# ABSTRACT

Title of Thesis: AN EVALUATION OF EMBEDDED SYSTEM BEHAVIOR USING FULL-SYSTEM SOFTWARE EMULATION

Degree Candidate: Christopher Michael Collins

Degree and Year: Master of Science, 2000

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

With embedded processor technology moving towards faster and smaller processors and systems on a chip, it becomes increasingly difficult to accurately evaluate real-time performance. This research describes an evaluation method using an embedded architecture software emulator that models the Motorola M-CORE processor architecture. This emulator is used to evaluate and compare the real-time performance of a public-domain experimental Real-Time Operating System (RTOS) against a bare-bones multi-rate task scheduler. The results of the experiment, as shown in arrival time JITTER, response-time DELAY, and CPU BREAKDOWN figures, show the trade-offs between job load, job frequency, and kernel overhead. This research suggests full-system software emulation to be a valid method of evaluating embedded systems' behavior and real-time performance.

# AN EVALUATION OF EMBEDDED SYSTEM BEHAVIOR

# USING FULL-SYSTEM SOFTWARE EMULATION

by

Christopher Michael Collins

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
2000

Advisory Committee:

Professor Bruce L. Jacob, Chair
Professor Shuvra S. Bhattacharyya
Professor Donald Yeung

# ACKNOWLEDGEMENTS

I would like to thank Eric Fiterman and Tiebing Zhang for their assistance in the research that led up to this report.  They both aided in the creation of the emulator and spent many hours debugging and running simulations.

I would like to thank Moussa Ba, Julian Requejo, Sujaya Srinivasan, and Dr. David B. Stewart in the SERTS lab for their assistance with their Echidna operating system and the Motorola M-CORE evaluation board.

I would like to thank Dr. Bruce L. Jacob for giving me the opportunity to work on this project and for many hours of assistance with the research itself in addition to the development of this report.

On a personal note, I would like to thank my parents, John and Joan M. Collins, for their support and understanding throughout this project.

Finally, I would like to thank Danesha R. Fitzgerald, for her support, for keeping me sane, and most importantly, for believing in me.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| ASIC | Application Specific Integrated Circuit |
| AP-IPC | Aperiodic Inter-Process Communication |
| CL | Control Loop |
| CPU | Central Processing Unit |
| ELF | Executable-Linking Format |
| EPC | Exception Program Counter |
| EPSR | Exception Program Status Register |
| EX | Execution Stage |
| EXWB | The boundary between the Execution and Write Back stages |
| FPC | Fast Interrupt Program Counter |
| FPSR | Fast Interrupt Program Status Register |
| ID | Instruction Decode Stage |
| IDEX | The boundary between the Instruction Decode and Execution Stages |
| IC | Integrated Circuit |
| IF | Instruction Fetch Stage |
| IFID | The boundary between the Instruction Fetch and Instruction Decode stages |
| IPC | Inter-Process Communication |
| IRAM | Intelligent RAM |
| FIR | Finite Impulse Response |
| MAIN | The boundary before the Instruction Fetch stage, the instruction about to enter the pipe |

| | |
|---|---|
| NOS | Non-Operating System |
| PC | Program Counter |
| PSR | Program Status Register |
| RAM | Random Access Memory |
| ROM | Read-Only Memory |
| RTOS | Real-Time Operating System |
| SOC | System on a Chip |
| VBR | Vector Base Register |
| WB | Write Back Stage |
| WBEnd | The boundary after the WB Stage, when the instruction leaves the pipe |

# Chapter 1: Introduction

With embedded processor technology moving towards faster and smaller processors and systems on a chip, it becomes increasingly difficult to accurately evaluate real-time performance. Probing a piece of silicon, or accurately measuring values approaching less than one nanosecond becomes more expensive and more difficult, if not impossible. It becomes necessary to find additional methods to evaluate and debug embedded systems.

## 1.1: Goal and Motivation

The goal of this research is to provide an additional method for evaluating and debugging embedded systems. This research presents a method of using full-system emulation to evaluate the real-time performance of an embedded system. An embedded architecture emulator was created, using the C programming language, that emulates the Motorola M-CORE embedded processor down to the register level and is accurate to within 100 cycles per million as compared to actual hardware. This work touches on several different aspects of embedded systems design, such as the testing and debugging of increasingly integrated systems, hardware/software codesign methodologies, and the evaluation of real-time systems.

One of the motivators of this research is that it is becoming increasingly difficult to evaluate system behavior at the hardware level. Apart from the unpleasantries of waiting for actual fabrication of the hardware, or the expense of such a task, it is sometimes difficult to obtain information from the actual hardware. Five, ten years ago it was easy enough to hook up a probe to the bus connecting the processor to the main memory or the connections between the processors and other pieces of the hardware. However, with the advent of systems on a chip and application-specific integrated circuits, it is no longer possible to obtain those signals, for they never leave the silicon [32, 45]. The only way to debug these systems is to either probe the silicon itself, or to add additional logic to the chip so that it brings the signal off the chip, and even that option is limited by the number of physical pins that can be put on a chip and spared for simple debug and evaluation purposes. Also, with the speeds that some of today's embedded processors are running, it becomes difficult to find a logic analyzer that can keep up with the processors, not to mention costing tens to hundreds of thousands of dollars [20, 29]. If there were another method to evaluate these systems early on, both valuable time and money could be saved.

