ABSTRACT

Title of Thesis:	ARCHITECTURAL SUPPORT FOR EMBEDDED OPERATING SYSTEMS
Degree candidate:	Brinda Ganesh
Degree and year:	Master of Science, 2002
Thesis directed by:	Professor Bruce L. Jacob Department of Electrical and Computer Engineering

This thesis investigates hardware support for managing time, events, and process scheduling in embedded operating systems. An otherwise normal content-addressable memory that is tailored to handle the most basic functions of a typical RTOS, the CCAM (*configurable content-addressable memory*) turns what are usually O(n) tasks into O(1) tasks using the parallelism inherent in a hardware search implementation. The mechanism is modelled in the context of the MCORE embedded microarchitecture, several variations upon μC/OS-II, a popular open-source real-time operating system and Echidna, a commercial real-time operating system. The mechanism improves the real-time behavior of systems by reducing the overhead of the RTOS by 20% and in some cases reduces energy consumption 25%.

This latter feature is due to the reduced number of instructions fetched and executed, even though the energy cost of one CCAM access is much higher than the energy cost of a single instruction. The performance and energy benefits come with a modest price: an increase in die area of roughly 10%. The CCAM is orthogonal to the instruction set (it is accessed via memory-mapped I/O load/store instructions) and offers features used by most RTOSes.

ARCHITECTURAL SUPPORT FOR

EMBEDDED OPERATING SYSTEMS

by

Brinda Ganesh

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
2002

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr Bruce Jacob for having given me this oppourtunity to work on this project and for all the time, help and encouragement that he has given me over the course of its development.

I would like to thank Dr Stewart who has given valuable input to the development of this idea.

I would also like to thank my colleagues at the university - Aamer , Paul, Zahran, Ankush, Tiebing, Lei, Dave, Vinodh , Anasua, Aneesh for everything from technical discussions , insight and suggestions to bicycle rides and cups of tea.

On a personal note I would like to thank my parents , my sister Rupa , my grandmother and Deepak who have believed in me always. I am very grateful to Himanshu for the invaluable advice he has given me over the past couple of years. Finally I would like to thank my housemates - Vijaya, Anu, Lakshmi, Sharmila who lived with me and accepted my indifferent cooking and mood swings when the going got tough.

TABLE OF CONTENTS

Acknowledgements v

List of Tables ix

List Of Figures x

1 Introduction and Motivation 1

1.1 Characteristics of Embedded Systems 1

1.2 Current Trends in Embedded Systems 3

1.3 Motivation 7

1.3.1 Support for Process, Time and Event Management . 10

1.4 Overview 12

2 Background: Real-Time Operating Systems 13

2.1 Real-Time Systems 13

2.1.1 Real-Time Terminology 14

2.1.2 Classification of real-time systems 16

2.1.3 Real-time system requirements 17

2.2 Real-Time Operating Systems 20

2.2.1 Scheduling 21

2.2.1.1 Terminology 21

2.2.1.2 Types of Schedulers 23

2.2.2 Time Services. 31

2.2.3 Inter process Communication and Synchronization 37

2.3 Related Work 40

3 CCAM: Description 42

3.1 CCAM Structure 42

3.2 CCAM Module Interface 48

3.3 Implementation 54

3.3.1 CCAM Energy Model 62

3.3.2 Die Area Cost. 64

4 Experiments 65

5 Results. 74

5.1 Overhead 74

5.1.1 Preemptive µC/OS Configurations 74

5.1.2 Non preemptive µC/OS configurations 81

5.1.3 Echidna 86

5.2 Operating System Functional Measurements 90

5.3 Jitter 97

5.3.1 Preemptive μCOS Configurations 97

5.3.2 Non preemptive mC/OS Configurations 104

5.3.3 Echidna 108

5.4 Response Time 111

5.4.1 Preemptive μCOS Configurations 111

5.4.2 Non preemptive μCOS Configurations 115

5.4.3 Echidna 117

6 Conclusion and Future Work 118

6.1 Conclusions 118

6.2 Future Work 119

Bibliography 120

LIST OF TABLES

CCAM Module Address Map 46

State Assignments for Run Status 52

CCAM Instructions 53

LIST OF FIGURES

Fig. 2.1. Ready List in μC/OS 27

Fig. 3.1. CCAM Structure 43

Fig. 3.2. State Machine representation of Run Status of a Task. 44

Fig. 3.3. Instruction Layout of CCAM instructions 51

Fig. 3.4. CCAM Tick Module for handling Timeout Delays..... 55

Fig. 3.5. Structure of Input Comparator Element....................... 59

Fig. 3.6. Enable Logic for Priority /Delay Sort.......................... 60

Fig. 3.7. Enable Signal for Release Event Priority Sort............ 61

Fig. 3.8. Comparator Network 62

Fig. 4.1. Experimental Setup... 68

Fig. 4.2. Jitter Example 70

Fig. 4.3. Response Time to External Interrupt 71

Fig. 5.1. Energy Consumption for preemptive μCOS configurations 75

Fig. 5.2. Utilization for preemptive μCOS configurations 79

Fig. 5.3. Energy consumption of non preemptive μCOS configurations83

Fig. 5.4. Utilization Overhead for non preemptive μCOS configurations85

Fig. 5.5. Energy Consumption of Echidna based configurations88

Fig. 5.6. Operation System Overhead for Semaphore Pend 91

Fig. 5.7. Operation System Overhead for Semaphore Post........ 92

Fig. 5.8. Operation System Overhead for Time Delay............... 93

Fig. 5.9. Interrupt Overheads 95

Fig. 5.10. Jitter for Preemptive μCOS configurations 100

Fig. 5.11. Jitter for Preemptive μCOS configurations with a system clock rate of 2 kHz103

Fig. 5.12. Jitter for nonpreemptive μCOS configurations.......... 106

Fig. 5.13. Jitter for Echidna based configurations 109

Fig. 5.14. Interrupt Response Time for Preemptive μCOS configurations112

Fig. 5.15. Response Time to Aperiodic Interrupts for Non preemptive μCOS configurations113

Fig. 5.16. Response Time to Interrupt for Echidna based configurations116

CHAPTER 1

INTRODUCTION AND MOTIVATION

The modern world is characterized by the integration of computing power into our lifestyles. The incorporation of “intelligence” into every device and gadget forms the basis of the ongoing computing revolution. This new wave is characterized by freeing computing power from a traditional desktop environment and making it an all pervasive yet invisible force. It is this demand for “hidden computing power” which has fuelled the growth of the embedded systems marketplace. Current estimates indicate that there are more than 5 billion embedded processors being used in the world[1] and that this number is constantly growing with nearly 4 billion dollars worth of micro controllers being sold annually.

1.1 Characteristics of Embedded Systems

Embedded System design factors in various issues which are not considered in the design of high performance desktop systems. These include:

Low Cost : A large number of embedded products including cell phones, electric shavers, toasters etc. are in markets where the end user is unwilling to spend a little extra for slightly better performance or a few extra features. Embedded designers are thus forced to design systems with optimal price/performance ratios and extract the required performance at the least cost. The result of this is that the resources available to the designer are minimal. For example an embedded system rarely has more than a few MB of memory available.

Low Power: Power consumption is another critical factor in the design of these systems. This is largely because most of these systems are battery operated and are expected to remain active for long periods of time. e.g. pacemakers, sensors, cell phones, PDAs. They are often deployed in harsh uncontrolled environments.eg digital sensors. The size of these systems is often a deterrent to incorporating the cooling systems required. Thus low power is an essential feature.

Predictability: Owing to the large number of real-time applications that most embedded systems run it is of paramount importance that they exhibit predictable behavior. This implies that under all possible events and conditions one can predict the behavior of the system. In these systems the underlying hardware has to guarantee that any code running on it will have the same execution overhead every time it is run. The software is usually designed for the worst case scenario and such that the overheads are deterministic.

Responsiveness: Embedded applications, like control system applications, are event driven and need to respond quickly to these events. Such systems rely heavily on interrupts and have to designed to have low interrupt response times as well as low interrupt processing overheads.[9]

Temporal Accuracy: Distributed embedded applications which manage remote databases and network devices all require highly precise time granularities [8]. The resolution and accuracy is expected to be in the order of microseconds for most of these applications.

1.2 Current Trends in Embedded Systems

The absence of general features and extremely tight design constraints has led to most embedded applications being extremely specialized and highly optimized. A typical embedded design procedure goes through several steps [3],[10]

- Design Specification
- Hardware/ Software Partitioning

- Parallel HW/SW development

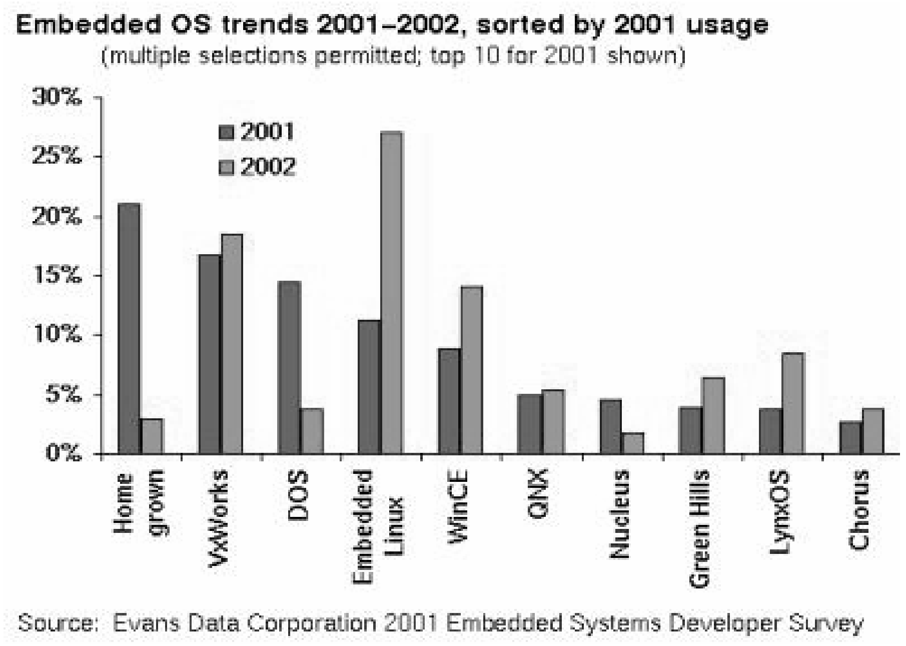
- System Integration and Testing

Each of these steps relies heavily on the experience of the designer. The actual development involves several pain staking steps including writing hand-coded assembly, developing your own specific job scheduler, resource manager and device drivers, addressing code placement and memory space issues and customizing hardware modules for particular design specifications. [3]

All these factors have lead to little scope for errors in the design space. The inflexibility of the design makes the overhead of correcting errors in the design or accounting for modifications in the specifications very high. Additionally it allows for very little reusability of the design or portability. New versions of the product may require complete redesign or in some cases considerable rework for incorporating additional functionality. Further, studies show that more than fifteen percent of developers report that between 26 and 50 percent of their projects are never finished. Forty-one percent report that up to 25 percent of their projects are abandoned. [4]

With more and more players in the embedded devices market a reduced time-to-market is one of the chief concerns. In order to achieve faster time-to-market and prevent abandonment of projects embedded developers are moving towards designing systems which are more generic and flexible. Designers now chose more generic hardware, program in higher level languages and use third party tools for their development.

So what then does the market offer to an embedded system developer? In terms of hardware it is a set of processors which are reminiscent of the desktops of a bygone era. e.g. 68000 and Z80 designs [17, 16] or stripped-down versions of contemporary high-performance desktop processors, e.g. MIPS, PowerPC and x86 designs. [12,14,15]. By definition, these designs are born of high-performance design goals, not embedded-systems design goals. Similarly, many hardware structures such as caches and memory-management units that appear in embedded processors—even in those processors that are designed specifically for the embedded market—are based on high-performance designs, for example the MMUs and caches in the ARM and Hitachi SH7750 architectures [11,13].Some of the embedded microprocessors have incorporated embedded design requirements like low power and reduced die area. e.g. the StrongARM, and Motorolas's Mcore. Several processor vendors have also come up with strategies to improve code density. These strategies are based on compressing 32 bit instructions to 16 bit opcodes and decompressing it on the fly e.g. ARM Thumb extension, using variable length instruction length e.g. Infineon Camel [18] or using fixed 16-bit instructions e.g. Hitachi Super H architecture.



Embedded software developers also have a growing number of third party tools at their disposal. These tools help embedded engineers to work at higher levels of abstractions on designs and specifications, reduce the number of design errors and reduce development time by allowing reusability. They include:

- Design automation tools for system-on-chip designs and hardware - software co-design e.g. Synopsys' System Studio, ARM® Integrator/CP (TM) development platform
- software modelling tools e.g. Artisan,
- integrated development environments e.g. Code Warrior, Multi 2000,.

- Operating Systems e.g. vxWorks, embedded Linux

Their growing popularity is reflected by several strong market indicators. The EDA embedded design market was a 1.5 billion dollar market in the year 2000. Also a recent study about operating system usage in the embedded industry indicates that “home grown” operating systems are soon to be a thing of the past. The percentage of developers using their own operating system is expected to drop by nearly 80% from 25% in the year 2001 to 5% the following year. This rapid downturn in the roll your own category is the direct consequence of the growing complexity of embedded applications and the unwillingness of companies to invest precious man hours on reinventing the wheel, i.e. developing their own operating system services when they can get it from a reliable third party.

1.3 Motivation

Because of the architecture-level focus on high performance computing, many of the microarchitecture structures in use today—even those used in architectures aimed specifically at embedded systems—are geared toward the goals of high-performance computing, not embedded computing. As a result, there is very little architectural support in today’s embedded processors for predictable performance or high resolution timing.

In addition to the previously mentioned legacy mechanisms at the architecture level that frustrate the design of embedded systems (mechanisms such as branch prediction, data prediction, hardware-managed caches, out-of-order issue, etc.), there are numerous legacy mechanisms at the system-software level that cause similar problems. For instance, the time management scheme found in most operating systems uses a periodic clock interrupt to increment an internal counter and thereby update the operating system's internal notion of time in the external world. This scheme is simple and effective and is generally used to handle clock resolutions on the order of ten milliseconds; the scheme does not scale particularly well beyond that, though interpolation schemes exist to increase the resolution further, as discussed later. Nonetheless, to achieve resolutions down to microsecond accuracy is non-trivial using conventional means, and yet these are exactly the types of resolutions necessary for high-performance embedded systems [19,20].

Real-time literature is rich with innovative scheduling schemes targeted towards achieving the timing requirements of the workload in an optimal fashion.[21,22] A larger number of these schemes including EDF are rarely implemented in commercial real time operating systems owing to the substantial and variable software overhead.

Our solution to the problem is to support embedded systems by looking at architectural mechanisms and combined hardware/software schemes that address the goals of embedded systems. In particular, we present an investigation of the following instances of hardware support for embedded operating systems:

special CAM-based hardware[1] to help turn scheduling tasks that typically run in O(n) time into tasks that run in O(1) time; and

a medium-resolution internal clock (500 µsec.) that provides both good accuracy and low overhead, especially if coupled with the CAM hardware.

We find that, by using these mechanisms, one can achieve better timing accuracy and more predictable performance without requiring more CPU overhead or energy. In fact, the scheme simultaneously reduces both CPU overhead and energy consumption. Our experimental set-up is a highly accurate software model of the Motorola MCORE processor [14]; our simulator runs the same unmodified application and operating system binaries as our test hardware and models energy consumption of the CPU as well [2]. The embedded operating system under study is µC/OS-II, a popular public-domain, open-source RTOS [23]. In our benchmark tests of the CCAM hardware, we saw the maximum jitter in the system decrease, we

1. “CAM” stands for *content-addressable memory*, another term for *fully associative cache*. The hardware data structure is used to search a small set of objects in parallel for a matching value or least/greatest value; it essentially behaves like a small, fast hardware database.

were able to increase the number of tasks in the system by a factor of two beyond that achievable by software means alone, and on an average we saw a factor-of-two reduction in energy consumed.

The gains come at a price, though detailed discussion lies beyond the scope of this paper. The cost is an increase in die size that is significant when compared to the processor core alone but less so when compared to the size of a typical embedded chip that includes RAM, ROM, and various peripheral devices. The CAM structure and associated timing hardware account for roughly 2000 register-bit equivalents (RBEs, a process-independent unit of die-area measurement [27]) for a 64-task CAM. Given that a 32x32 register file requires roughly 1000 RBEs, this can double the size of a small cacheless embedded core such as MCORE. However, once peripherals and memory structures are added, the overhead is less dramatic; the 64-task CAM structures represent more moderate 5% increases in the size of a typical embedded processor, such as the MCORE-based MCM2001 [15], that includes external RAM, ROM, and peripheral I/O devices.

1.3.1 Support for Process, Time and Event Management

There are numerous facilities one could investigate to help the performance of embedded systems. We chose to attack the facility that is common to all embedded operating systems and represents a variable overhead in each one (i.e. its overhead scales with the number of tasks or processes in the system). This facility is the management of processes and events, or scheduling.

The design of the mechanism that we investigate is based on the needs of the operating system, which must perform the following functions:

determine, from a set of tasks, which has the highest priority or nearest deadline;

determine which in a set of tasks need to be updated when a semaphore or lock is released;

provide to tasks a mechanism to suspend themselves for a specified period of time or until a specified event occurs.

Scheduling is intrinsically an O(log n) or O(n) operation, as it involves ordering and searching a set of n tasks, based on priority, earliest deadline, etc. Scheduling represents an overhead that is variable, depending on the size of the task set, and is thus a good target for optimization. The scheduling process as implemented in most embedded operating systems is actually O(n) because the task sets are typically small (rarely more than a few dozen tasks) and maintained in short lists or arrays—not binary trees. The constants involved in maintaining a balanced binary tree would tend to make the O(log n) implementation actually take more time than the simpler O(n) linked-list implementation.

We note that the embedded operating system’s requirements (e.g. search a small set of items for a matching value or a greatest/least value) map extremely well to a CAM structure. By implementing these functions in hardware, one can take advantage of hardware's ability to perform multiple comparisons in parallel and so turn this O(n) operation into an O(1) operation; i.e. the procedure takes the same amount of time no matter how large the task set, up to the size of the CAM structure.

1.4 Overview

This report is based on the following lines. Chapter 2 will provide a background on the working of real-time operating functions. Chapter 3 will be an overview of the CCAM- a description of its interface, its operation and design. Chapter 4 will describe the experimental setup while in Chapter 5 the results of the experiments will be presented. Chapter 6 will provide a summary of the work and give an insight into future work.

CHAPTER 2

BACKGROUND: REAL-TIME OPERATING SYSTEMS

2.1 Real-Time Systems

Donald Gilles' definition of a real-time system[28] is as follows:

"A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."

This implies not just that real-time systems have to respond to external stimuli or service tasks within a specified timberland but that, irrespective of the system load, this action has to be predictable. Additionally it is also desirable that real-time systems achieve their functional correctness and timeliness while being highly utilized.

A good example of a real-time system is robot arm working on a conveyor belt. The robot arm has to pick a certain part off the belt. In this case the robot arm will not be able to pick up the part if it attempts to do so either late or early. Another more complex example is an air traffic controllers. These systems are required to monitor the movement of aircraft and changes in the weather conditions in their coverage area and based on this information they have to be able to calculate the trajectory of aircraft in the system, schedule its landing and provide all this information to a human controller. These systems have to perform a wide variety tasks including data and image processing, updating data bases and intensive computation.Each of these tasks have different response times e.g. weather updates may take place every 2 seconds while display refreshes have to occur every tenth of a second. Additionally the system has a variety of workloads some of which are periodic, like weather updates or display refreshes and others which are aperiodic e.g. aircraft entry into airspace. The system response to each of these tasks has a well defined upper bound which has to be achieved irrespective of the system load i.e. changing weather conditions or number of aircraft.

2.1.1 Real-Time Terminology

There are several terms which are commonly used in discussion of real-time systems. This section is an overview of some of these terms.

Job : This refers to a unit of work which can be scheduled or executed by the system. For example a function which reads the input off a memory port can be considered a job.

Task : This is a collection of related jobs which collectively achieve a single function. For example a collection of jobs which read data from a memory location, perform certain computations and write the result to another location can all be considered to constitute a task. A task is said to be periodic if it has to be run at regular intervals. An aperiodic or sporadic task is run when a certain infrequent event occurs.

Release Time : This is the earliest time at which a task becomes available for execution. A job can be executed any time at or after its release time. If all the jobs in a system are available for execution right at the outset than it is assumed that no job has a release time. For example a periodic job may be released every 100 ms for execution. The release times for this job is 100 ms, 200 ms, 300 ms and so on and so forth.

Deadline : This refers to the instant of time within which the job has to complete execution. If the periodic job mentioned above has to complete prior to the release of the next job than its deadlines are 200 ms, 300 ms, 400 ms etc.

Response Time : This refers to the time interval between the release time of a task and its completion time i.e. the instant it completes execution. For example if a job is released at 10 ms and completes execution at 25 ms, than the response time is 15 ms.

Tardiness : This refers to how late a task completes with respect to its deadline. A task is said to have a tardiness of zero if it completes at or before its deadline while a task which completes after its deadline has a tardiness equal to the difference between its completion time and its deadline.

2.1.2 Classification of real-time systems

Real-time systems are classified based on the strictness of their timing requirements or the criticality of their deadlines. There are two categories

Hard Real-Time Systems : A real-time system is said to be a hard real-time system when it has extremely tight timing constraints and strict deadlines. These systems execute critical tasks whose delayed execution or non execution would have catastrophic consequences. Hard real-time tasks are expected to have a tardiness of zero and exhibit deterministic temporal behaviour.This implies that they are expected to complete at or before their deadlines and that their execution is absolutely guaranteed. Typical hard real-time applications include anti-lock braking systems, pace makers and air flight controllers.

Soft Real-Time Systems : On the other hand soft real-time systems are more concerned with best-effort services. [6,7] These systems execute less critical tasks and can tolerate missed deadlines. Unlike in a hard real-time system where results from late tasks have no usefulness, tasks in a soft real-time system continue to be useful even if late. The usefulness of their results gradually tapers off with the tardiness of the task rather than dropping abruptly to zero at the instant after the deadline. In general the performance of these systems is described in probabilistic terms i.e. 95% of the time the data arrives on-time.

Typical examples include multimedia applications like MPEG decoders and encoder, telecommunications applications like internet telephony and some sorts of data acquisition applications.

