http://www.tsmc.com/uploadfile/whitepaper/22/index.html.

[2] fCoder Group International, http://www.lookrs232.com/serial_port/serial_port_vs_parallel_port.htm.

[3] C. Liu and J. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." *Journal of the Association for Computing Machinery (JACM)*, Vol. 20, No. 1, January 1973, pp. 46-61.

[4] J. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

[5] J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. R & D Books, Lawrence, KS, 1999.

[6] *M*CORE, MMC2001 Reference Manual*. Motorola, 1998.

[7] *M*CORE microRISC Engine Programmer's Reference Manual*. Motorola, 1997.

[8] µCOS-II, http://www.ucos-ii.com.

[9] C. Lee, M. Potkonjak, and W. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, Research Triangle Park, NC, December 1997.

[10] M. Guthaus, J. Ringenberg, et al. "MiBench: A Free, Commerically Representative Embedded Benchmark Suite." In *IEEE 4th Annual*

Workshop on Workload Characterization, Austin, TX, December 2001.

[11] J. Adomat, J. Furunäs, L. Lindh, and J. Stärner. “Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems.” In *Proceedings of Eighth Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 1996, pp. 164-168.

[12] R. Dick, G. Lakshminarayana, A. Raghunathan, and N. Jha. “Power Analysis of Embedded Operating Systems.” In *Proceedings of the 37th Design Automation Conference*, Los Angeles, CA, June 2000.

[13] P. Kuacharoen, M. Shalan, and V. Mooney III. “A Configurable Hardware Scheduler for Real-Time Systems.” In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, June, 2003.

[14] S.W. Moon, J. Rexford, and K. Shin. “Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches.” *IEEE Transaction on Computers*, Vol. 49, No. 11, November 2000, pp. 1215-1226.

[15] M. Wirthlin, B. Hutchings, and K. Gilson. “The Nano Processor: a Low Resource Reconfigurable Processor” In *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April, 1994, pg. 23-30.

[16] J.M. Lee. *Verilog Quickstart, 3rd edition*. 2002, Kluwer Academic Publishers.

[17] P. Kurup and T. Abbasi. *Logic Synthesis Using Synopsys, 2nd edition*. 1997, Kluwer Academic Publishers.

[18] Synopsys On Line Documentation, Synopsys Inc.

[19] Cadence Design Software Documentation (CDSDoc), Cadence Inc.

[20] Atmel Inc, FPSLIC data sheet. http://www.atmel.com/dyn/resources/prod_documents/1138S.PDF

[21] Altera Inc, Excalibur data sheet. http://www.altera.com/literature/ds/ds_arm.pdf

[22] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. "Using On-Chip Configurable Logic to Reduce Embedded System Software Energy." *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, April 2002.

[23] D.B. Stewart, D.E.Schmitz, and P.K.Khosla. "The Chimera II real-time operating system for advanced sensor-based applications." *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1282-1295, November/December 1992.

[24] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. "The Performance and Energy Consumption of Three Embedded Real-Time Operating Systems." In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01)*, Atlanta, GA, November 2001, pp. 203-210.

[25] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. "The Performance and Energy Consumption of Embedded Real-Time Operating Systems." In *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1454-1469. November 2003.

[26] http://www.lifl.fr/west/courses/SoCdesign/problematique.pdf

[27] F. Vahid. "Recent Results at UCR with Configurable Cache and HW/SW Partitioning." http://www.cs.ucr.edu/~vahid/pubs/triscend02_sept.ppt. *Talk given at Triscend Corp*, Semptember 2002.

[28] QuickLogic pASIC 1 Family FPGA Datasheet. http://www.quicklogic.com/images/pasic1_datasheet.pdf