2.1.3 Real-time system requirements

A real-time system has several requirements to fulfill including:[28, 25]:

Timeliness: There is a certain upper bound within which the system is expected to finish certain tasks. The time within which the task has to be completed is termed the deadline. The criticality of the deadline depends on the function of the system.

Simultaneity or simultaneous processing: The system should be able to process events that occur simultaneously and still meet all deadlines. This implies that a real-time system should be inherently parallel in nature. This can be achieved by making the system either a multi-processor system or by using a multi-tasking model.A distributed real-time system or a heavily interrupt driven system are cases where simultaneous events occur on a regular basis.

Predictability: One common goal in real-time system design is predictability. Predictability implies that it can be demonstrated that the system fulfills its requirements under a variety of situations. This implies that predictability is subject to the assumptions made while determining it[29].

Predictability of simple static real-time systems is much easier to determine and design for. This is because the number of tasks in these systems are known and their worst case execution times are available up front. In addition the system consist of largely either hard real-time or soft real-time applications. The predictability of these systems then can be expressed using a single number depending on which component i.e. interrupt latency or fixed periodic task behavior, is the dominant factor.

Predictability of a complex real-time system on the other hand is more difficult to design for. This is because these system have a variety of tasks with different levels of criticality. The environment in which these systems function is also non-deterministic in nature. In general predictability for these systems can be viewed to have two components.

The first which is a more macroscopic is concerned with overall system performance. This component requires demonstration that critical tasks performance is 100% guaranteed and that non-critical tasks achieve a performance which is close to the maximum actual performance. The other is with respect to individual tasks or task groups in the system. The second more microscopic look at the system considers the specific periodicity timing and deadline requirements of the individual tasks.

Dependability: As real-time systems are often deployed on extremely critical missions, they have to be dependable. Dependability is defined as “the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers.” [30]. A dependable system has the following attributes

- Availability This is the extent to which a system is ready for use
- Reliability This is a measure of the extent to which the system’s actions and results can be trusted.
- Safety This is the extent to which the system does not harm the surrounding environment itself.
- Security This is the extent to which the system is safe from unauthorized access or tampering.

A dependable system has to provide for the following [5]

- Fault Prevention: Prevention of fault occurrence,
- Fault Tolerance: Correct functioning in the event of faults
- Fault Removal: Elimination of current faults
- Fault Prevention: Forecast future faults and prepare for their prevention or effective handling.

To make a real-time system dependable it would require providing for exception handling, incorporating fault tolerant features into the scheduler and designing for correct functioning in the event of partial hardware failure.

2.2 Real-Time Operating Systems

The Posix Standard 1003.1 defines real-time in operating systems as:

"Real-time in operating systems: the ability of the operating system to provide a required level of service in a bounded response time."

This implies that a real-time operating system aka the RTOS is expected to provide basic operating system functionality like scheduling, timer services, synchronization primitives, inter-process communication mechanisms among other things in a deterministic fashion. This implies that the overhead of the operating system should be consistent for all invocations.

RTOSes are expected to have a small size as they are used on systems with scarce memory resources. Typical RTOS memory footprints are on the order of several 100 kilobytes. As the subset of operating system services used is heavily application dependent these systems have to be modular, scalable and completely configurable.Besides the basic real-time kernel services commercial RTOSes offer a variety of optional services including support for networking, file system I/O, multiprocessor systems, graphics etc. The extent of the extensibility of modern day RTOS' can be gauged by looking at the flexibility offered by the commercial RTOS, vxWorks. vxWorks offers nearly 100 different options to the real-time programmer which can be used to generate a variety of configurations. [25]

In this chapter we examine some of the basic real-time kernel services, design choices and implementations.

2.2.1 Scheduling

Scheduling is the mechanism which determines which job has to be executed from the pool of jobs in the system. The logistics behind the decision making mechanism is based on the scheduling algorithm implemented.

2.2.1.1Terminology

Typical scheduling algorithms used in RTOSes are termed correct because they generate valid schedules

Valid Schedule: Jane Liu [25]says that a valid schedule has to guarantee that all tasks are scheduled only after or when they are released and such that all resource and precedence constraints are met.

Feasible Schedule : A valid schedule in which every task meets its deadlines.

Optimal Scheduler : A scheduler which always generates a feasible schedule if the given set of jobs can have one.

Preemption : This refers to the ability of the scheduler to halt execution of a task in favour of another task with higher priority or criticality. Permitting preemption allows the scheduler to service sporadic tasks and higher priority tasks which are released during the execution of the current task. A system where a task can continue executing without being ever switched out is a nonpreemptive one.

Schedulable Utilization : This represents the upper bound of the utilization of the periodic tasks in the system for which a particular algorithm can generate a feasible schedule. Clearly the higher the schedulable utilization the better the algorithm is.

Ready Queue: It is the set of all tasks in the system which have been released are available for execution.

Pause Queue /Timeout Queue/ Delay Queue: It is the set of tasks which are not available to run till the passage of certain time i.e. the delay of the task

2.2.1.2Types of Schedulers

Scheduling is one of the most heavily researched areas in the real-time world. The result is a plethora of schedulers which tackle a wide range of application types and offer varying services and utilizations. The three most commonly used approaches to scheduling are

Clock-driven : This is an off-line scheduling mechanism common in systems where the nature of the workload i.e. execution times, relative deadlines etc. are known exactly at the point of system design. Jobs in systems employing a clock-driven scheduler are executed in a predetermined manner. The run time overhead of scheduling in these systems is negligible.

Weighted round robin: This is the commonly used approach in systems with time-shared applications. Tasks which are ready to run are placed in a FIFO queue. Starting from the head of the queue each task is executed in turn for a single time-slice. If the job does not complete at the end of its time slice it is preempted and placed at the end of the queue.

This technique is used in networking applications where messages have to be sent or received from several ports. The advantage of this scheme is that the overhead of inserting a task on the queue or removing it off the queue is constant. The downside is that the response time of a task is proportionate to both number of tasks in the queue and the number of time slices it takes to complete.

Priority driven approaches : Schedulers of this type base their scheduing decision based on some sort of priority. In these systems each task is associated with a priority. This priority is based on either off-line computations or on-line run-time calculations. Priority schedulers always schedule the task with the highest priority.

Thus there are two types of priority driven schedulers

Fixed or Static Priority Schedulers: The priority of tasks in these systems is fixed at the outset using standard off-line methods like Rate-Monotonic Analysis [21] and Deadline -monotonic analysis. RMA assigns tasks priorities based on their periods. The shorter the period, the higher the priority of the task. These priorities are fixed for the lifetime of the system.

The characteristics of a fixed-priority scheduler are:

- Good predictability: Systems using a fixed priority scheduler can be validated easily using static off-line methods.Additionally in an overloaded system or in the event of job over runs one can easily predict that higher priority jobs will be serviced and the lower priority jobs will miss their deadlines.
- Schedulable Utilization: It can be mathematically proven that the schedulable utilization of a fixed-priority scheduler is not very high. The schedulable utilization of RMA degrades exponentially from the best case of 82% for a system with two tasks to nearly 69% to a system with 10 or more tasks.
- Simplicity: As tasks are associated with the same priority level through out their life this makes for easier system design, implementation and validation.

Fixed priority schedulers are the common choice in most standard real-time systems. Real-time systems support anywhere from 64 different priority levels to 256 different priority levels.Some operating systems allow multiple tasks to share the same priority level while others allow only one task to be at a certain priority level.

In order to support fixed priority scheduling in a system with unique priority levels one of the approaches is to maintain a ready queue which is sorted based on priority. Selecting the next task to run involves just looking at the first element in the queue and hence has a constant low overhead. Inserting tasks onto the list has a variable overhead and is dependent on the number of priority levels and tasks in the system. Another approach would be to maintain an unsorted ready queue. Every scheduling instance the scheduler traverses the entire queue and determines which task to run in the next interval. This mechanism is common in general-purpose operating systems like Linux where the operating system traverses the entire queue, calculates the current goodness of the task and uses this value when it makes it selection. The scheduling operation in this case has a constant overhead. But this determinism is achieved at very high costs.

All the mechanisms described above have overheads which scale with the number of priority levels. This is not desirable in an embedded system where it is desired that the RTOS overhead is not just low but also does deterministic and ideally independent on the workload i.e. its nature, size etc.

One of the methods used to reduce the overhead and still achieve determinism is to employ bit vectors. This mechanism is employed in μC/OS.

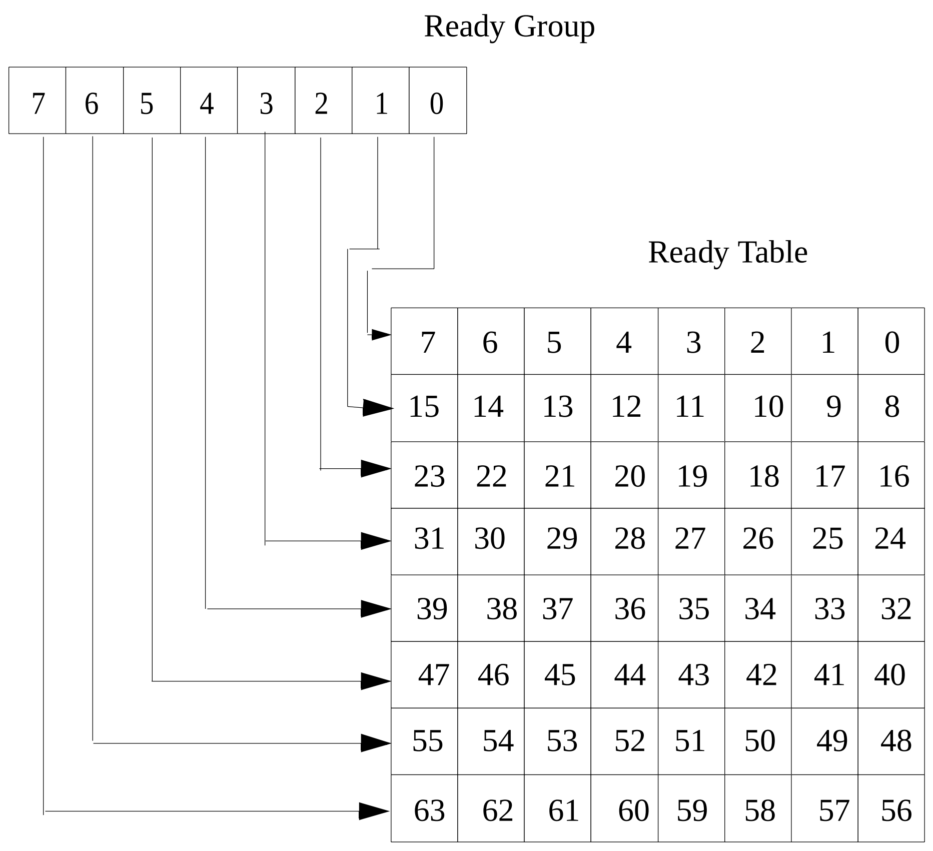


Fig. 2.1. Ready List in µC/OS. The figure shows the ready list in µC/OS which is divided into a two level data structure consisting of a ready group and the ready table. Individual tasks have an entry in the ready table and a group of tasks have a single bit entry in the ready group.

This is achieved by first partitioning the tasks into groups. This grouping is based on the priority levels of the individual tasks. The ready list of µC/OS is shown in Figure 2.1. For e.g. µC/OS has 64 tasks which are partitioned into 8 unique groups where tasks in group 1 will have a higher priority than those in group 2 but a lower priority than those in group 0. Each group is associated with a bit vector which is set when any task in the group is ready to run. In μC/OS a single byte can be used to hold the group bit vectors. Each group in turn is associated with a byte of bit vectors which are associated with individual group members. Entering a task on the ready list or the two level bit vector structure involves setting the bit associated with the task's group and the bit associated with the task in its groups bit vector table. Scheduling in μC/OS essentially involves using this two variable ready list to determine which task to run. It does this by performing a 2 step inverse lookup operation. Thus the overhead of inserting or removing a task from the ready list or selecting the next task to run is constant. But the trade-off is the increased kernel size required for maintaining an inverse lookup table of 256 entries.This particular implementation of the selection procedure is not very scalable as the size of the inverse lookup table increases exponentially with the number of priority levels in the system.

Alternate implementations with bit vectors do away with the bulky inverse lookup table. This is done by associating a unique bit vector to every priority level in the system and performing a number of comparisons in order to determine the highest priority level job available. The number of such comparisons depends on the number of priority levels in the system, the size of the data in the system and the actual profile of tasks which are available at a given instant. Though there is a factor of variability in scheduling the worst and best execution times are not so vastly different to be an issue as in the earlier cases.

Dynamic Priority Scheduler: These schedulers assign different priorities to different jobs in a task. The priority of a task varies dynamically with respect to the other tasks in the system.Some dynamic priority schedulers include Earliest Deadline First - EDF and Least Slack Time First - LST.

EDF assigns priorities to jobs based on their absolute deadlines. Once a job is placed on the ready queue its order with respect to the other tasks on the queue is fixed. Thus the priority of a task can vary with each run making it a dynamic scheduler.

Some characteristics of EDF schedulers are

- Schedulable Utilization The EDF algorithm has a 100% schedulable utilization. This makes it an optimal algorithm for all loads.
- Predictability The disadvantage of this algorithm is that it produces unpredictable schedules in overloaded scenarios. Late jobs which have missed their deadlines have higher priority than later jobs which have not missed their deadlines in turn to be late. This implies that mechanisms to handle system overload and job overruns have to be implemented.

- Validation This system cannot be validated due its unpredicatability using known off-line methods. It requires dynamic validation which may not be feasible.

Echidna, a commercial RTOS, implements a non preemptive version of the EDF algorithm. The ready queue in Echidna comprises of tasks which are sorted on the basis of their deadline. The task at the top of the queue has the earliest deadline and hence the highest priority. Selecting a task to run has a constant overhead but inserting a task onto the queue is an O(n) operation. For preemptive system the overhead of placing tasks on the queue would be very significant.

An alternate way to implement this would be to maintain FIFO queues of threads with the same relative deadline [25]. The tasks in each queue are ordered on the basis of their absolute deadlines. Thus task selection involves just searching the heads of each of these queues. The overhead of this operation is dependent on the number of relative deadlines in the system. The complexity can be decreased from O(n) to O(log n) by using a priority queue for the first task of each FIFO queue. Inserting a task would involves adding the task at the end of the queue associated with its relative deadline. The complexity of this operation is 1 when the FIFO queue is non-empty and is O(n) when the queue is empty.[DIAGRAM]

As can be seen EDF schedulers are not easily implemented in software with low overheads due to their very dynamic property. This and their unpredictable behavior have not made them popular choices in the real-time operating system world.

2.2.2 Time Services

The concept of time is central to a real-time operating system. Both task level and system level decisions are made on the basis of time. Time management can be said to have two aspects. The first aspect is concerned with the actual representation of the current time, its update and retrieval while the second aspect is concerned with the usage of this ‘time’ itself or its passage to make scheduling decisions, provide timestamps, exact delays, time-outs and periodic signals.

To keep the operating system apprised of the current time in the external world, typical systems use a high priority interrupt driven by an accurate clock at 100-1000 times per second. Most operating systems maintain a counter which gets updated every timer interrupt. The value of this counter, the timer tick counter, represents the number of timer ticks or interrupts that the system has seen since startup. Operating systems like µC/OS use this counter as the system clock. These operating systems offer a *current-time* function that returns the time as the number of ticks since the system started. The granularity of this measurement is dependent on the frequency of the timer interrupt, and the user has to perform the conversion from ticks to the desired time units.

Other systems like Linux maintain the system time relative to the Epoch Jan. 1 1970 in terms of seconds and microseconds as opposed to timer ticks. In these cases, updates to the system clock involves adding a timer tick interval in seconds and microseconds to the system clock and handling possible microsecond overflows. As the overhead of this update is far greater than the simple counter increment approach, the timer interrupt handler is treated as a long interrupt. A typical ‘long interrupt handler’ in Linux splits the workload into work that needs to be done immediately and work that can be done a little later. The interrupt handler handles only the former and schedules the latter in what is termed as the bottom half of the interrupt. Thus, the timer interrupt handler in Linux only updates the timer tick counter. The system clock update takes place in the bottom half of the interrupt. To handle possible infrequent updates the system clock, Linux maintains an additional counter, the wall clock tick counter, which keeps track of the last update to the system clock in terms of timer ticks [26].

When asked for the *current-time*, Linux returns the value of the software system clock after accounting for the time that has elapsed since it was last updated. The elapsed time is comprised of two components. The first component takes care of possible infrequent updates to the system clock. This is done by accounting for differences between the timer tick counter and the wall clock tick counter. The second component accounts for the time since the last timer interrupt. Computing this involves using counters that are updated independently by the hardware, like the timer modules count-down register. The timer module’s countdown register value gives the amount of time remaining until the next timer interrupt. The operating system converts this value appropriately to compute the time elapsed since the last interrupt. In architectures with on-chip counters, like the Time Stamp Counter in Intel’s microprocessors, Linux estimates the time since the last interrupt occurred by using the difference between the TSCs current value and its value at the last interrupt.

There are problems with each of these schemes. The simple interrupt-driven scheme is not scalable, as more frequent interrupts would simply increase the operating system's overhead and increase the chance that a high-priority clock interrupt would delay the execution of an otherwise high-priority task. Using the TSC counter or the timer counter registers in order to achieve finer granularities complicates the process of reading the system clock. The complexity of the operation can result in a substantial overhead that can effect the accuracy of the readings. This is the case when the overhead involved in obtaining the higher resolution measurement is of the same order as the precision of the measurement itself. On the MCORE processor the Linux like scheme takes approximately 8 microseconds to complete assuming that there are no overheads involved in accessing the timer counter registers. Thus in the best case it is within 6% of the precision of the timer measurement. The accuracy of a time measurement that uses interpolation techniques is considered to be less accurate than one which is based exclusively on interrupt based clocks [Kailas2000]. In addition, this process of interpolation is extremely platform dependent and is not easily portable across systems.

Often a task in a real-time system has to pause its execution for a finite amount of time. This time is termed the timeout or delay.This delay value is typically expressed in the terms of system clock units. All operating systems provide facilities for tasks to place themselves on the timeout queue or delay queue or pause queue. Every timer interrupt, besides updating the system clock, the RTOS updates the state of tasks on this timeout queue. If the timeout value associated with the task has expired it is released and placed on the ready queue. By associating the timer interrupt with the timeout queue update, the granularity of time delays automatically gets restricted to that of the system clock. Typical embedded applications like those in networking and controls require granularities far finer than that of the system clock timer interrupt which is typically 10 ms. Finer resolutions would imply an increase in the RTOS overhead.

One of the solutions to circumvent this problem has been to use hardware based timers to schedule individual tasks which require timing resolutions higher than those provided by the system clock.

Timer implementation is a POSIX 1004.3 standard.Several OSes offer this facility including Real - Time MACH [31], real-time extensions to Win NT, vxWorks[25] etc. This facility includes providing the task the capability to create individual timers. Each of these timers is associated with a period or one-shot timeout and an expiration action. Upon the creation of a timer the OS would program one of the multiple clocks (hardware counters which countdown and release an interrupt when they hit zero) based on the information provided by the task. When the timer throws an interrupt the associated interrupt handler takes the appropriate expiration action which normally involves releasing a task. Some RTOSes extend this single user task - single timer concept by associating multiple tasks with a timer. The timer is associated with a software based queue of tasks sorted on the basis of expiration time. This timer timeout queue is rather similar to the system level timeout queue. This timer mechanism is efficient when the number of tasks using it is low and the number of interrupts do not completely overwhelm the system.

One of the chief issues associated with any form of timeout queue is the overhead of maintaining it. Typically the timeout queue is maintained as some form of linked list structure. RTOSes like Echidna, NOS and real-time versions of Linux maintain the timeout queue as a queue of tasks sorted on the basis of their timeouts. The timeout of any task on the queue is relative to that of the task immediately before it on the queue. Insertion onto the queue has an overhead of O(n) while update is O(1) because only the timeout of the first element has to be updated. The complexity of removing tasks from the queue i.e. releasing a task after its timeout has expired is also an O(1) operation. But at an given update event, more than one task may become ready to run. Thus the total overhead associated with releasing tasks from the timeout queue is dependent on the number of tasks in the system. This chief advantage of this method is the low overhead associated with the update. The chief disadvantage of this scheme is the non-deterministic overhead.

μC/OS tackles the problem of non-determinism associated with inserting a task on the timeout queue by eliminating the need to perform this operation completely. All the tasks in the system have an entry in the timeout queue by default but only some tasks have delays associated with them. Timeout queue updates require traversal of the entire queue and updating the status of each individual entry. Though this operation has a high overhead, it is constant in a system with a fixed number of tasks.

2.2.3 Inter process Communication and Synchronization

Most operating systems provide a variety of facilities for two processes to exchange data, control information and synchronize this information exchange in order to ensure that it occurs at the right time and such that the two processes do not interfere with each other. Typical operating system methods for exchanging information include shared memory and message queues. Typical OS synchronization primitives include semaphores, mutexes and monitors.

As virtual memory is not a common feature among RTOSes, most uniprocessor RTOSes do not support shared memory. Shared memory in these systems is normally implemented in these systems as global data. In order to synchronize access to this global data synchronization primitives like semaphores and mutexes can be used. Sometimes the need for these synchronization mechanisms is eliminated by intelligently scheduling the processes accessing this data.

Message queues provide a means for one or many threads to communicate with some other thread or threads. Message queues can be implemented as FIFO or LIFO queues when all messages have the same priority or as a list of messages sorted on the basis of priority. A mailbox is a message queue which can hold only one message at a given point.

Semaphores and mutexes are common synchronization primitives implemented in RTOSes. Each semaphore is associated with a count which keeps track of the number of tasks which can access some shared data at a given point.

The following are operations common across all IPC mechanisms.

When a task makes a request for a semaphore or a message on the message queue one of two things can happen. If the resource is available the task is granted its request right away. If it is not available the operating system places the task on the blocked list of tasks waiting for the particular resource.

Similarly when a task posts a semaphore or a message there are two possible events that can occur. If there is no task waiting for the resource than the operating system notes that an additional resource has been made available. On the other hand if there are tasks waiting on that resource than the OS selects a task from the list of waiting tasks to release. This selection policy can be either based on the order in which requests for the resource were made i.e. FIFO or based on priority i.e. the task with the highest priority is released every time or in a random fashion.

μC/OS offers various inter-process communication mechanisms including semaphores, message queues etc. Each mechanism is classified as a type of event and is associated with a data structure called the event block. Event blocks contain information like the nature of the event (semaphore etc.), an event counter, and a list of tasks blocked on the event (Blocked task list). This blocked list is similar to the RTOS' ready list. Blocking a task on a particular event involves updating the event's blocked list as well as removing it from the RTOS' ready list. Determining which task needs to be released upon the posting of an event involves a procedure similar to that of determining which task needs to be run. The overheads of these operations are constant. The disadvantage of this implementation is that each event is associated with its own blocked list and using an IPC heavy system will require high memory overheads.

Echidna on the other hand provides a synchronization primitive similar to the semaphore called the syncaphore. The blocked list of a syncaphore is a FIFO queue. Inserting a task on the queue has a complexity of O(n) in the current implementation as the OS traverses the entire queue of blocked tasks and adds the new entry to the end of queue. This can be optimized to an O(1) operation by maintaining an additional pointer to the tail of the queue. The task which is released when a syncaphore is posted is always the first task in the blocked queue. Task selection is thus a O(1) operation.

In Echidna the task blocks are moved between the various queues. Thus using many syncaphores does not increase the memory overhead of the operating system.

2.3 Related Work

A software based clock in general is susceptible to variations in the interrupt latency of the processor and the time to execute the timer interrupt handler. Various suggestions have been made to provide more accurate timing [31,32,20]. In particular, Savage [31] suggests using additional high resolution hardware timers. Each timer is associated with a sorted timeout queue onto which a task can add itself. The scheduler of the main operating system (Mach) is multiplexed with the interrupt handler of these hardware timers. The resolution in this scheme is achieved by increasing the interrupt overhead of the system. Kailas [20] focuses on building an on-chip time management unit that provides accurate and precise time readings. In their scheme the system time is updated independently of the software. The scheme's focus is to provide an accurate time stamp for events. Adomat [31] transfers the entire RTOS functionality into hardware in an attempt to get better determinism and performance. They suggest building a real time coprocessor unit to achieve this. The coprocessor is capable of making scheduling decisions, etc. This does not translate into flexibility as far as the RTOS developer goes.

The off chip hardware unit is well suited for multiprocessing real time environments. Lindh [34] discusses incorporating the real time coprocessor into a multiprocessor environment. Klevin [35] also discusses using it for bus monitoring in a multi processor unit. Furuns [33] suggests hardware means to support asynchronous interprocess communication in a message intense system.

CHAPTER 3

CCAM: DESCRIPTION

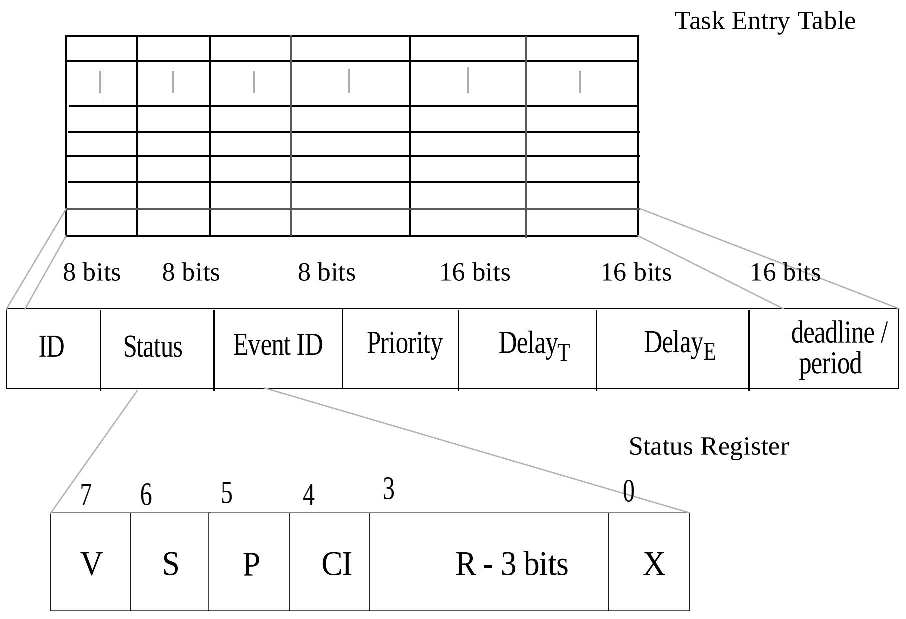
The CCAM is a hardware data structure that maintains the state of all the tasks running on the system. Figure 1 shows the layout of the CCAM Task Entry Table. We use a 64 entry CCAM table for our experiments. Each entry is 72 bits wide and has 6 fields.

3.1 CCAM Structure

Each task in the system is associated with an entry in the CCAM. The following are the fields in the CCAM.

ID : This field gives the address of the task entry in the CCAM table. It serves as the unique identifier for a task and is the value which the hardware uses to tag the result of any requested search.

Status : This 8 bit entry indicates the status of the associated task entry. The structure of the status entry is given in Fig 3.1. All the fields are single bit wide except for the Run Status field which is 3 bits wide.



Task Status Register Bit Descriptors

V - Valid	CI - Cannot Interrupt
S - Suspend	R - Run /Scheduling Status
P - Periodic Task	X - Unused

Fig. 3.1. CCAM Structure. The structure of the CCAM's Task Entry Table is shown. Each individual task entry comprises of several fields which contain pertinent information for a task. The status, event id and priority are all 8 bit fields while the remaining fields are all 16 bits long.

Valid: This bit is used to indicate if the entry in the CCAM table has valid data or not.

Suspend : Setting this bit indicates that the task has been suspended.

Periodic : When this bit is set it indicates that the task is periodic in nature. The CCAM provides support for periodic tasks whose deadlines are the release times for the next iteration of the task.

Cannot Interrupt : Tasks with this bit set cannot generate CCAM interrupts when they become available to run.

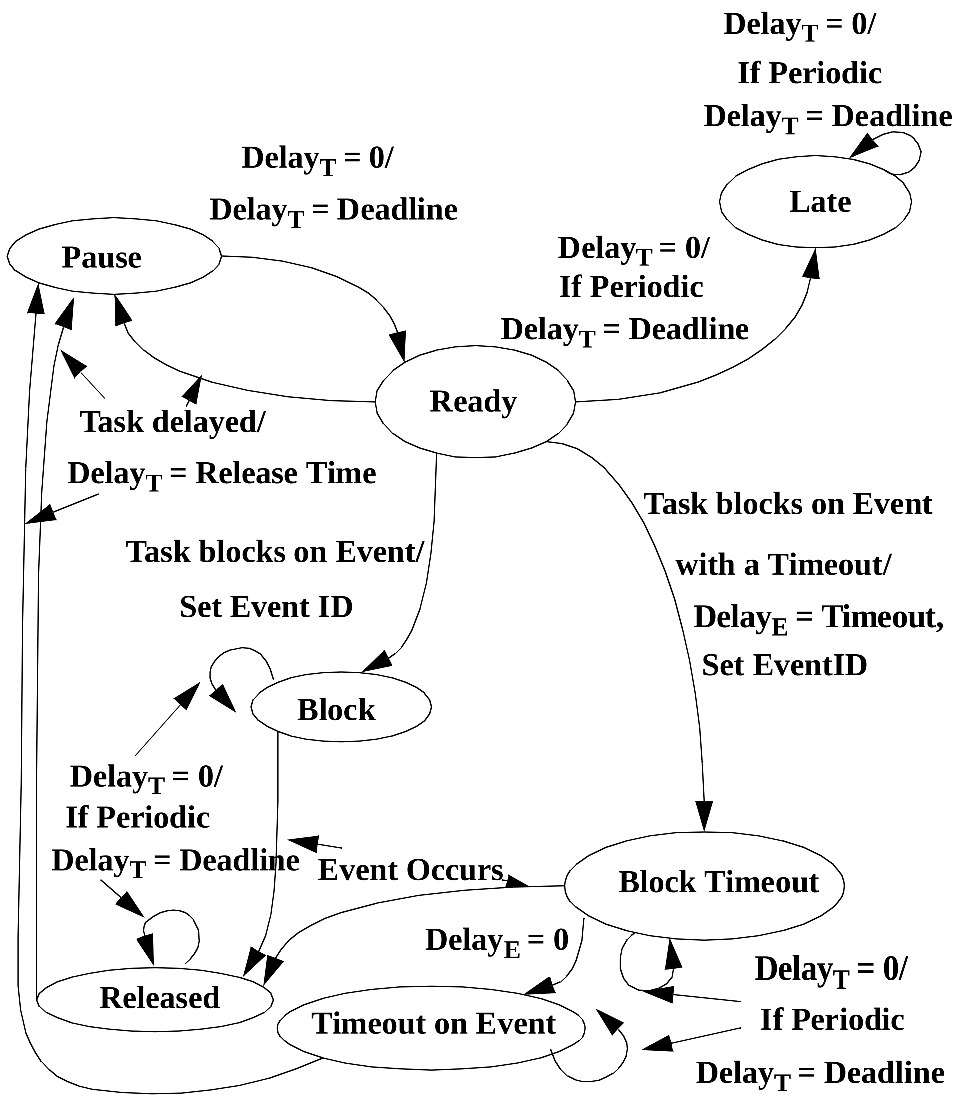


Fig. 3.2. State Machine representation of Run Status of a Task. The diagram shows how the run status of a task changes in the course of its execution. Each task is associated with a release time and a deadline. In the case of a periodic task the deadline is the period of the task. $Delay_T$, $Delay_E$ are the two types of delays possible for tasks. They are updated every timer tick. The former is mapped onto the Delay - Timeout field and the latter to the Delay - Event field.

Run Status : This 3 bit field is used to represent the scheduling status of the task.Tasks have a time related or event related status. Tasks may have any of the following time related status,

- Pause i.e. waiting to be released,
- Ready i.e. available for execution or
- Late i.e. has missed its deadline.

In addition a task may have event related status like

- Blocked or waiting on an event,
- Blocked with a timeout i.e. waiting on an event for a finite amount of time,
- Released i.e. becoming available to run after the event is posted and
- Released after timing out on an event.

Fig3.2 shows how the run status of the task changes during the course of its execution.

These states are equivalent to the various queues maintained by the OS to help manage tasks. For example a task which is available for execution is associated with a ready status which is the same as being placed on a ready queue. Tasks which are waiting on an event are placed on the blocked queue associated with the particular event. The various event queues are

Table 1: CCAM Module Address Map

Address	Name	Use	Access Type
100010e0	ID Register	Task ID on which operation needs to be performed; Holds results of a CCAM Instruction	Read / Write
100010e4	Instruction Register	CCAM Instruction and required data are written into it	Write
100010e8	CCAM Tick Data Register	Duration of single tick in microseconds	Write
100010ea	CCAM Tick Register	Number of system ticks which have occurred since initialization	Read/Write
100010ee	Control Register	Used for configuring the CCAM operation mode	Read/Write

integrated on the CCAM. All blocked tasks are given the status of blocked or blocked with a timeout. The event ID is used to help determine which event queue a task is on.

Priority : This 8-bit field holds the priority of a task as per operating system convention. This value is used to by the Get Task- Priority Sort instruction and also to break ties when an event on which two or more tasks are waiting on is released.

Event ID : This field holds the ID of the event on which a task is waiting. It helps distinguish between all blocked tasks in the system.

Delay Fields : There are totally three 16-bit fields in a CCAM entry which are dedicated to handling time related constraints.

- **Delay$_{\mathbf{T}}$** This 16 bit field holds either an unsigned or signed value depending on the CCAM configuration. Negative delay values indicate that the timeout value has expired. The delay values are updated every time the timeout queue is updated. In the case of preemptive systems this is every timer tick while for a non preemptive systems this would be when a task has relinquished the processor and a timeout queue update needs to be performed or every timer tick if the system is idling.

  The interpretation of the value in this field depends on the run status of the task. When the task in paused the value in this field represents the release time of the task. Once the task is ready to run, the value in the field would represent the deadline for the task. When the CCAM is configured to support negative delays, the delay field will continue to be updated once the deadline has passed and the task’s status has changed to late. The delay value then indicates how late the task is.

  In the case of periodic tasks the value in this field represents the release time for the next iteration of the task. If the periodic task is ready it also represents the relative deadline of the current iteration of the task.

- **Delay$_E$** This value is used to track event timeout values. When a task blocks on an event it can specify the maximum amount of time in timer ticks that it is willing to wait for the event. The CCAM puts this value in this delay field.
- **Periodic Delay / Deadline** This 16 bit location serves as a storage location. When an EDF scheduler is used it is used to store the value of the deadline of the task. When the task is released i.e the delay$_T$ field reaches zero, it is this value which gets automatically loaded into the delay$_T$ field. For a periodic task this value represents the period of the task.

3.2 CCAM Module Interface

The CCAM module interface has 5 registers. These registers are mapped into the memory space of the system and can be accessed using load and store instructions. The memory module map of the CCAM interface is given in Table1.

CCAM Control Register: This 8 bit register is used to configure the operating mode of the CCAM.

The bits have the following meaning

Auto Decrement AD When this bit is set the CCAM automatically updates the state of the delays every “timer tick” i.e. performs automatic timeout queue maintenance. The RTOS sets the rate of the timer tick at initialization.

Enable Interrupt EI When this bit is set along with the AD bit the CCAM generates interrupts every time it performs a timeout queue update and finds that a task has become available for execution i.e. status of any of the tasks in the CCAM table has changed from Pause to Ready or from Blocked Timeout to Released Timeout following an update.

Enable Repetitive Interrupt ERI When only the EI bit is set the CCAM interrupt is a one-time interrupt. Enabling this bit makes the CCAM interrupt a repetitive one.

Lock L When this bit is set the CCAM cannot perform any automatic updates. The RTOS sets this bit every time it wants to access the CCAM and the AD mode is enabled.

Negative Delay ND This bit determines if the CCAM delay is a signed or unsigned number. When set delay values of tasks, which are late or have been released following timeout on an event, will continue to be decremented during timeout queue updates.

Release Time - Deadline RD When this bit is set it implies that the RTOS uses both release time and deadline information to perform its scheduling.

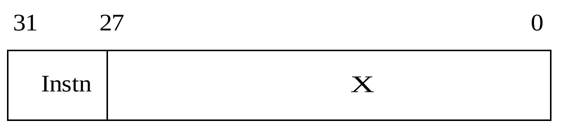
This mode is useful in deploying dynamic priority based RTOSes onto the CCAM.

At startup all bits in the control register are disabled. The CCAM operating mode is normally set up during system initialization. The configuration choice is made based on whether the system is preemptive or nonpreemptive, how many tasks it is running, the nature of its scheduling algorithm etc. For example an operating system like Echidna would set the RD bit to indicate that tasks have a release time and a deadline.

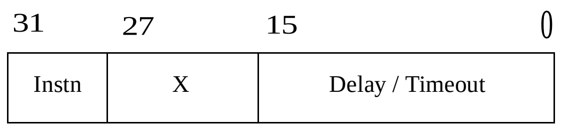
The operating system uses two registers to access the information in the CCAM. These registers are the CCAM ID register and the CCAM Instruction Register.

CCAM ID Register: This register holds the id of the task which the next CCAM instruction works on. The ID for a 64 entry table is a number between 0 and 63 and is the index of the task's entry in the table. The CCAM returns information regarding the status of the last operation to the operating system via this register.

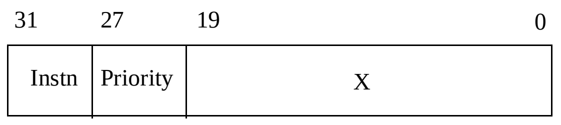
CCAM Instruction Register: The OS uses this register to load the CCAM instruction. Writing to this register initiates a CCAM operation. The CCAM has 15 valid instructions of which 10 operate on specified CCAM task entries while the remaining 3 access every valid task entry in



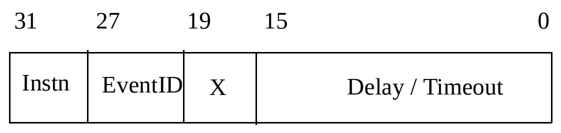
Format 1: Instructions with no data e.g. Priority and Delay Sort



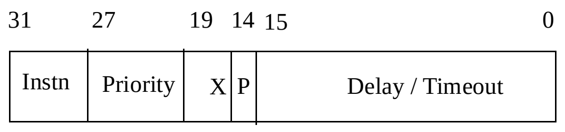
Format 2: CCAM Tick, Set Delay Instructions - 16 bit Data



Format 3: Set Priority and Remove Task



Format 4: Set Event Instruction



Format 5: Create Task Instruction

Fig. 3.3. Instruction Layout of CCAM instructions. This figure represents the interpretation of the different bits in a CCAM instruction word.

the CCAM. Issuing a CCAM instruction in majority of the cases is a two step procedure. In the first step, the operating system writes the index of the task on which the instruction is being performed into the CCAM ID register. Following this it writes the instruction into the CCAM Instruction Reg-

Table 3: State Assignments for Run Status

State	Assignment
Pause	000
Ready	100
Late	101
Block	010
Block Timeout	011
Release	110
Release Timeout	111

ister. The CCAM performs the operation and returns the result of the operation via the CCAM ID register. For example, when the OS issues a release event operation, the CCAM returns the id of the task that was made ready to run. Details of the instructions and their operations are given in Table 2. A layout of the different types of instructions is given in Figure 3.3.

The instructions occupy the upper 4 bits of the CCAM instruction word. The remaining 28 bits are used to pass extra information required to execute the instruction to the CCAM. For example, during task creation, information regarding the task - its periodic nature, its priority are all packed in the CCAM instruction word.

Table 2: CCAM Instructions

Instruction	Use
Create Task	Creates a task entry at a specified CCAM slot; Information regarding a tasks periodicity, priority, interrupt status are passed in the instruction word.
Delete Task	Invalidates the associated task entry
Set priority	Sets the priority of the specified task
Get priority	Returns the priority of specified task
Set Suspend	Sets suspend flag of task based on value passed
Get Status	Returns the Status Register Value of specific task
Set Delay: Release Time	Sets the Release Time for the specified task
Set Delay: Relative Deadline	Sets the relative deadline of a specified task or its period in the case of a periodic task
Get Delay	Returns the delay value in the delay1 field. This value is either the
Set Event	Blocks the task on specified event; Updates the delay field of task with event timeout value
Release Event	Searches for task with highest priority waiting on specified event id and releases it
CCAM Tick	Decrements all the delay values associated with valid task entries and updates their status appropriately
Get Task - Priority Sort	Returns the task ID of the highest priority task which is ready to run
Get Task Delay Sort	Returns the ID of the first task with the least delay (delay 1 field) which is ready to run

CCAM Time Tick Data Register: This register specifies the duration of a single CCAM tick. The value of the time is given in hundreds of microseconds.

CCAM Time Tick Register: This register serves as the periodic time source for systems using the CCAM for timeout queue management. The register is a counter that is incremented every CCAM tick. The operating system can reset this value during runtime.

3.3 Implementation

Run Status State Machine : The run status of each task can be implemented as a state machine with 7 states. Each state is associated with 3 bits. The state machine flow is given in Figure 3.2. The individual state assignments are given in Table 3.

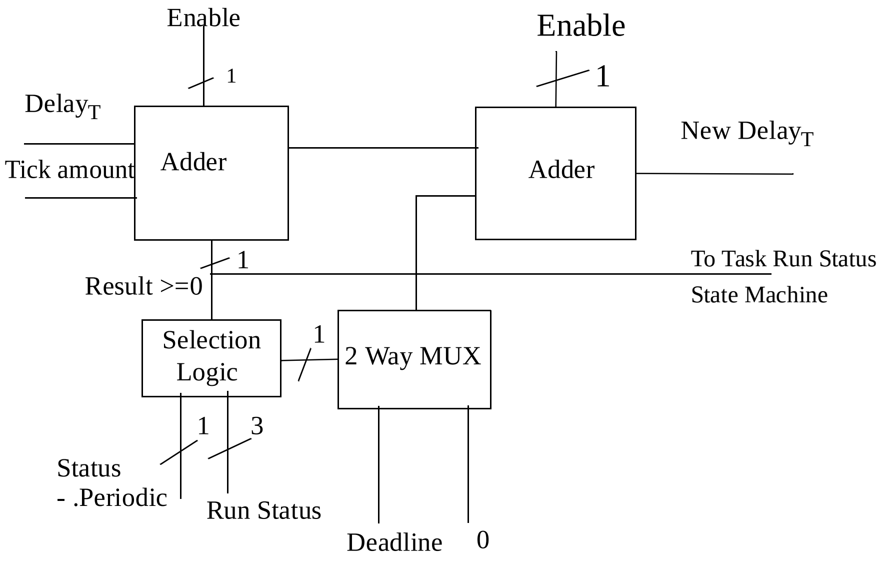
The inputs to the state machine are the following signals

Status. Periodic - This bit indicates if the task is periodic or not.

Set Event This signal goes high when a task is made to wait on an event i.e. a Set Event instruction is executed.

Set Delay_E This signal goes high when a value is loaded into the Delay_E field.

Set Delay_T This signal goes high when a value is loaded into the Delay_T field i.e. a Set Delay: Release Time Instruction is executed.



Selection Logic Cell

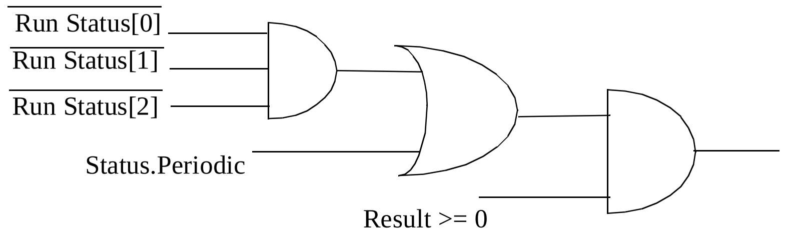


Fig. 3.4. CCAM Tick Module for handling Timeout Delays. This is a simplified representation of how the Delay$_T$ field gets updated every timer tick. Unless specified the wires are all 16 bit wide. The module to handle Event timeouts would be similar without the presence of the additional adder stage.

Delay$_T$ hits zero This signal is sent to the task status machine from the CCAM tick module. It is high when the task delay$_T$ field has reached zero.

Delay$_E$ hits zero This signal is sent to the task status machine from the

CCAM tick module which processes the tick for the Delay$_E$ field. It is set when the task times out on the event.

The output of the state machine is the 3 bit run status.

Tick Operations: The CCAM decrements the delay values every time a CCAM tick instruction is generated. Based on the result of the operation the run status of the machine gets updated. There are two delays that have to be serviced. The first is a delay value associated with release times and deadlines i.e. scheduling while the second is one associated with event timeouts i.e. synchronization based.

Servicing the delay value in the Delay$_T$ field of the CCAM is a two step procedure. The first involves subtracting the tick amount from the delay. The second involves updating the Delay$_T$ field if the result of the previous operation is non-positive. The module which handles the CCAM tick has two adder stages as seen if Figure 3.4. The first stage subtracts the tick amount from the current contents of the Delay$_T$ field.It thus takes as input the contents of the Delay$_T$ and the tick amount. The second adds to the result of this operation either zero or the stored deadline depending on the status of the task, the result of the operation etc. The inputs to this stage are thus the output from the preceding adder stage and the output of a 2 way MUX whose inputs are the contents of the deadline storage field and the value 0. Both adders have the same enable signal but the second one is enabled a clock cycle after the first. The operations are performed only if they are enabled.

The enable signal is high only if the following conditions are met

Valid Entry AND ((Run Status = Pause OR Run Status = Block Timeout OR Run Status = Ready AND CCAM Control. RD) OR Periodic Task OR CCAM Control.Negative Delay)

One of the inputs to the second stage as mentioned earlier is from a 2 way MUX. The output of the mux is dependent on the result of the previous stage as well as the status of the task. The output of the mux is the deadline of the task if the following conditions are met

($Delay_T$-TickAmount <=0 AND Periodic Task) OR ($Delay_T$-TickAmount <=0 AND Run Status = PAUSE)

The CCAM Tick handler module can be designed with a single adder and a state machine to keep track of what operation to perform next.

The $Delay_E$ field Tick handler module is similar to the $Delay_T$ module except that it only has the first adder stage. This module is enabled only if the following conditions are met

(State = Block Timeout) OR (State = Release Timeout AND CCAM Control.Negative Delay)

Sort Operations: The CCAM performs 64 element sorts in a single instruction. Instructions sort either 8 bit priority values or 16 bit delay values and return the tag of the task with the least magnitude priority or delay. Sorts are required by the following instructions

Release Event: This instruction identifies which one among a group of tasks waiting on a particular event to release. This requires an 8 bit priority sort. All tasks which are waiting on the event are sorted.

Get Task - Priority Sort: This instruction helps identifies which one of the tasks which are available to execute to run. This too requires an 8 bit priority sort of all tasks which are available for execution.

Get Task - Delay Sort: This instruction which is used by dynamic priority schedulers like Echidna helps select the task with the least delay or earliest deadline. This requires a sorting 16 bit elements which can be signed or unsigned depending on the CCAM configuration. The criterion for being among the set of elements which are sorted for this instruction is the same as the earlier instruction.

We use a tree of 16 bit comparators to perform the sorting. Each comparator element in the tree can sort 2 values and is capable of performing both signed and unsigned comparisons. Each comparator element has 2 16 bit inputs. It takes as input the value to be sorted and the ID of the tasks. Its

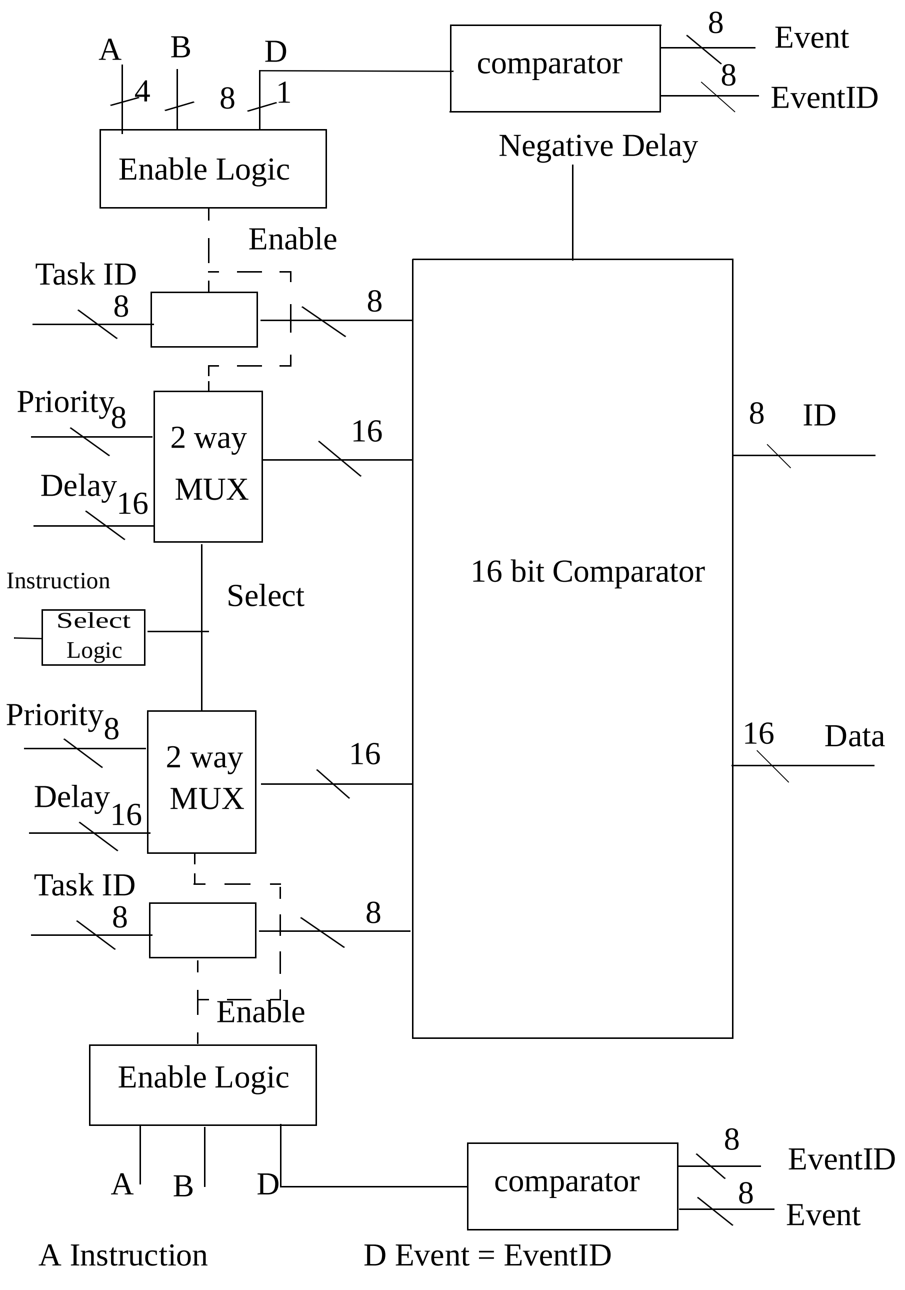


Fig. 3.5. Structure of Input Comparator Element. The figure shows the basic design of the comparator element that forms part of the CCAM sort tree. The comparator is capable of both signed and unsigned comparisons.

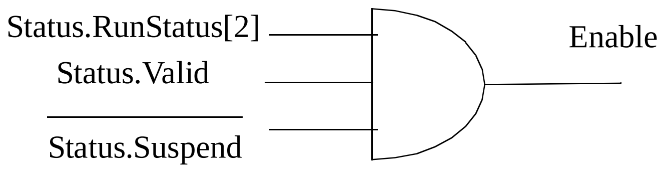


Fig. 3.6. Enable Logic for Priority /Delay Sort. The figure shows the test for a task element to participate in the sorts for the Get ID/Priority Sort and Get ID /Delay Sort instructions.

output is the ID of the task with the lower delay or priority value and the corresponding delay / priority value. For comparators in the first level the value which enters the comparator is selected using a two way tristated mux. The selection logic for the mux is based on the instruction word. The mux output is enabled depending on whether the task satisfies the criterion for the sort.

Enable Logic

- Enable - Priority Sort / Delay Sort

The pseudo-logic equation used to determine if a task entry satisfies the criterion for the sort is

(Status = Ready OR Status = Late OR Status = Released OR Status = Released Timeout) AND Valid AND Not Suspended.

Substituting the various literals for the above expression and minimizing it based on the state assignments made in Table 3 we get

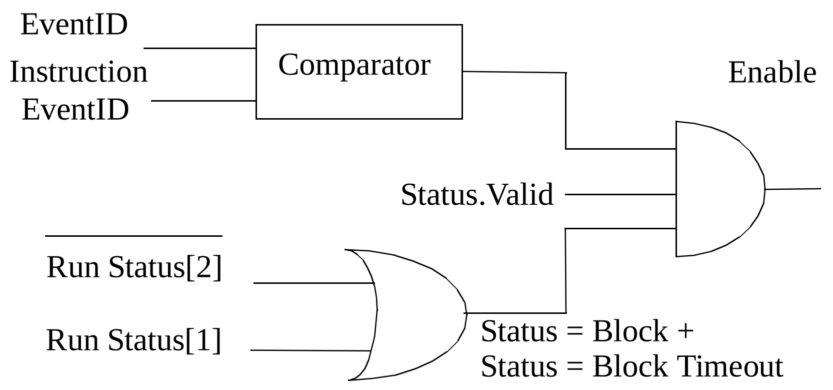


Fig. 3.7. Enable Signal for Release Event Priority Sort.

(Status. Run Status[2]) & Status. Valid & ~Status.Suspended i.e.

Additionally it should be checked that the instruction is either the Priority Sort or the Delay Sort.

It can be expressed then as

(Instruction = Priority Sort OR Instruction = Delay Sort) & Enable1

- Enable - Release Event

Tasks who participate in this sort have to satisfy the following criterion

(EventID in instructions = EventID in Entry) AND (Status = Block OR Status = Block Timeout)AND Valid

A comparator checks that the event ID in the instruction is the same as that which the task is waiting on. Additionally it should be checked that the instruction is the Release Event Instruction.

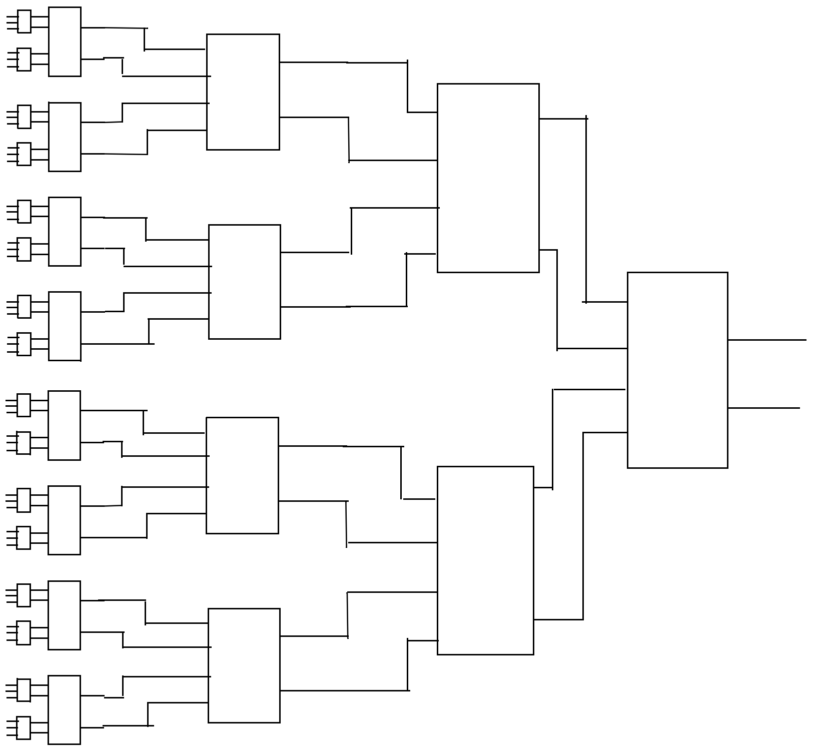


Fig. 3.8. Comparator Network. This figure shows a comparator network for a CCAM with 8 task entries. For 64 entry CCAM there would be 2 additional levels. The result at the end is a Task ID and its corresponding delay or priority.

Select Logic

This logic helps select the input data values based on the instruction. For Get Task - Priority Sort and Release Event instructions it selects the priority as input and for the Get Delay instruction it selects the Delay value.

The Comparator tree is given for a CCAM with 16 tasks in Figure 3.8.

3.3.1 CCAM Energy Model

The cost associated with a CCAM instruction depends on the number of basic operations the instruction performs. We assume that each of these basic operations—add, compare, load/store, logic operation—cost as much as an equivalent instruction on the Mcore processor. The basic operations arise from two distinct aspects of the CCAM instruction, namely the lookup and the actual instruction execution itself. Some CCAM instructions require fully associative lookups while others directly map into the CCAM. Fully associative lookups are required for operations like the release event, which requires identifying all tasks waiting on a particular event id. These lookups are more expensive as the number of comparisons is proportional to the number of elements in the CCAM. Thus depending on the nature of the lookup the number of compare operations associated with the instruction will change. The cost of executing the instruction is the cumulative cost of all the operations.

For example, the total cost of a set priority instruction is that of a direct lookup and a byte store. The CAM release event instruction, on the other hand, has associated with it the cost of a fully associative lookup and execution costs arising from the compare operations required to determine which of the valid entries has to be released.

3.3.2 Die Area Cost

The area occupied by the CCAM can be expressed by referring to models for on-chip memories proposed by Flynn et al[27]. The area can be expressed as

area = logic + data + tags + status.

The logic refers to the logic associated with the sort comparator trees, the associated state machines etc. The tags are the elements that are used in a fully associative look up which includes the event field. These fields are associated with a comparator element. We assume that the delay-timeout fields and delay-event fields which are associated with adders occupy the same die area. The data fields include the delay-deadline and the priority field. The total value after computation is 2600 rbe which is equivalent to a 48 wide 32 bit register file. This approximates to an area increase equivalent to that of a 1.5 register files or 8% of the chip area.

CHAPTER 4

EXPERIMENTS

All real time operating systems perform a certain set of operations to provide a framework for task management, event management and scheduling. In nearly all systems, these tasks are performed exclusively in software. We attempt to ascertain the benefits and trade-offs associated with moving some of the operating system's workload from software to hardware. We measure the overhead involved in performing these operations in terms of energy consumption, reliability in execution time and overall performance of the system.

The operating system that we used for our study was µC/OS, a multitasking preemptive real time operating system and Echidna, a cooperative multitasking operating system.

µC/OS : It is a public domain real-time operating system which is representative of current commercial RTOSes which employ static priority based scheduling algorithms[23] .µC/OS supports upto 64 tasks with each task being assigned a unique priority level. µC/OS also provides support for standard operating system services including IPC mechanisms like semaphores etc. It has been designed for modularity, scalability and determinism. We use two versions of μC/OS—the original preemptive version and a modified nonpreemptive version.The non preemptive version uses the same framework as that of the preemptive one but does not have any support for handling possible race conditions. Additionally the timer interrupt in the non preemptive version merely updates the time. Unlike its preemptive counterpart, the nonpreemptive OS does not perform any timeout queue management in the timer interrupt handler. All timeout queue update operations are performed at scheduling boundaries. The granularity of the system clock for μC/OS is dependent on the frequency of the timer interrupt.

Echidna : This is a commercial dynamic priority RTOS which treats threads as port based objects [37 , 38] . It provides support for reconfigurable software-based components. Echidna provides its tasks a non-standard API. Echidna does away with the standard while(1) construct. It provides an API where the developer has to partition the task into different sections based on their function viz. initialization, termination, synchronization, cycle or main body. The Echidna scheduler is based on a non-preemptive implementation of EDF. It also provides high resolution system clock by performing interpolation using the timer counter registers.

The configurations that we study include the following:

Original: In this configuration all the task, event and time management and scheduling is done exclusively in software. The system clock is maintained in software and updated every timer interrupt.

With the CCAM: In this configuration the operating system uses the CCAM to hold information regarding the state of the tasks in the system. The operating system queries the CCAM when it makes scheduling decisions, performs event management etc. The system clock continues to be maintained in software. Although the hardware maintains the timeout queue of the system the operating system directs the update of this queue. For the preemptive version, this configuration has a regular timer interrupt during which the system performs these two tasks. For the nonpreemptive case the timer interrupt serves only to update the system clock. The timeout queue is only updated prior to scheduling.

With CCAM in Auto Decrement Mode: This configuration is similar to the earlier mentioned scheme. The main difference is that the system clock management and timeout queue update is performed independently of the software. The CCAM updates the clock and task timeout delays automati-

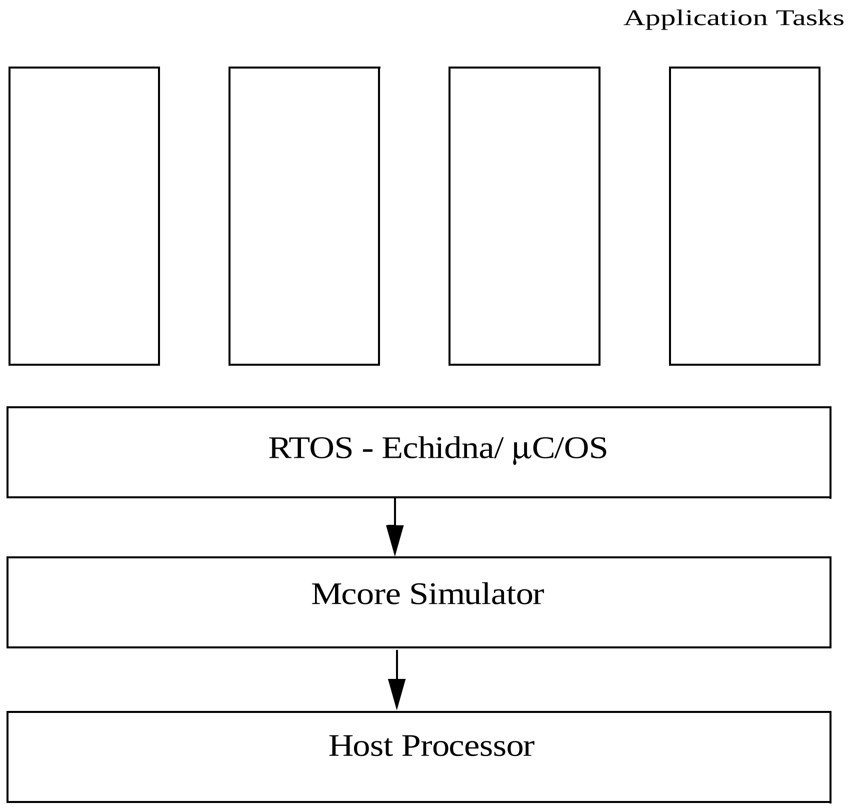


Fig. 4.1. Experimental Setup. The figure shows the experimental setup for this study. The Mcore C simulator runs on a desktop machine. The RTOSes execute directly on the Mcore simulator.

cally every “timer tick”. The CCAM can be configured to interrupt the system when a new task is available to run following the update.

Using the periodic feature: In this configuration the CCAM supports periodic tasks. When these tasks are created, the period is specified. For these tasks the CCAM automatically reloads the period into the delay field every time it hits zero. We do not use this feature with Echidna.

We also measure the effect of varying the resolution of the system clock on the above from 500 Hz to 2 kHz. The Echidna configuration is also tested for a higher resolution clock.

We test the configurations using different workloads. The workloads are selected from the MediaBench suite of benchmarks [7]. The applications used for this study include the following:

G721: This application implements the CCITT's (International Telegraph and Telephone Consultative Committee)g.721 voice compression algorithm.

ADPCM: This application performs adaptive differential pulse code modulation on a given audio sample.The encode converts a raw 16 bit PCM sample to a 4 bit ADPCM sample while the decode does the reverse operation.

GSM: This application performs speech transcoding. It compresses frames comprising of 160 13-bit samples to 260 bits.

The Mediabench applications are compute-intensive and each consists of two tasks: an input task that runs at a predetermined rate and an output task that is released every time there is valid data for it to process. The two tasks communicate to each other using standard interprocess communication mechanisms like semaphores. In addition to the application workloads we

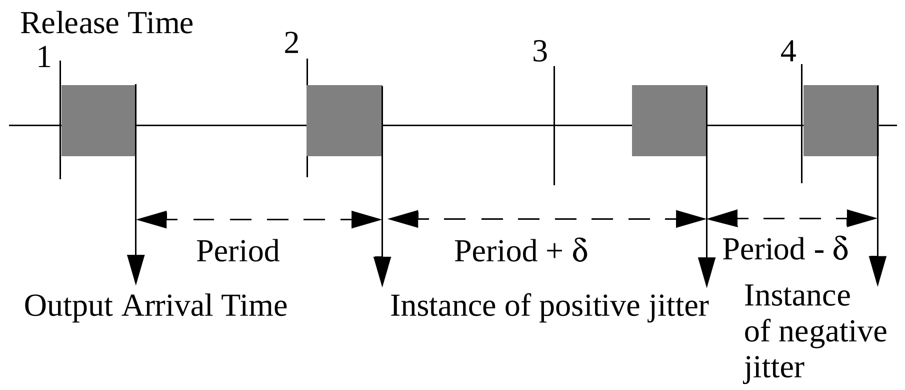


Fig. 4.2. Jitter Example. The figure shows how jitter is measured for a periodic task with a fixed period. The result of invocation 2 of the task arrives on-time i.e. one period interval after the last result. Invocation 3 starts late and the inter-arrival time is off from the period by δ. The next invocation starts execution on-time but its output arrives early with respect to the last output.

also have various background applications running on the system to provide variability in system load, scheduling, and thus task timing. These include the following:

control loop task: This is a periodic background task that runs every 32ms and represents background workload commonly seen in embedded systems.

aperiodic hardware-generated I/O interrupt: This interrupt occurs with a fixed probability on every cycle, yielding an exponential inter-arrival time distribution with average interrupt rate of 100 Hz.

We measure the performance of the system in terms of jitter, energy consumption, overhead and execution time of operating system tasks.

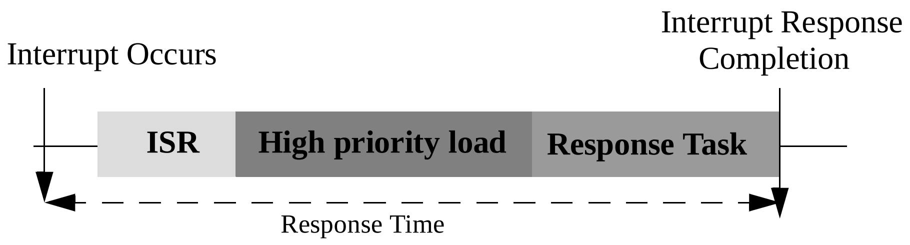


Fig. 4.3. Response Time to External Interrupt. The figure depicts the handling of an interrupt. The interrupt is handled by an ISR which releases a response task which completes servicing the interrupt. The response time is the time between the arrival of the interrupt and the completion of the response task servicing it.

Jitter: We study the variation in the inter -arrival times between output of periodic tasks. If the inter-completion time is different from the period we say that the task has a jitter. If the output arrives early we say that the task has negative jitter and if it arrives late we say that it has positive jitter.

We consider the example in Figure 4.2 where the periodic task is executing with a fixed period. The output from invocation 2 arrives exactly one periodic interval from the last seen output. Thus the particular invocation is said to have zero jitter. By contrast the output of invocation 3 is available a time interval Period + δ after the output of invocation 2. The output is off by an amount δ or it has a positive jitter of δ. Invocation 4 starts execution on-time but its output is available in time period -δ after the previous output or the invocation has a negative jitter of δ.

Response Time: The response time to an interrupt is the time since the arrival of the interrupt and the completion of its servicing. This completion is marked by the end of execution of the tasks which are executed in response to the interrupt. We consider a system which uses a split interrupt servicing strategy. The interrupt servicing is done in two stages. The first being the execution of the actual ISR itself which handles any hardware specific responses and releases a high priority response task to complete the handling of the interrupt.The second stage is the execution of the high priority response task which executes the non-critical part of the interrupt handling.

Figure 4.3 depicts the typical interrupt handling in a system using split interrupt servicing. The ISR is not invoked as soon as the interrupt arrives because interrupts may be disabled when the interrupt arrives. Besides this there is an additional overhead of flushing the pipeline and loading the ISR. The ISR releases a high priority task which is executed only after the other higher priority tasks viz critical OS and user level tasks, complete execution.In the absence of these critical tasks there will always be a delay due to the overhead of performing scheduling and a context switch.

RTOS Overhead: Assuming that the application's energy consumption will be roughly constant across different operating systems, we measure the additional energy that the operating system consumes while managing the workload. We separate the energy values based on the function of the OS code including semaphore management, time management, context switching, interrupt handling, enabling/disabling of interrupts, scheduling, task management and initialization of the system.We also measure the CPU overhead that RTOS functionality represents.

Execution Time: We measure variability in the time taken by the operating system to perform operations such as pending and posting a semaphore, delaying a task and to service an interrupt. We measure the time between the initiation of these tasks and the scheduling of the next task on the system.

CHAPTER 5

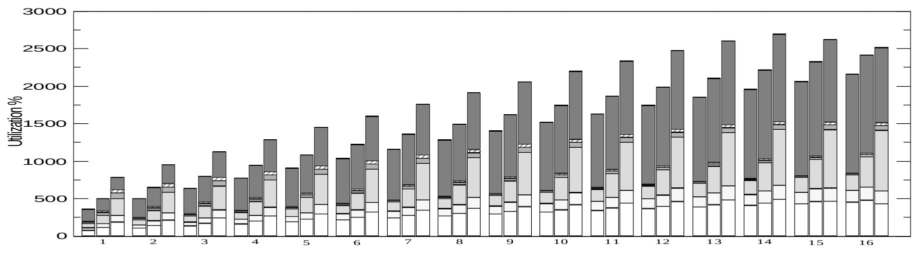
RESULTS

5.1 Overhead

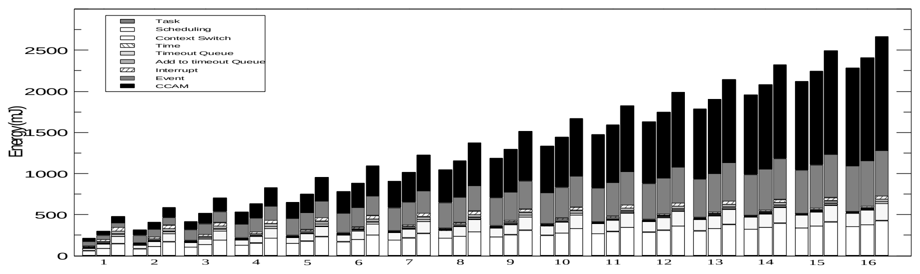
5.1.1 Preemptive µC/OS Configurations

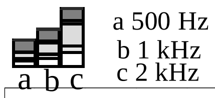
Timeout queue maintenance has two aspects - entering a task on the timeout queue and updating the timeout queue every timer interrupt. The former factor is dependent on the behavior of the tasks in the system while the latter varies with the system clock frequency.

The bulk of the energy savings gained while using the CCAM in preemptive configurations of µC/OS arises from altering the fashion in which the timeout queue is updated. Timeout queue updates in µC/OS are performed by walking through a task queue containing all the tasks in the system, decremented non-zero delays and releasing those tasks whose timeout values have reached zero. We replace this queue traversal with a single CCAM instruction, the CAM tick. This instruction automatically decrements the delay associated with every task entry in the CCAM. The cost of this instruction for a single task is that of the decrement operations to update

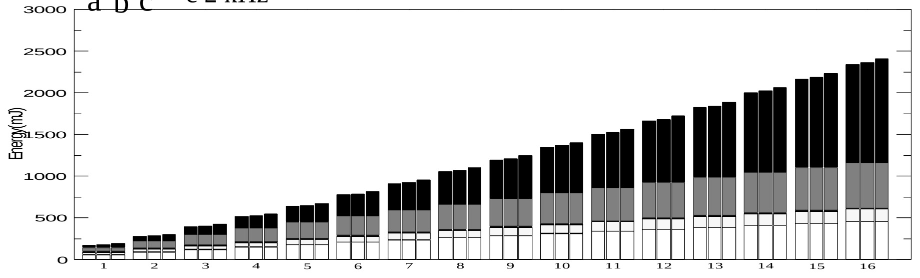


(a) Original µCOS

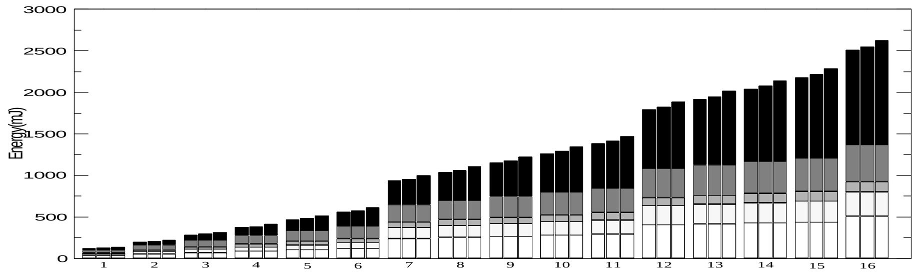




(b) With CCAM



(c) With CCAM in Auto Decrement Mode



(d) With CCAM using Periodic Feature

Fig. 5.1. Energy Consumption for preemptive µCOS configurations. The graphs show the variation in the energy consumption of the RTOS running the g721 decode benchmark at 4 ms periods. The y-axis represents the energy consumed by the kernel while the x-axis represents the workload size. The data for different system clock rates are grouped at each workload point.

the timeout value, comparison and logical operations to determine if the timeout value needs to be updated and store operations required to update the task status and its timeout value if needed. The energy overhead for performing timeout queue maintenance decreases to 6% - 20% of the original value on an average as seen in Figure5.1(a) and (b). Increasing the system clock frequency in μC/OS results in a proportional increase in the number of times the timeout queue is updated. The overhead in a software managed timeout queue system is far greater than in systems with hardware maintained timeout queues.

The μC/OS timer interrupt handler updates the timeout queue as well as the system clock. Both these operations are performed automatically by the CCAM when it is used in the auto decrement mode. An interrupt is generated every time a task on the timeout queue becomes available to run. Thus the interrupt only serves as an indicator to the operating system that a new scheduling point has been reached and has a negligible overhead. This allows the system to continue executing without being sidetracked to perform housekeeping chores. This reduces the interrupt arrival frequency approximately to that of the minimum task frequency. Operating system configurations using the auto decrement mode of the CCAM have approximately the same software overhead for the same workload irrespective of the frequency of the system clock as can be seen in the Fig 5.1(c).Another interesting side effect of this is the reduction in context switch and scheduling overhead. As there are less interrupts the RTOS has to perform less context switches and this can be seen in Fig 5.1.

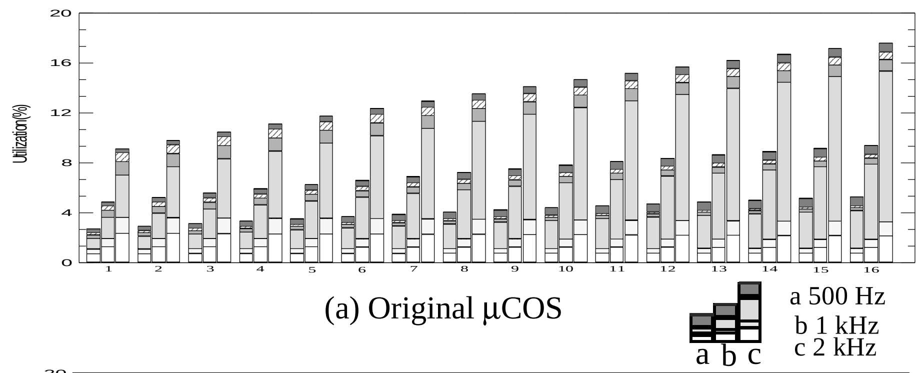
Using hardware for event management reduces the number of instructions executed upon a semaphore pend or a semaphore post by nearly 50%. This is because the hardware based scheme eliminates the need to update software data structures like the ready list and the event block. Semaphore pend operations are now reduced to just merely updating the event field, the delay field and the status of the associated task in the CCAM. When a semaphore is posted the hardware sorts all tasks waiting on the particular event id and returns the id of the task that got released. It also updates the status of the task simultaneously. These hardware operations reduce the software overhead to 30 - 40% of its original values. Event overhead does not change with the system clock frequency as it is workload dependent.Thus all configurations of the CCAM have the same event overhead.

Using the CCAM reduces the scheduling overhead by only around 20%. The current scheme employed by µCOS which uses bit vectors has a low constant overhead irrespective of the number of tasks in the system.

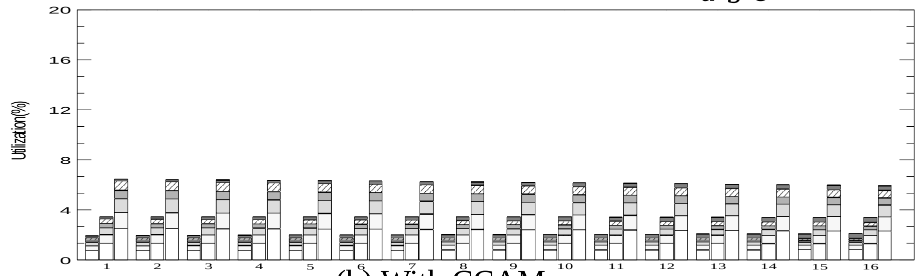
Overall employing the CCAM can reduce energy overhead by as much as 20-40%.

Fig 5.2 shows the variation in the processor utilization by the RTOS for different configurations.The data is represented for the benchmark gsm decode running with a period of 160 ms.

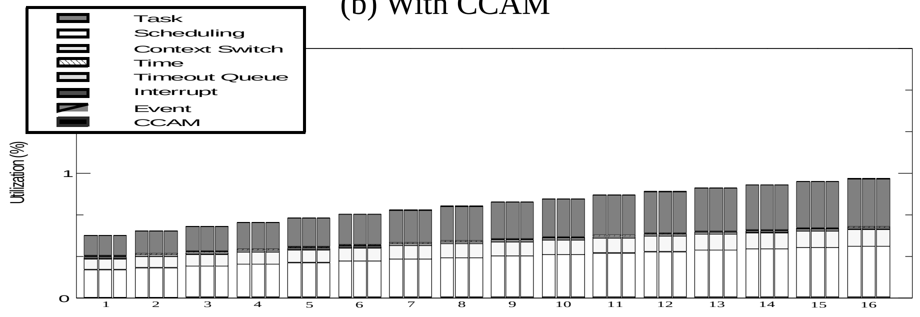
RTOS overhead is of two types - a workload dependent fraction and a workload independent section. The workload dependent fraction comprises of those functions of the RTOS which tasks explicitly request for like being placed on the event queue or on the timeout queue. The overhead of these operations depends largely on the frequency at which applications request these services. Workload independent fractions comprise of those operations which the RTOS has to perform independent of the workload behaviour. This includes maintaining the system clock and updating the timeout queue. The frequency of these operations are independent of the workload but the overhead i.e. execution time may vary with the workload size. Scheduling has both a workload dependent and workload independent fraction. The former arises from the fact that scheduling has to be performed every time a task requests to be blocked or gets unblocked either when it hits its release time or an event is posted. The latter refers to the scheduling and context switching which is performed following every timeout queue update.



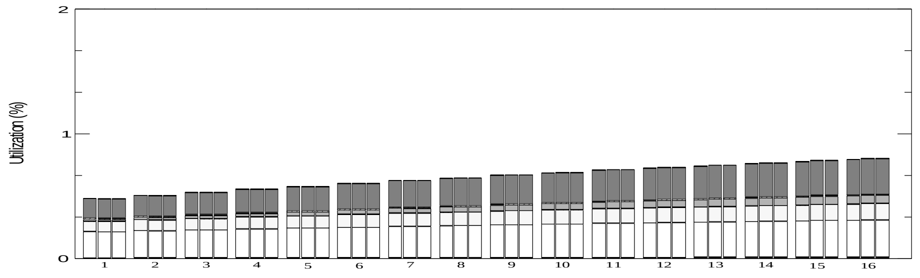
(a) Original μCOS



(b) With CCAM



(c) With CCAM in Auto Decrement Mode



(d) With CCAM using Periodic Feature

Fig. 5.2. Utilization for preemptive μCOS configurations. The graphs show the utilization of the various components of the RTOS for 4 different preemptive configurations. The graphs are representative of data for the gsm decode benchmark executing with a period of 160 ms. The x-axis represents the number of tasks and the y-axis the processor utilization. Data for different system clock rates are grouped together at each workload point.The y-axis scale of graphs c and d are from 0-2% while that of a and b are from 0-20%.

For the original μC/OS configurations the processor utilization increases linearly with the workload. The bulk of the increase comes from the time-out queue management overhead which can have an overhead of as little as 1% to as much as 12%. As the workload runs with a 160 ms period it uses RTOS functionality like semaphores etc. less often than an application like g721 decode which runs with a period of 4 ms. Thus the event and scheduling overhead are less affected by changes in the workload size.As the system clock overhead increases we observe that the scheduling and context switch overhead also increase proportionally. This is because following every timer queue update the RTOS has to invoke the scheduler and perform a context switch as well. Increasing the timer tick frequency increases the number of such scheduling operations and in turn the overhead.

For the μC/OS which uses the CCAM the timeout queue management overhead is constant for all configurations. This is because the complexity of this operation has decreased from O(n) operation to that of a O(1) operation. Thus the overhead of the RTOS is less affected by workload increases and does not go beyond 6% of the total usage. The scheduling and context switch overhead continues to scale with the system clock frequency like in the earlier scheme. This is because the configuration sees as many interrupts over a run as the earlier scheme does.

Moving the timeout queue updates to the hardware further reduces the RTOS overhead to under 2%. This is because the overhead which was workload independent has been eliminated. Timer ticks are processed in the hardware and the software is invoked only when a new task is available for execution. Thus scheduling is performed less frequently than the earlier configurations. As the workload behavior does not change with the system clock overhead we see that for a given workload the RTOS overhead stays the same as the system clock rate changes. Additionally using the periodic feature reduces the RTOS overhead because it no longer has to execute and schedule a separate task which ensures that all periodic tasks stay on beat.

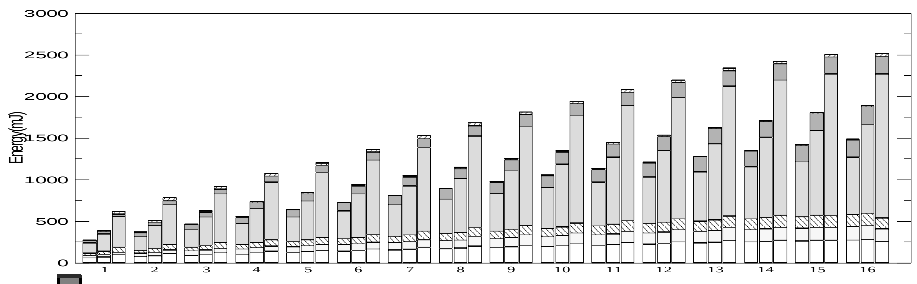
5.1.2 Non preemptive μC/OS configurations

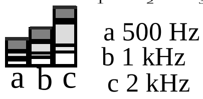
The bulk of the energy savings gained from using the CCAM in the non preemptive case comes from moving the timeout queue update from software to hardware. In the case of a non preemptive system timeout queue updates take place either when a task completes and the RTOS schedules a new task or when the RTOS is idling and a timer interrupt occurs. Thus the frequency of timeout queue updates depends on the system clock frequency as well as the nature of the workload.The overhead of a single update operation is dependent on the size of the workload.

For lightly loaded systems there is a higher probability that the RTOS will be idling when a timer interrupt arrives and a timeout queue update can be serviced at the instant the system clock gets updated. In addition to these timer interrupt driven updates the RTOS attempts to perform an update at every scheduling point if required. Thus if there are more scheduling points the resulting timeout queue overhead will also be higher.

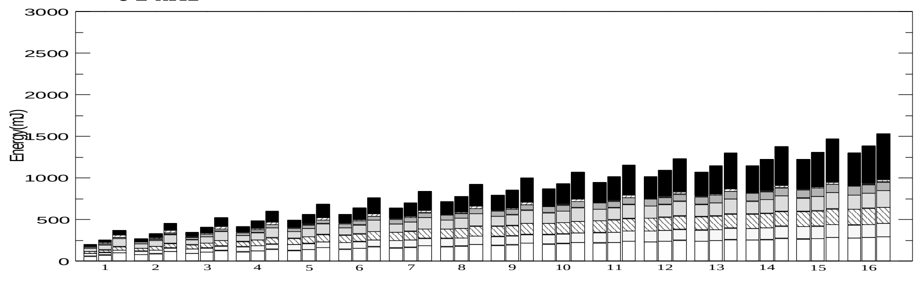
The benefit of the CCAM tick instruction as in the preemptive case arises because it reduces the overhead of an individual timeout queue update operation. This reduces the energy consumption overhead of the timeout queue update to nearly 10% of the original. Further using the auto decrement mode eliminates the need for the software to perform any sort of timeout queue update.

In the case of a heavy workload like gsm decode whose utilization numbers we have in Figure we see that the timeout queue update overhead initially increases with workload size but decreases after a certain workload. This is because as the number of tasks increases the overhead of a single timeout queue update operation increases. The number of scheduling points at which the operation is performed also increases. These increases are accompanied by a decrease in the idling time and a chance that more timer ticks pass before a timeout queue update is performed. The two fac-

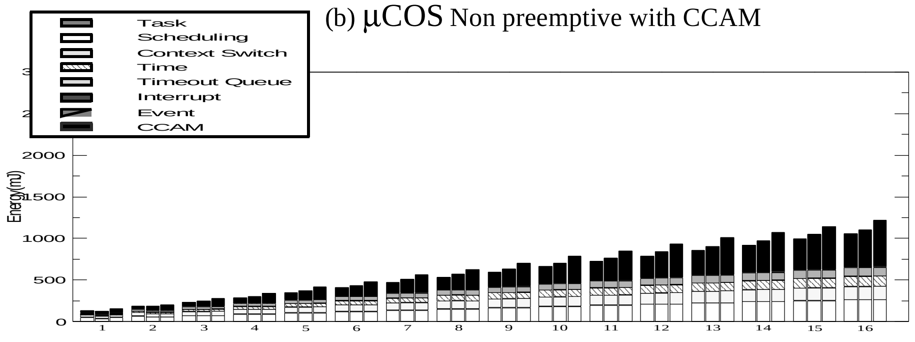




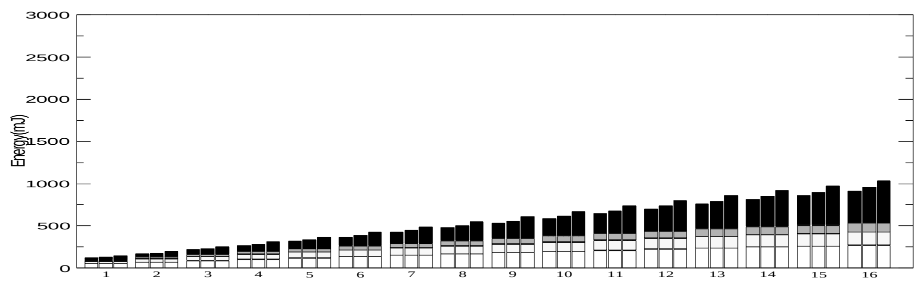
(a) µCOS Non preemptive



(b) µCOS Non preemptive with CCAM



(c) µCOS Non preemptive with CCAM in auto decrement mode



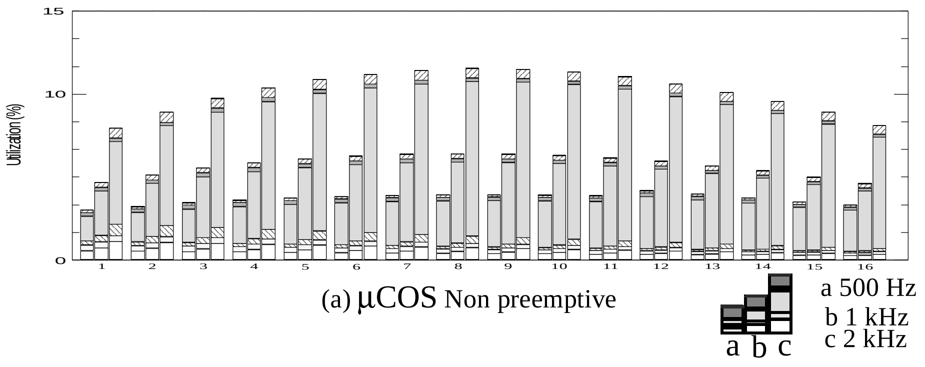
(d) µCOS Non preemptive with CCAM in auto decrement mode using period featu

Fig. 5.3. Energy consumption of non preemptive µCOS configurations. The graphs show the variation in the energy consumption of the RTOS running the g721 decode benchmark at 4 ms periods. The y-axis represents the energy consumed by the kernel while the x-axis represents the workload size. The data for different system clock rates are grouped at each workload point.

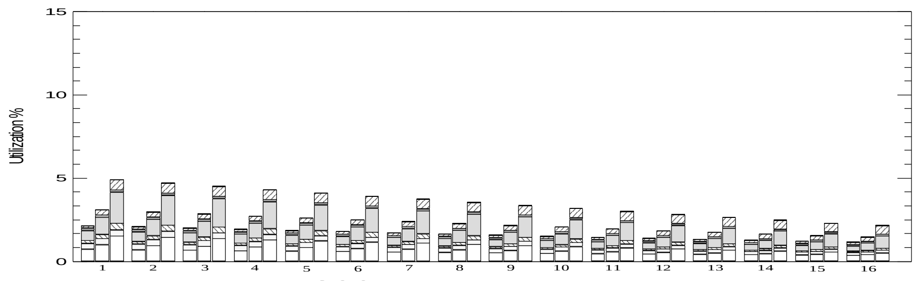
tors work in opposite directions and depending on which one dominates the overhead of the timeout queue operation either increases or decreases with workload. Using the CCAM eliminates the first factor i.e. increase in overhead of single timeout queue update operation with workload. This results in a drop in the utilization as the workload increases. This is less obvious for the CCAM employed in auto decrement mode.

Using the CCAM does not reduce the overhead of adding a task to the timeout queue. The overhead of this operation in μCOS is constant and small and involves updating the delay field in the task entry block and removing it from the ready queue. The hardware instruction performs the similar operation i.e. updates the task entry and the task status register.

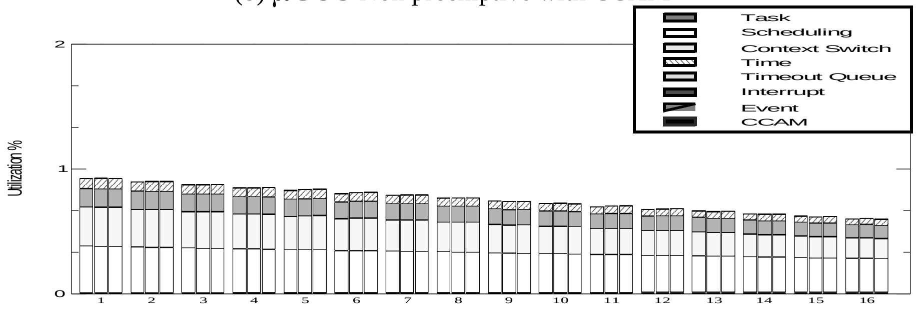
A nonpreemptive OS polls time more often than a preemptive OS. Every time a timeout queue update is performed or attempted the RTOS reads the current time in order to determine how many timer ticks have elapsed since the last update.This polling overhead increases with the workload and timer frequency as can be seen in Figure 5.3 (a). Using the CCAM does not lower the overhead as can be seen in Figure5.3 (b). Employing the CCAM in auto decrement mode eliminates the need to poll the time in order to maintain the timeout queue. There is a further decrease in this component



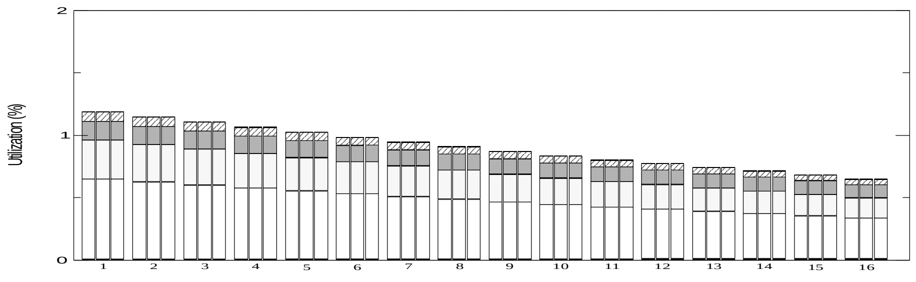
(a) μCOS Non preemptive



(b) μCOS Non preemptive with CCAM



(c) μCOS Non preemptive with CCAM in auto decrement mode



(d) μCOS Non preemptive with CCAM in auto decrement mode using period feature

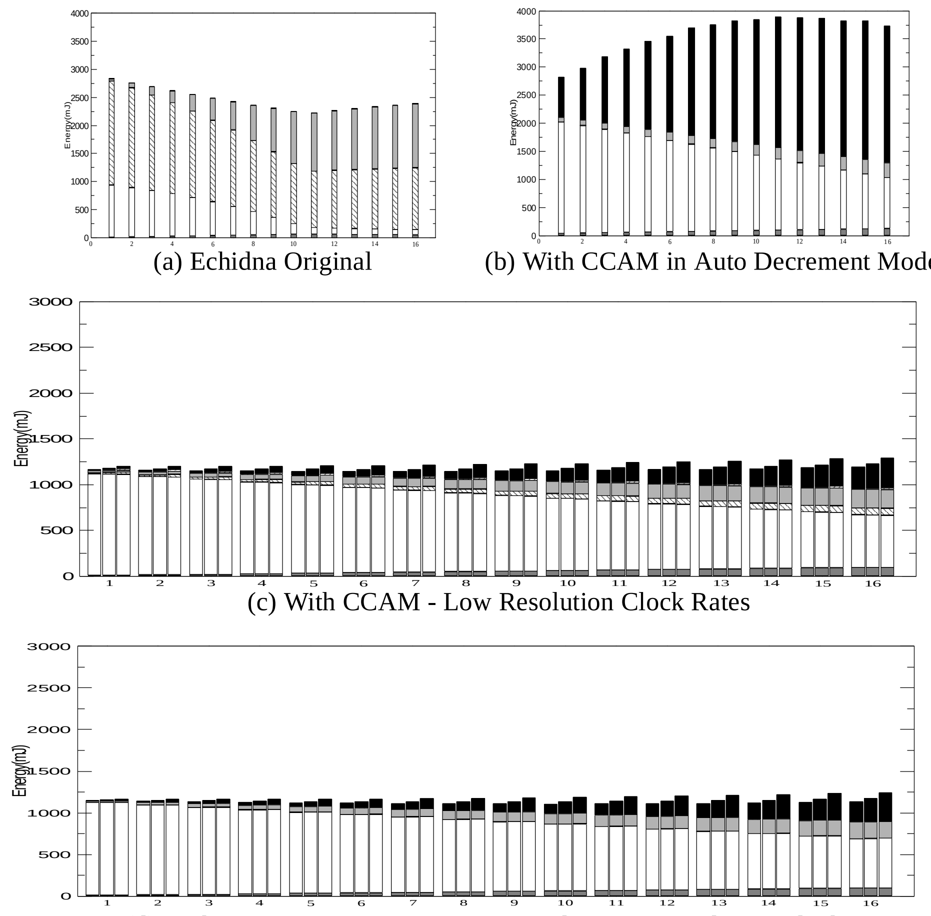
Fig. 5.4. Utilization Overhead for non preemptive μCOS configurations. The graphs show the utilization of the various components of the RTOS for 4 different non preemptive configurations. The graphs are representative of data for the gsm decode benchmark executing with a period of 160 ms. The x-axis represents the number of tasks and the y-axis the processor utilization. Data for different system clock rates are grouped together at each workload point.The y-axis scale of graphs c and d are from 0-2% while that of a and b are from 0-20%.

in the case of the configuration using the periodic feature because tasks no longer need to determine their next release time.

5.1.3 Echidna

The bulk of the RTOS overhead is seen in three components viz. the scheduling, the polling of the system clock and adding tasks on the various scheduling queues. Echidna maintains a ready queue that is sorted based on deadlines and a pause queue i.e. timeout queue with tasks sorted based on release times. Every time a task completes the scheduler checks to see if there are any tasks which are ready to be released. This involves comparing the current operating system time with the release time of the task. If the release time is past then the tasks are moved from the pause queue to the ready queue. When all newly released tasks have been moved the RTOS runs the task with the earliest deadline first. If there are no currently ready tasks the RTOS continues to poll the pause queue till a task becomes available to execute. Thus even when the system is idle the RTOS constantly polls the ready and pause queues and the system clock. Thus a system with a low workload i.e. one task has a high RTOS overhead. This idling polling overhead decreases as the number of tasks in the system increases. Thus we see that the RTOS overhead gradually decreases with system load. After a certain workload this no longer holds true. This is because the overhead of adding tasks to the queue now increases. Inserting a task into a sorted queue is an O(n) operation. The overhead of this operation increases with the number of tasks in the system. When this increase dominates the RTOS overhead we see that the RTOS overhead gradually increases.

Using the CCAM with a system clock resolution of the same order increases the overall energy consumption of the system. For a configuration with 1 task it is nearly the same but this energy consumption gradually increases with workload and saturates at a workload of around 12 tasks. The CCAM configuration has no software overhead for adding tasks to the various tasks queues. The CCAM reduces the complexity of adding a task onto a queue from O(n) to O(1). Adding a task onto the pause queue or timeout queue involves setting the timeout delay or updating the delay field and the task status. This eliminates the need for an O(n) insertion onto the pause queue. At the time of placing a task onto the pause queue the RTOS can set the deadline value also. This value is loaded into the Deadline field and automatically loaded into the Delay-Timeout field when the release time reaches zero. Thus the overhead of placing the task on the ready queue has also been reduced to a constant overhead. The scheduling overhead is higher than the case of the original configuration because the processor has



(a) Echidna Original (b) With CCAM in Auto Decrement Mode

(c) With CCAM - Low Resolution Clock Rates

(d) With CCAM in Auto Decrement Mode - Low Resolution Clock Rates

Fig. 5.5. Energy Consumption of Echidna based configurations. The graphs show the variation in the energy consumption of the Echidna kernel with workload for the g721 decode benchmark executing with a 4ms period. The y-axis shows the energy while the x-axis depicts increasing workload. Graphs a and b have a clock resolution on the order of 100 us. Graphs c and d show the energy overhead for CCAM configurations using a lower system clock resolutions. Each workload point has data for 3 different system clock rates - 500 Hz, 1 kHz and 2 kHz.

more idle time and thus checks if there is a task available for execution more often.

Bulk of the CCAM energy is consumed in performing a regular update to the timeout values and from checking if a task is available to run. The former operation is performed every 100 microseconds. In the case of the software based configuration the kernel performs a comparison with the timestamp of the task element at the head of the pause queue or the ready queue at every scheduling point or when the system is idling. The update is more expensive than this simple comparison. In addition the idle time in the system increases by employing the CCAM. This results in the RTOS constantly polling the CCAM in order to determine if there is a task to run.

The energy consumption scales with workload and tapers off. This is because like earlier there are two conflicting trends seen with increasing workload - one is the decreasing idling time or lowering of polling overhead and the other is the overhead in providing the services to the various tasks. The former increases far more rapidly than the latter decreases till at a workload where they both compensate each other.

Figure 5.5 c and d show the variation in energy consumption for lower resolution system clocks 500 hz, 1 kHz and 2 kHz. These configurations have a lower energy overhead than both the previous operations. They also avail of the doze instruction to lower the energy overhead. These configurations have the benefit of reducing the overhead to add a task onto the various system queues and also eliminating the need to poll the time constantly in order to determine if a task is ready to go. This reduces the overhead of the RTOS by 50%. The lower clock resolution does not adversely effect jitter but decreases energy consumption considerably.

The energy consumed the CCAM increases proportionately with workload as can be seen in the graphs Figure 5.1, Figure 5.3 and 5.5 But for a given configuration the CCAM energy increases marginally with increase of the frequency of the system clock. This is because the CCAM operations that increase when the system clock rate increases are those used in timeout queue management and involve only a single compare and decrement operation for each valid task entry.

5.2 Operating System Functional Measurements

We measured the time between the initiation of an operating system function, like taking a semaphore, to the time that control is handed back to the user level code. The operating system functionality that we measured includes posting a semaphore, pending on a semaphore, delaying a task and the interrupt overhead. The numbers include the time spent in the operating system function as well as those associated with scheduling and performing a context switch.

The duration of all these operations is independent of the workload running on the system or the resolution of the clock. The design of the µC/OS kernel is such that the overhead of nearly all system calls is constant and

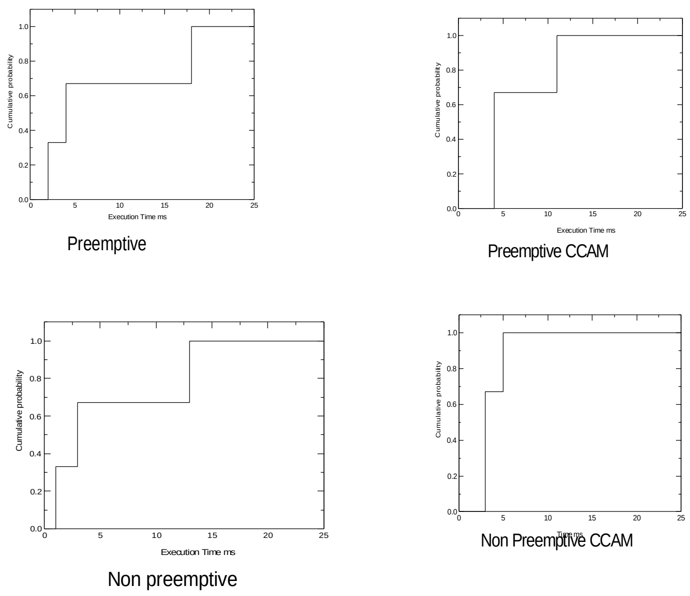


Fig. 5.6. Operation System Overhead for Semaphore Pend. The x-axis represents the time in ms between the initiation of a particular OS provided function and the execution of the next scheduled task. This time includes the overhead of performing the operating system function and scheduling the next task. The y-axis represents the cumulative probability for each point.

does not scale with the workload. Increasing the system clock rate increases the RTOS management overhead only and not the overhead of the interfaces it offers tasks. In Figure 5.6, 5.7 and 5.8 we present the results for the system running 6 g721 decode tasks running with a periodicity of 4 ms.

For all operations the overhead of the non preemptive system is much lower than that of the preemptive system. This is because non preemption guarantees that no task will ever be interrupted thereby eliminating the need for marking code as protected. Protection of the code represents two thirds of the overhead of operating system operations. For example a non-preemptive time delay takes around 5 μsec. while the preemptive configuration takes nearly 15 μsec. to perform the same operation.

The semaphore pend operation's overhead is dependent on whether the resource is already available or not. When the semaphore is available the operating system merely decrements the resource count and then schedules the task again. This overhead is approximately the same for configurations using the CCAM or not. When the resource is unavailable the operating system has to update the event's control block in addition to performing

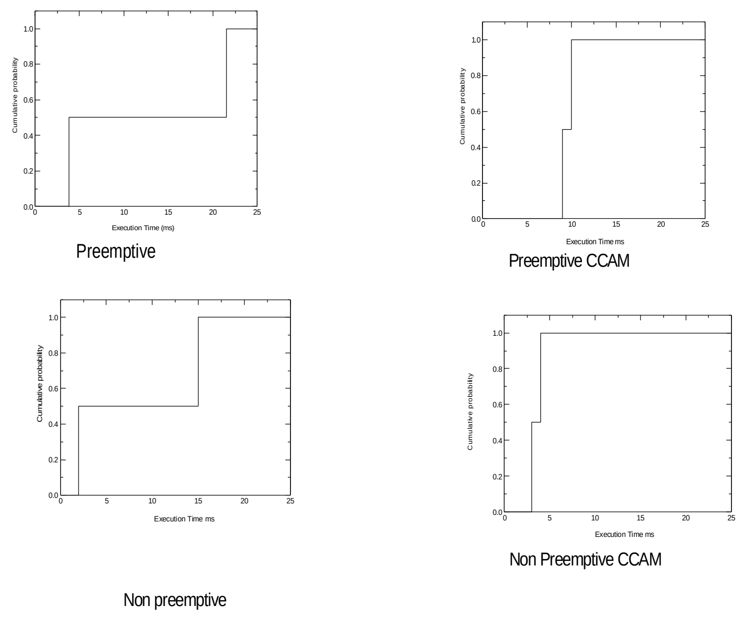


Fig. 5.7. Operation System Overhead for Semaphore Post. The x-axis represents the time in ms between the initiation of a particular OS provided function and the execution of the next scheduled task. This time includes the overhead of performing the operating system function and scheduling the next task. The y-axis represents the cumulative probability for each point.

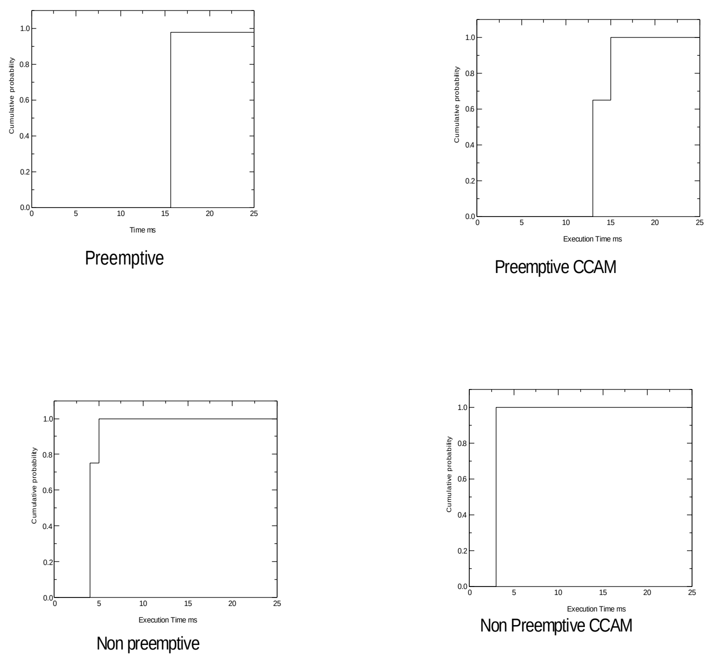


Fig. 5.8. Operation System Overhead for Time Delay. The x-axis represents the time in ms between the initiation of a particular OS provided function and the execution of the next scheduled task. This time includes the overhead of performing the operating system function and scheduling the next task. The y-axis represents the cumulative probability for each point.

scheduling and a context switch. The overhead of updating the event block in both preemptive and nonpreemptive versions of µC/OS is expensive and results in overheads as high as 17 µsec. for preemptive configuration to 14 µsec. for nonpreemptive configurations. For RTOS configurations using the CCAM the update to the event block is no longer required. Each task entry in the CCAM is associated with an event ID. Writing into the latter is equivalent to the update event block operation performed in software. This reduces the overhead of semaphore pend operations to 10 µsec. in preemptive operations and to 6 µsec. in non preemptive configurations.

Similarly semaphore post operations executed in operating systems with hardware support take less time than those without. The greatest overhead in semaphore post operations is determining which task needs to be unblocked and then removing its entry from the event control block. Upon a semaphore post, the CCAM does a search based on the event id and returns the id of the task with the highest priority. It also simultaneously marks the task as ready to run and resets the event field associated with the particular tasks entry. As a result the overhead of a semaphore post is reduced by nearly 5 µsec.

Delaying a task for µC/OS versions with and without the CCAM have about the same overhead. This is because in both versions this involves updating a single delay field either in a software block or in a hardware data structure.

In general, one of the most important results as concerns real-time systems is the clear reduction in variability of execution time by using the CCAM. As the semaphore operations show, using the CCAM all but eliminates variability in all operations, save semaphore pend for preemptive CCAM.

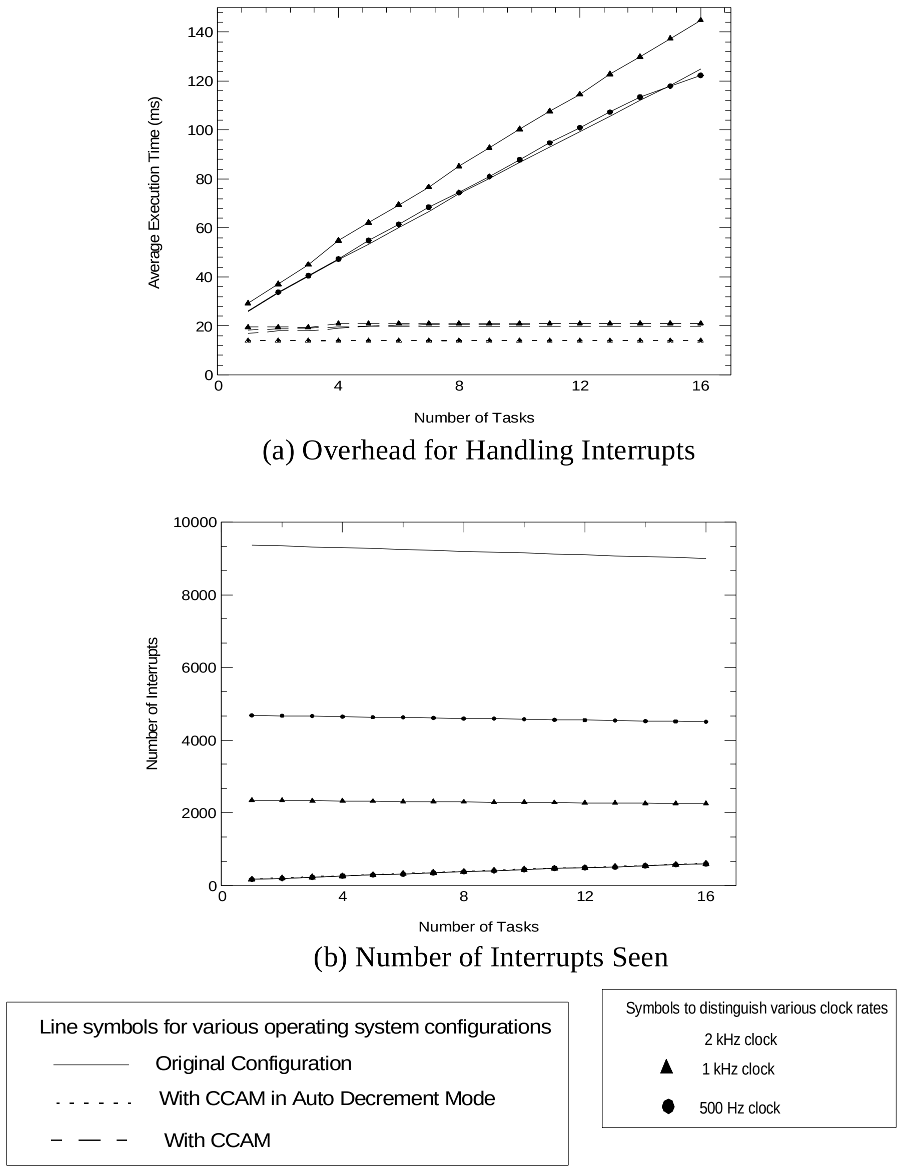


Fig. 5.9. Interrupt Overheads. Graph (a) shows the overhead of handling the timer interrupt as workload increases for various operating system configurations. Each of the configurations is studied with different system clock rates. The benchmark is G721 Decode running at 4 ms periods. Graph (b) shows the variation in the number of interrupts seen with increasing workloads. The benchmark used is ADPCM Decode running at 144 ms periods.

In Figure 5.9 we study the interrupt overhead across the different configurations. As expected, the overhead linearly increases with the workload for the original configurations of μC/OS. As the system clock rate decreases the average interrupt overhead increases marginally. This is because the number of tasks that become ready to run following a timeout queue update increases when the clock rate decreases. By contrast, schemes that move the timeout queue maintenance to the hardware are unaffected by varying the system clock rate or the workload.

We measured the number of interrupts seen by the operating system configurations for workloads with large periods. The configurations that we consider are the preemptive version of μC/OS with a regular timer interrupt and the preemptive version of μC/OS that uses the CCAM in auto decrement mode. In the latter, interrupts are generated only when a task becomes available to run following the automatic hardware based timeout queue update. Using the CCAM reduces the number of interrupts by more than 90% for all workloads and system clock rates. The number of interrupts increases marginally with the workload. The increase is due to the number of scheduling points increasing with workload. The number of interrupts does not change with the system clock rate. This is in contrast with the configuration using a regular timer interrupt where the number of interrupts increases proportionally with the clock rate.

5.3 Jitter

We represent the jitter value for the system running g721 encode and adpcm decode applications. These tasks are representative of two types of workloads. The first is a relatively light workload with each job not taking more than 200 us to execute i.e. it completes within a single timer tick. The latter is representative of compute intensive workloads which comprises of jobs which require more than a single timer tick interval to execute to completely. We note that in most cases jitter occurs in an early - late pair. If a job runs late i.e. has a positive jitter, then the scheduler attempts to bring it back in step. This causes the inter-task period to be less than that of the designated period or the task to have negative jitter. We concern ourselves only with the absolute jitter in terms of a percentage of the period of the application.The data is represented in terms of a discreet probability distribution function. For a given workload size the probability of a certain jitter value is proportional to the size of the circle or symbol at that point. The graphs plot jitter on the y-axis and workload size on the x-axis.

5.3.1 Preemptive μCOS Configurations

We consider the jitter initially for preemptive operating system configurations running G721 Encode benchmarks at 8 ms periods. Every 10 microseconds of jitter is grouped together for the purposes of representation. For a given configuration the reasons for jitter can be briefly summarized as follows:

Variations in execution time of the workload In the case of g721 Encode a few frames in the stream take approximately 10 microseconds longer to process. This results in an occasional early -late jitter of 10 us for every task in the system. This contributes to a jitter of around 125th of a percent for the g721 encode tasks.This jitter is considered as zero jitter for the purpose of representation.

Periodic background task This background task executes every 32 ms. When the release time of the tasks coincides with that of the background load, the output arrives late and it arrives early in the late next invocation. For a workload with tasks executing with 8 ms periods this causes a quarter of the invocations to execute late and another quarter to execute early. As the period is increased the number of such affected invocations also increases proportionally. For the original configuration of μCOS this can cause task executions to be delayed by approximately 80 microseconds or around 1% jitter.

Aperiodic interrupt Aperiodic interrupts occur with an approximate frequency of 100 Hz. In a preemptive system servicing this interrupt typically involves executing an ISR and a high priority response task. Servicing an interrupt thus has a noticeable overhead. This overhead can be as much as

40 - 50 microseconds or 0.5 - 0.6 % jitter for original configurations of μCOS. If an interrupt occurs after a task has been released but prior to its completion it will delay the arrival of this output.

Combinations of above phenomenon The remaining jitter points can be explained as a combination of the above factors. Interrupts and the periodic background task can combine to result in the workload having a jitter anywhere from 40 microseconds to 140 microseconds or 1.75%.When multiple interrupts are serviced this jitter can increase further to around 2%.

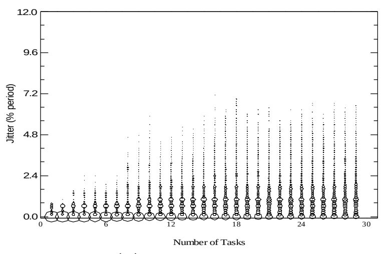
Using the CCAM decreases the overhead of operating system functions like scheduling, semaphore posting or pending etc. Thus the jitter due to the interference with other workload decreases by nearly 10 microseconds. Periodic task jitter is nearly 0.9% of the period while aperiodic interrupt jitter decreases to 0.4% The combined jitter due to the two phenomenon also decreases.

Operating System Overhead

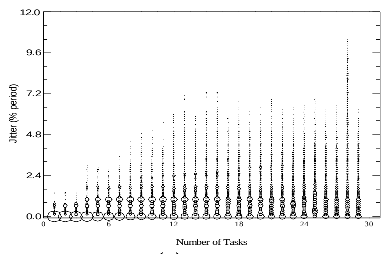
The RTOS performs periodic housekeeping activity involving time related functions like updating the timeout queue and the system clock. This is a regular periodic event and by itself will not affect the execution of periodic tasks. Its effect is seen only if the execution period of a task spans more than a single timer tick interval or the collective execution time of the

500 Hz System Clock

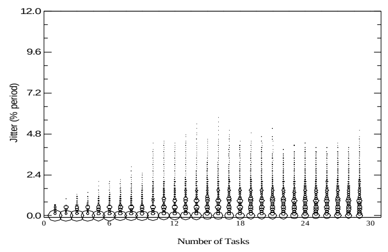
1 kHz System Clock



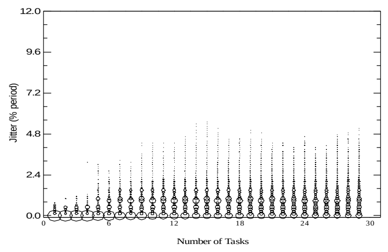
(a) μCOS



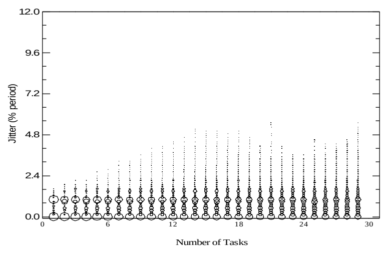
(e) μCOS



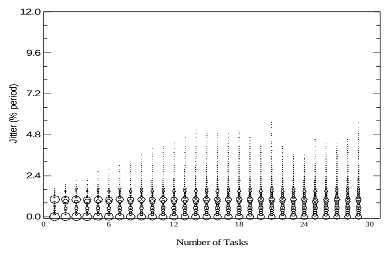
(b)μCOS with CCAM



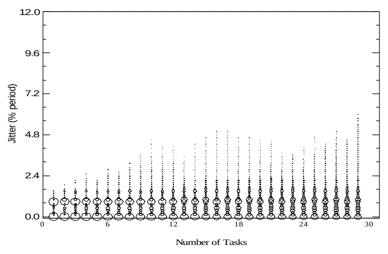
(f)μCOS with CCAM



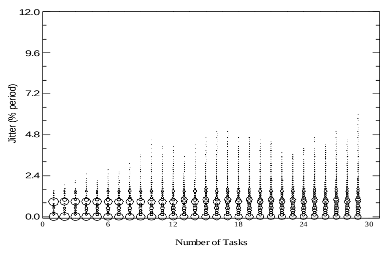
(c) μCOS with CCAM in Auto decrement mode



(g) μCOS with CCAM in Auto decrement mode



d) μCOS with CCAM with periodic task support



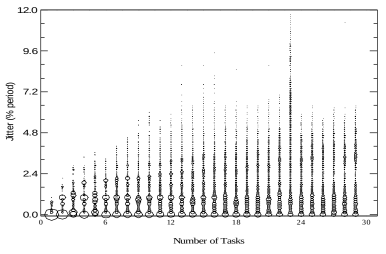
(h) μCOS with CCAM with periodic task support

Fig. 5.10. Jitter for Preemptive μCOS configurations. The graphs show the variation in jitter for the g721 encode benchmark executing with a period of 8 ms. Graphs (a)-(d) are for systems using system clock rates of 500Hz while graphs (e)-(h) are for configurations using 1kHz clocks. The graphs plot absolute jitter as a percent of the application period against the workload size. Data is represented in the form of a probability distribution plot with the size of the data point being proportional to the probability value at that point.

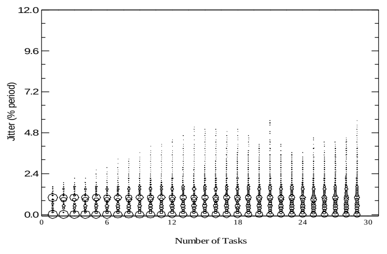
tasks in the system is greater than or approximately equal to a single timer tick interval. Even then a infrequently executing high priority task has to execute and push the servicing of the application task to the next timer tick interval. The servicing of this high priority background load will result in the timer tick interrupt arriving prior to the completion of the application task. The task resumes servicing after the timer interrupt service routine has completed and any other high priority task released during the timeout queue update has also been completed. The handler overhead contributes to the jitter. The timer interrupt handler execution time scales with the work-load from 40 microseconds for 4 tasks to 110 microseconds from 16 tasks. The jitter due to the timer interrupt ISR alone scales from 0.5% of the period to nearly 1.3%. As the timer tick frequency increases we observe that the execution window available to tasks decreases further and they are more likely to spill over into the next timer interrupt interval. The result is that corresponding probability of the jitter increases as well. The jitter due to the interrupt handler is accompanied by the jitter due to the various other factors outlined above. This results in the jitter getting distributed across a wider region and having much larger values.

Systems which maintain their timeout queue configuration completely in software are subject to overload faster than those who do so in hardware. During overload the system drops lower priority tasks. These systems tend to have a a positive invalid rate i.e. missed tasks and dropped tasks. Tasks can be either dropped after partial execution or no execution at all or can be late. This infrequent execution causes the affected task as well as the other tasks in the system to have large amounts of jitter. This can be seen in the case of µCOS running with a 1kHz clock and a workload of g721 encode tasks with a period of 8 ms.When there are 28 tasks executing in the system it has a miss rate of around 2%. The infrequent execution of the low priority task causes the overall jitter of the system to be spread across a larger region than that for a system with a lower workload. Increasing the system clock resolution causes this overload to occur earlier. Doubling the clock rate to2 kHz results in the system results in an invalid rate of 2% at a workload of 24 tasks.

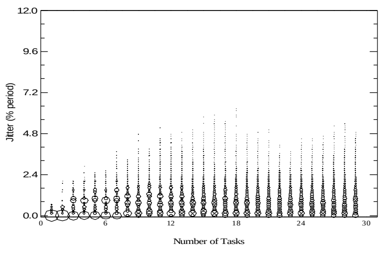
The jitter for a system with 29 tasks is less widely distributed than that with 28 tasks for the a clock rate of 2 kHz because the system tends to drop more tasks completely than execute them partially or late. The distribution



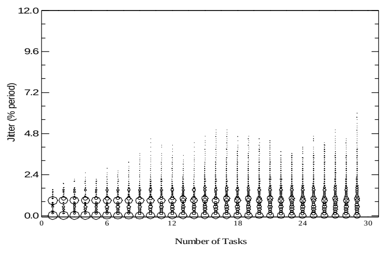
(a) μCOSμCOS



(c) μCOS with CCAM in AutoDecrement Mode



(b) μCOS with CCAM



(d) μCOS with CCAM+periodic feature

Fig. 5.11. Jitter for Preemptive μCOS configurations with a system clock rate of 2 kHz. The graphs show the variation in jitter for the g721 encode benchmark executing with a period of 8 ms. The graphs plot absolute jitter as a percent of the application period against the workload size. Data is represented in the form of a probability distribution plot with the size of the data point being proportional to the probability value at that point.

is thus more representative of a setup with 27 jobs and a possible 28th running very rarely.

We see that employing the CCAM allows the timeout queue management to be performed in hardware and increases the time available to the application to execute. Thus with a 1kHz clock the configurations using the CCAM are able to all comfortably execute 29 tasks and have near-zero invalid rates. Increasing the clock rate has next to no effect on the invalid rate or the jitter as can be seen in the graphs for μCOS with the CCAM.

As the timer interrupt overhead does not scale with the number of tasks the jitter due to this value also does not change with workload. This results in the maximum jitter percent dropping to under 5% when the CCA M is used. As the overhead of the RTOS interference is low increasing the timer tick frequency has a less visible effect in the case of the configurations using the CCAM. As earlier for the same workload an increase in the system clock resolution is accompanied by an increase in the number of tasks with non zero jitter.

We also studied the jitter for larger workloads like the ADPCM benchmarks. We observed that for the same clock rate and the same workload size the larger benchmark, adpcm decode, tends to have a more evenly distributed jitter. This is because as each individual task takes longer to complete its execution is more likely to coincide with the servicing of some other high priority background task like an interrupt etc. This causes the jitter values to be nearly evenly distributed between the maximum and minimum values possible. Increasing clock rate or using the CCAM has the same advantages as seen in the case of the smaller benchmark.

5.3.2 Non preemptive μC/OS Configurations

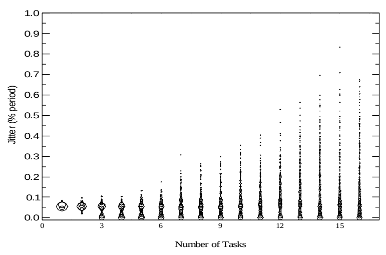
We study the jitter in a non preemptive configuration for a system executing the adpcm decode benchmark at a period of 144 ms.The workload consists of two tasks which have execution times that are longer than a single timer tick interval. The factors effecting the jitter in these systems is similar to those in the case of a preemptive configuration.

Variations in workload The adpcm decode benchmarks execution overhead can vary with each run by as much as 60 microseconds to as little as 10 microseconds. When there is more than one adpcm decode task executing the collective variation in execution time can result in jitters of the order of 300 microseconds or 0.2%. for a system with 7 tasks to nearly 600 us or 0.4% jitter for a system with 10 tasks.

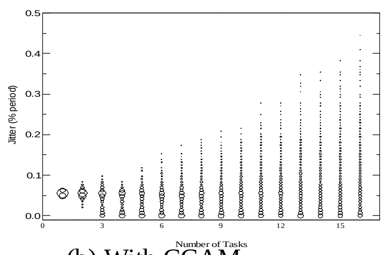
Periodic Background task This task executes with a period of 32 ms. For light workloads the task tends to interfere with every alternate execution of the tasks in the system. The task causes a jitter of around 80 microseconds on its own. As the workload increases the periodic task will interfere with more than one task in the workload. The task with which it interferes is not fixed. The result is that the effect of this interference on an individual task varies over time. As tasks execute back to back a single periodic task iteration may effect different number of tasks.

Polling of Aperiodic Interrupt Source The interrupt in this system is polled every 2 ms. The application workload consists of tasks which take more than 2 ms to execute. The result is that the polling task is executed between every task execution. Occasionally when the task completes exe-

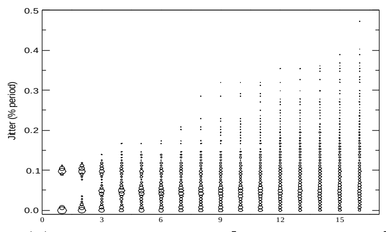
System Clock Rate 500 Hz



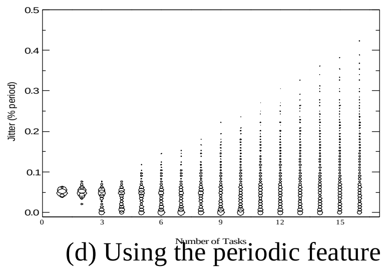
(a) Original



(b) With CCAM

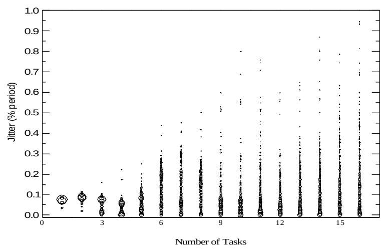


(c) CCAM in auto decrement mode

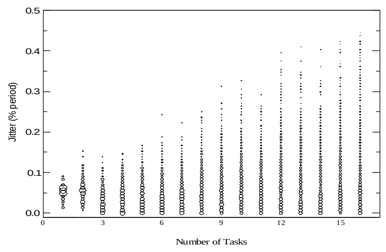


(d) Using the periodic feature

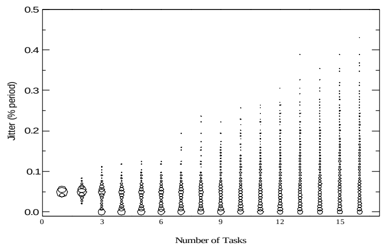
System Clock Rate 1 kHz



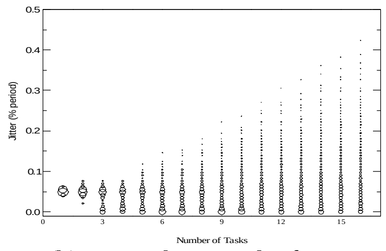
(e) Original



(f) With CCAM



(g) CCAM in auto decrement mode



(h) Using the periodic feature

Fig. 5.12. Jitter for nonpreemptive μCOS configurations . The graphs show the variation in jitter for the adpcm decode benchmark executing with a period of 144 ms. Graphs (a)-(d) are for systems using system clock rates of 500Hz while graphs (e)-(h) are for configurations using 1kHz clocks. The graphs plot absolute jitter as a percent of the application period against the workload size. Data is represented in the form of a probability distribution plot with the size of the data point being proportional to the probability value at that point.

cution before a polling period has elapsed another read input or write output task may execute before the polling task. This causes a jitter of 8 microseconds or a jitter of less than 0.1%.

Combinations of the above phenomenon The jitter of the various workload based factors is amplified when they occur together. The cumulative effect of these factors can result in jitter as high as nearly 1% of the period.

Operating Systems Overhead Non preemptive configurations have less operating system overhead as they tend to do away with critical section protection and eliminate race conditions. Additionally timeout queue updates are performed only either when a task completes and an update needs to be performed or when the system is idling and an update needs to be performed. Tasks are not halted in order to perform this operation and hence this operation does not interfere as much as in the case of the preemptive case. Occasionally during the execution of a hyper period of the tasks the timeout queue update may not occur at the same points in an earlier hyper period and it can contribute to a jitter depending on the size of the workload - 40 - 120 microseconds or less than 0.1% jitter. All this allows the non preemptive system to execute more tasks than a comparable preemptive configuration. As in the preemptive case this jitter occurs in combination with other interference jitter and is more considerable. Using the CCAM in non preemptive μCOS configurations has the same benefits as in the preemptive case. The maximum jitter values are decreased as operations are performed faster. But the gains in terms of decreased jitter is minor compared to those achieved by switching from a preemptive configuration to a non preemptive one.In the case of a non preemptive configuration using the CCAM it reduces the maximum jitter from 1% to nearly 0.5%.

5.3.3 Echidna

Tasks executing in echidna experience low jitter for light and moderate workload sizes. The benefit of using a high resolution timer is seen in this system. Jitter is of the order of 100 us which is less than 1% in the case of a workload consisting of g721 encode tasks executing with a period of 8 ms.The aperiodic interrupt polling task executes every 4 ms. Every time an interrupt is serviced there is a jitter of around 10 microseconds or 0.0125% jitter. The periodic task which occurs every 32 ms interferes with every 4th task and results in a jitter of just under 1%.

The system tends to overload rapidly. The original configuration of echidna cannot support more than 14 g721 encode task pairs running with 8 ms periods. This is because the EDF scheduler of echidna selects to execute late tasks rather than execute newly released tasks which haven't

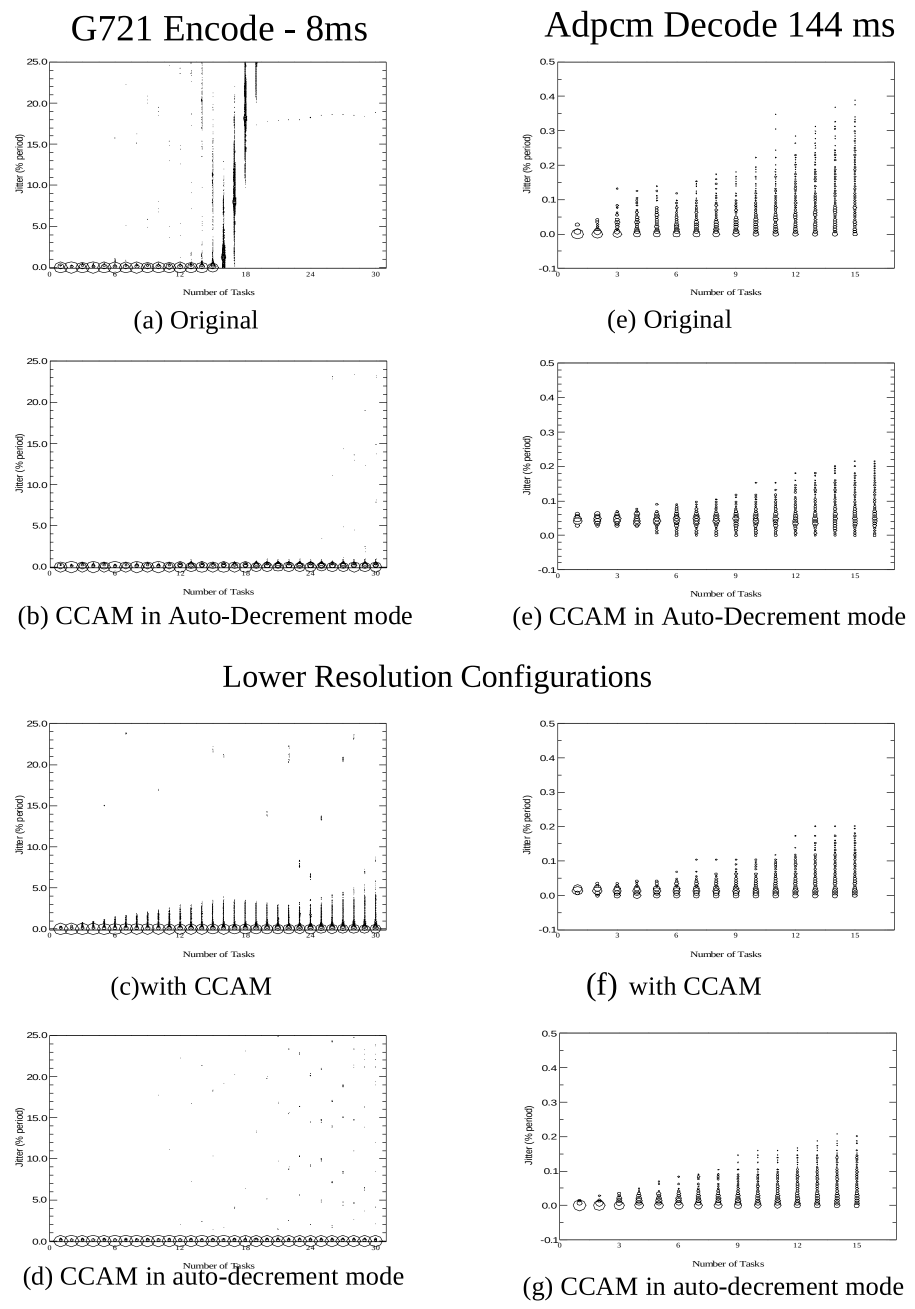


Fig. 5.13. Jitter for Echidna based configurations. The graphs show the variation in jitter for the two different benchmarks. Graphs (a)-(d) are for g721 encode applications with an 8 ms period and (e) - (g) are for adpcm decode with 144 ms period. The graphs in the top half all have high system clock resolutions on the order of 10kHz while the bottom graphs have resolutions on the order of 1 kHz. The graphs plot absolute jitter as a percent of the application period against the workload size. Data is represented in the form of a probability distribution plot with the size of the data point being proportional to the probability value at that point.

missed their deadlines. Thus when the system overloads the situation tends to deteriorate rapidly and it exhibits unpredictable behavior. Additionally the echidna scheduler has a higher overhead because it has to constantly poll the time and check if the task at the head of the queue is ready to go and for adding tasks to the various system queues.

All schemes which employ the CCAM in auto decrement mode whether they are high or low resolution give low jitter values for all workload sizes.They do not get overloaded as rapidly as well. This is because they eliminate the RTOS O(n) operations and replace them with O(1) operations with low overheads in terms of execution time.

Configurations executing adpcm decode benchmarks at 144 ms have higher jitter. The variation in the workload size is considerable and can result in jitter in the order of several percent for larger workloads. In addition these tasks take longer to execute and a polling task may miss its deadline as a consequence. Echidna not only executes late tasks but also does not drop any iteration of a late task. It executes the task till all “dropped iterations” have been taken care of. As a result the polling task is executed repeatedly resulting in additional jitter. Using the CCAM does not eliminate this behavior. It instead lowers the jitter by reducing the overhead of the operations involved in placing it on the queues etc.

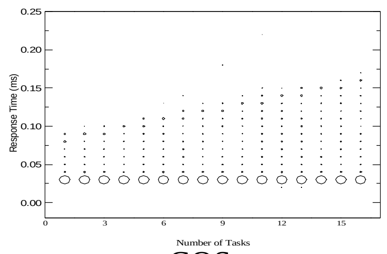
5.4 Response Time

5.4.1 Preemptive μCOS Configurations

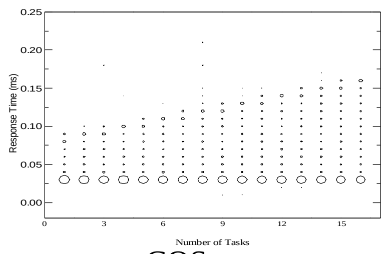
In a preemptive system when the interrupt occurs the system responds by executing the associated ISR - Interrupt Service Routine. The ISR performs two functions. The first of these is to acknowledge the interrupt. The second is to schedule a high priority aperiodic task which executes the non-critical portion of the interrupt handling routine. The best case interrupt latency can be as low as 36 us for μCOS. For preemptive configurations using the CCAM the minimum latency achieved is 27 us. This is because the CCAM enables scheduling and semaphore posting to be accomplished much faster.

The latency of the response depends among other things on the priority of the response task relative to that of the other ready tasks in the system. As the interrupt service task has the second highest priority among the tasks in the system it often ends up being the highest priority task available for execution resulting in a greater than 80% probability of an interrupt being serviced with minimum latency.

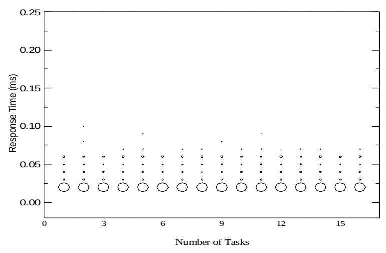
The worst case latency is seen when the aperiodic interrupt occurs when the timer interrupt is being serviced. Every timer interrupt, μCOS performs timeout queue maintenance by walking through the entire task list and updating the status of each individual task. During this timeout queue



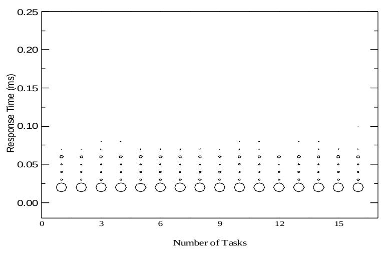
(a) μCOS Original



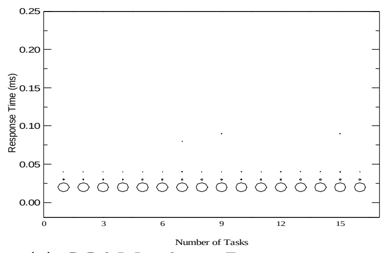
(b) μCOS Original



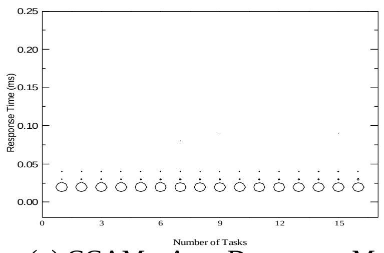
(b) With CCAM



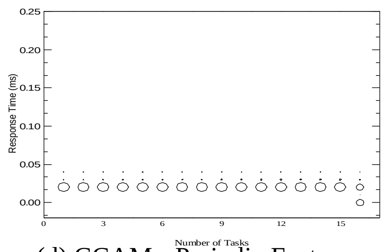
(f) With CCAM



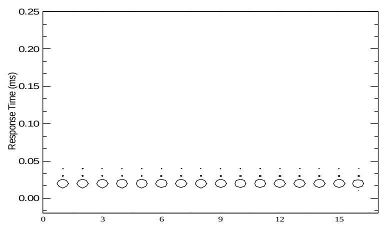
(c) CCAM - Auto-Decrement mode



(g) CCAM - Auto Decrement Mode

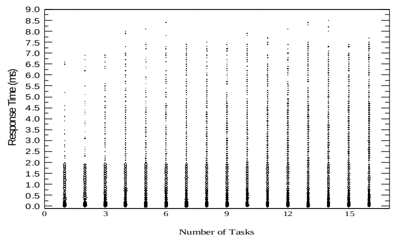


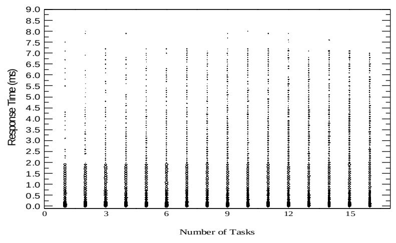
(d) CCAM - Periodic Feature

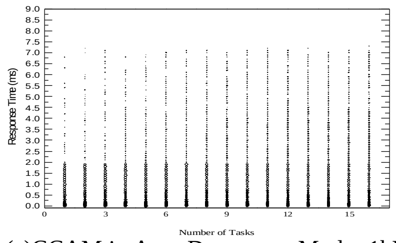


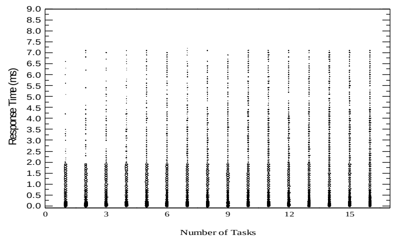
(d) CCAM - Periodic Feature

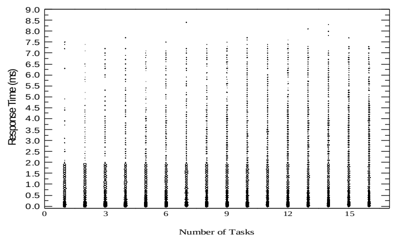
Fig. 5.14. Interrupt Response Time for Preemptive μCOS configurations. The graphs show the variation in interrupt response time with variation in workload size and system clock rates. Graphs (a) -(d) are for configurations with a system clock of 1 kHz while graphs (e) - (g) are for systems with clock rates of 2 kHz. All systems are executing the adpcm encode benchmark with a period of 240 ms.The graphs plot response time against the workload size. Data is represented in the form of a probability distribution plot with the size of the data point being proportional to the probability value at that point.


(a) Non preemptive μCOS - 1kHz clock


(b) CCAM- 1kHz clock


(c)CCAM in Auto Decrement Mode- 1kHz clock


(d) CCAM in Auto Decrement Mode with Periodic Feature- 1kHz


(e)Non preemptive μCOS - 2kHz clock

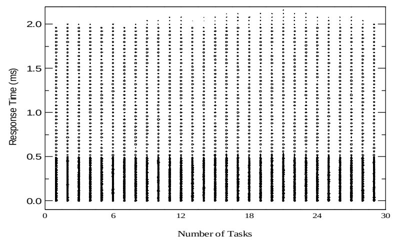

(f) Non preemptive μCOS - 1kHz clock

Fig. 5.15. Response Time to Aperiodic Interrupts for Non preemptive μCOS configurations. Data in graphs a-e are for the adpcm encode executing with a period of 240 ms while graph f is for the g721 encode benchmark executing with a period of 16 ms. Graphs (a)-(d) and (f) are for configurations with system clock rates of 1 kHz while graph e has a system clock rate of 2 kHz..The graphs plot response time against the workload size. Data is represented in the form of a probability distribution plot with the size of the data point being proportional to the probability value at that point.

update scheduling is locked. Thus even though the actual ISR gets executed, the high priority response task which it releases cannot be scheduled till the RTOS completes updating the timeout queue. The CCAM helps reduce the timeout queue maintenance overhead by providing a single instruction the “tick”. This helps reduce the overhead of the timer interrupt by nearly 40 to 80% and in turn the worst case latency to as much as 50 microseconds.In the case of configurations which use the CCAM in auto decrement mode the worst case interrupt latency is further reduced because the high priority interrupt now performs only a scheduling operation which has a lower overhead of approximately 10 microseconds.

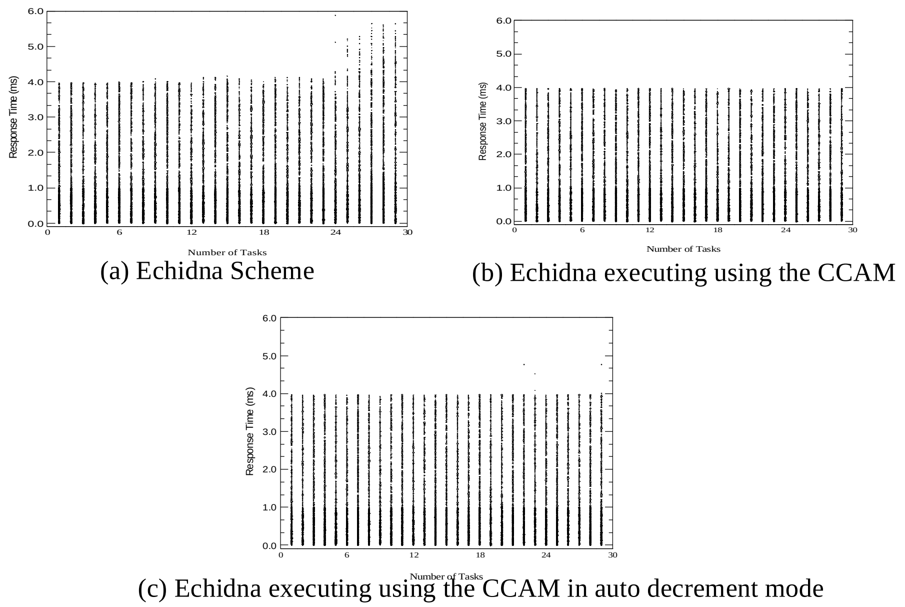
Further as the number of tasks in the system increases the overhead of timeout queue maintenance increases. This is because the timeout queue in μCOS contains all the tasks in the system. Thus we see that the worst case latency for the original configuration of μCOS increases proportionally with the number of tasks in the system. For configurations which have moved their timeout queue maintenance to hardware the worst case latency is independent of the number of tasks in the system.

We note that increasing the timer tick frequency results in a corresponding increase in the probability that a interrupt servicing may be delayed by a timeout queue servicing. In the case of systems using the auto decrement mode there is no alteration in the distribution. This is because the interrupt occurs at scheduling points which still remain the same.

From figure we see that nearly 20% of the data is uniformly distributed between the minimum and maximum latencies possible. There is a small concentration of data around the 90 microsecond region across all configurations. This point represents the interrupts which occurred when the system was idling in a low power mode. The arrival of the aperiodic interrupt causes the system to switch out of its low power mode. Switching out of the low power mode takes nearly 40 us. This switching overhead combined with the latency of servicing the interrupt itself results in a latency in the order of 90 microseconds.

5.4.2 Non preemptive μCOS Configurations

In the case of all non-preemptive systems, the interrupt is polled every 2 ms. When the polling task detects that an interrupt has been generated it acknowledges the same and immediately executes the response task. The minimum response time in non preemptive systems is thus on the order of 10 microseconds. In non preemptive systems the interrupt polling task may often end up running late because it has to wait for another task to complete. Thus the response time depends on not just the period of the polling task but also on the execution time of the tasks in the system. The worst case response time in a system with tasks with low execution overheads is the polling task period e.g. systems executing G721 Decode/Encode tasks as seen in Figure 5.15 f. On the other hand tasks running ADPCM Encode (a larger task) can have worse case interrupt service times on the order of



(a) Echidna Scheme

(b) Echidna executing using the CCAM

(c) Echidna executing using the CCAM in auto decrement mode

Fig. 5.16. Response Time to Interrupt for Echidna based configurations. The graphs show the variation in the response time to the aperiodic interrupt for Echidna based systems running the g721 encode benchmark at 16 ms. The x-axis plots increasing workload sizes while the y-axis represents the response time in ms. The data is represented in the form of a probability distribution function with the size of the circle being proportional to an interrupt being serviced at that instance.

execution time of the task, in this case roughly 8 ms. As the interrupts can occur randomly we see that the interrupt response time has a uniform probability distribution.

Using the CCAM does not alter the nature of this distribution as it is workload dependent and not operating system dependent unlike the preemptive configuration

We observe that altering the frequency of the system clock in a non preemptive system has no affect on the response time as can be seen in figure 5.15. The distribution is virtually identical to that in Figure 5.15e.

5.4.3 Echidna

In Echidna the interrupt polling task period has been increased to 4 ms. This results in the interrupt response time being uniformly distributed in the range of 0 to 4 ms as can be seen in fig 5.15. The distribution spreads for workloads containing more than 25 task pairs. This is because the system is overloaded and the polling task executes later than usual which results in the further spread. The configurations with the CCAM do not exhibit this behavior because they are not overloaded at these workloads.

Using the CCAM for echidna has the same affect on response time as in the case of nonpreemptive μCOS. It only improves the response time at certain workload levels. This is because the using the CCAM reduces the RTOS overhead and thus increases the maximum workload size the system can support.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusions

The CCAM effectively reduces RTOS overhead for operating systems which employ it. The gains in terms of energy vary with µC/OS based pre-emptive configurations from 20 to 40% of the original and are around 40% in the case of non preemptive configurations. In the case of Echidna using the CCAM with the same clock resolution as the original configuration results in an increase in the energy consumption by 30%. For lower clock resolutions though the gains are of the same order. Utilization overhead decreases in all cases. The kernel utilization µC/OS decreases from 50% to 90% depending on the CCAM configuration employed.

The overall jitter in the system decreases because of lowering the kernel utilization. The system is also able to run more tasks. In the case of echidna the number of supported tasks increases by a factor of two. Interrupt response time is improved in only the preemptive configuration of µC/OS. It is lowered to nearly 25% of the original value.

6.2 Future Work

The CCAM currently does not provide an effective solution to handling priority sorts for systems with tasks with multiple priority levels without introducing some sort of non-determinism. The model should move from its unique priority level implementation level.

Another possible area of exploration is the usage of alternate time representations which would lower the overhead of the timeout queue update in the case of high resolution clock systems.

Finally the CCAM energy and execution model should be further validated. This can be developing a hardware prototype to confirm timing and possibly energy measurements as well. Additionally porting operating systems which use different scheduling approaches would help in testing the generic nature of the device.

BIBLIOGRAPHY

[1] T. Coopee. “Embedded Intelligence.” InfoWorld, November 17, 2000.

[2] 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.

[3] Arnold Berger, *Embedded Systems Design An Introduction to Processes, Tools and Techniques*

[4] Evans Data Corporation. Embedded Systems Developer Survey, Volume 1 2002

[5] Martin Hiller , Software Fault Tolerance Techniques from a Real-Time Sstems Point of View - an overview.

[6] Lock,C.D , Best Effort decision maing for real-time scheduling PhD Thesis, CMU-CS-86-134, Department of Computer Science , Carnegie-Mellon University , May 1986

[7] Clark, R.K Scheduling dependent real-time activities PhD Thesis CMU-CS-90-155 Department of Computer Science Carnegie Mellon University August 1990

[8] H.Kopetz and W.Ochsenreiter, Clock Synchronization in Distributed-Real-Time Systems, IEEE Transactions on Computers,C-36 (8), Aug. 1987, pp. 933-939.

[9] Yangmin Seo and Jungkeun Park and Seongsoo Hong,Efficient User-Level I/O in the ARX Real-Time Operating System,Lecture Notes in Computer Science, vol 1474, 1998

[10]Philip J Koopman, Embedded System Design Issues (the Rest of the Story) , *Proceedings of the International Conference on Computer Design* , Austin, October 7-9 1996.

[11]J. Turley. “AMD pushes Elan to 100 MHz.” *Microprocessor Report*, vol. 11, no. 14, pp. 10, October 1997.

[12]J. Turley. “IBM matches Motorola embedded PowerPC.” *Microprocessor Report*, vol. 11, no. 17, pp. 10, December 1997.

[13]J. Turley. “ARM7, ARM9 cores to include cache.” *Microprocessor Report*, vol. 12, no. 5, pp. 10, April 1998.

[14]J. Turley. “MMC2001 launches M.Core odyssey.” *Microprocessor Report*, vol. 12, no. 4, pp. 13, March 1998.

[15]J. Turley. “NEC VR5400 makes media debut: NEC/Sandcraft project produces high-end embedded processor.” *Microprocessor Report*, vol. 12, no. 3, pp. 1–8, March 1998.

[16]C. P. Feigel. “Zilog introduces 32-bit Z80 to U.S.” *Microprocessor Report*, vol. 8, no. 9, pp. 1–8, July 1994.

[17]J. Turley. “68HC11 grows up to 16 bits: Motorola’s 68HC12 line boosts performance up to 10 times.” *Microprocessor Report*, vol. 10, no. 7, pp. 1–9, May 1996.

[18]Manfred Schlett, Trends in EmbeddedMicroprocessor Design , *Microprocessor Report*

[19]M. B. Jones and J. Regehr. “CPU reservations and time constraints: Implementation experience on windows NT.” In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle WA, July 1999, pp. 93–102.

[20]K. Kailas and A. Agarwala. “An accurate time-management unit for real time processors.” In *Proc. of 8th International Conference on Advance Computing and Communications (ADCOM’00)*, 2000

[21]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.

[22]Alia K. Atlas and Azer Bestavros, "Statistical Rate Monotonic Scheduling", *19th IEEE Real-Time Systems Symposium,* Madrid, Spain, December 1998

[23]J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. R&D Books (Miller Freeman, Inc.), Lawrence KS, 1999.

[24]C. Lee, M. Potkonjak, and W. Mangione-Smith. “MediaBench: A tool for evaluating and synthesizing multimedia and communications systems.” In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO’97)*, Research Triangle Park NC, December 1997, pp. 330–335.

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

[26]S. A. Maxwell. *Linux Core Kernel Commentary*.

[27]J. Mulder, N. Quach, and M. Flynn. “An Area Model for On-Chip Memories and its Application.” *IEEE Journal of SOLID-STATE Circuits*, Vol. 26, No. 2, February 1991, pp. 98-106.

[28]http://www.dedicated-systems.com/encyc/publications/

[29]John A. Stankovic, and Krithi Ramamritham.What is Predictability for Real-Time Systems?,UM-CS-1990-062, July, 1990

[30]http://www.dependability.org/wg10.4/

[31]Stefan Savage and Hideyuki Tokuda,Real Time - Mach Timers: Exporting Time to the User, *USENIX MACH III Symposium,* Apr 1993

[32]J. Adomat, J. Furunas, L. Lindh, and J. Starner. “Realtime kernel in hardware RTU: A step towards deterministic and high-performance real-time systems,” 1996.

[33]J. Furuns, J. Adomat, L. Lindh, J. Strner, and P. Vrs. “A Prototype for Interprocess Communication Support.” In *Proc. Hardware, 9th Euromicro Workshop on Real-Time Systems.* June, 1997.

[34]L. Lindh, J. Strner, J. Furuns, and J. Adomat. “Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems.” In *Proc. Seventh Swedish Workshop on Computer Systems Architecture*. June, 1998.

[35]T. Klevin and L. Lindh. “Scalable architectures for real-time applications and use of bus-monitoring.” In *Proc. 6th Int' Conf. on Real-Time Computing Systems and Applications (RTCSA'99)*. December, 1999.

[36]http://www.embedded.com/advert/update.htm

[37]D. B. Stewart, R. A. Volpe, and P. K. Khosla. “Design of dynamically reconfigurable real-time software using port-based objects.” *IEEE Transactions on Software Engineering*, vol. 23, no. 12, pp. 759–776, December 1997.

[38]Embedded Research Solutions. Embedded Zone — Publications. http://www.embedded-zone.com, 2000.