One of the methodologies gaining wide acceptance in both the embedded world and the general purpose world is that of hardware/software codesign [24]. As opposed to the traditional methods of developing the hardware and software for a system separately, the hardware/software codesign methodology recognizes the benefits inherent in the designing of the two together, at the same time. The hardware being designed with the software needs in mind as well as the designing of the software with hardware limitations and issues in mind benefit the design both in performance and time to market, given that if hardware and software designers communicate during the design process, there is less chance of problems happening due to ignorance [9]. This research offers a method for the software engineer to test his software on a C emulator, something that he will understand, as opposed to handing the software off to a technician to go run it on the actual hardware, or for him to try to understand how to operate a VHDL model.

Real-Time Operating Systems are commonly used in the development, productizing, and deployment of embedded systems. Unlike the world of general purpose computing, embedded systems are usually developed for a limited number of tasks. Any facilities that these tasks might need are often built directly into the code and the feeling is often that real-time operating system would just add unnecessary overhead [13]; in many cases, any RTOS functionality needed is provided by a homegrown design. However, these “roll-your own” [13] pseudo operating systems that are created on the fly are not very portable and often times include additional work that could easily be accomplished by using one of today’s many commercial RTOSs. What is needed is a method to test both commercially available Real-Time Operating Systems and in-house creations on the target architecture to verify which would give the best behavior.

Many of the projects in the area of real-time systems concern themselves with the development of scheduling algorithms and the demonstration that those algorithms work [1,2,53,54]. However, as others have observed [28], “there currently exists a wide gap between real-time scheduling theory and the reality of RTOS implementation.” The majority of the work in this field is done through theoretical analysis testing the scheduler code at the block level, or running the raw scheduler code by itself. Very little of that analysis follows those scheduling algorithms all the way to the RTOS implementation, where other mechanisms like inter-process communication and semaphores interact in subtle ways to make the behavior of the algorithms less easily understood and therefore less predictable. The analysis of these scheduling algorithms should be accompanied with experimental evaluation of RTOSs on the actual hardware. Unfortunately, this sometimes presents a problem when the hardware is not available, or there are questions of money or time. However, if it was possible to run tests on an emulation of that hardware, that would save both time and money and allow this analysis to be complete.

The research effort going on currently that most resembles this work is the SimOS project going on at Stanford [44]. Like the emulator described in this research, SimOS is an execution driven simulator that is accurate enough to run a full operating system on top of it. The primary difference between the emulator developed for this research and SimOS is the target application domain. SimOS is focused on studying high performance machines, while the emulator created for this work is interested in evaluating the real-time performance of low power embedded processors.

## 1.2: Results

In this research, an embedded system emulator was built in C. A study of two Real-Time Operating Systems was run on that emulator. Echidna [10] is a publicly available RTOS based on Chimera [48]. NOS is a fixed-priority, multi-rate executive [27] based on descriptions of bare-bones RTOS given by designers in the industry [13].

This study provides information about both of the RTOSs that might lead to a decision among them as to which one to use. Predictably, as loads increased, the RTOSs hit their job deadlines until system loads were reached and missed those deadlines afterwards. Also predictably, as the system became overloaded in NOS, lower priority tasks were completely ignored. It is seen that RTOS overheads are extremely high when compared to low overhead tasks. In some cases, the RTOS can account for more than 90% of the processor's busy time. However, as the periodic task's complexities and CPU requirements grow, the proportion of the RTOS diminished significantly, to a point where the RTOS accounts for only 20-50% of the processor's busy time. Lastly, this study has shown that this method of using a full-system software emulator can be used as a valid method for the evaluation of embedded system behavior.

## 1.3: Overview of Report

Chapter 2, Background, describes the work that has been done in this field and areas that relate to this field of research. Chapter 3, The Emulator, gives a detailed description of the emulator, the steps that went into making it, and the methods used to verify it. Chapter 4, Real-Time Performance Evaluation, first describes the two different Real-Time Operating Systems that were run on the M-CORE Emulator, Echidna and NOS, describes the four benchmarks that were run on each of the real-time operating systems, describes the two types of background load run on the real-time operating systems, and describes the experiment. Chapter 5, Results and Analysis, displays the results from the experiment listed in Chapter 4, and analyzes the different results for the several benchmarks. Chapter 6, Conclusions, gives the conclusions drawn from the findings of this paper, and Chapter 7, Future Work, describes possible continuation of this work.

# Chapter 2: Background

This chapter offers a brief background into the areas that are related to the research performed in the report as well as the areas that support the reasons for performing this research. The first section takes a look at embedded systems, the issues and tools involved in their design, current trends, and how they can benefit from this research. The second section examines Hardware/Software Codesign, the methodologies that it has produced, and how those methodologies can benefit from this research. The third section gives an introduction to real-time operating systems and breaks down the issues involved in their creation and use. Section four discusses the evaluation of real-time systems, the methods used to evaluate those systems, the metrics used to characterize them, and the current studies going on in the field. In the final section of this chapter, SimOS, a full-system simulation very much like the emulator created in this research is described, and the studies that have been performed with it are listed, as well as how it differs from the emulator created in this research.

## 2.1: Embedded Systems

Embedded systems has become a buzz word in the last five years, but embedded systems and processors have been around for much longer than that [46]. One only needs to look around to see embedded systems everywhere: cell phones, alarm clocks, personal data assistants(PDAs), automobile subsystems such as ABS and cruise control, etc. This section takes a look at embedded systems, the issues and tools involved in their design, current trends, and how they can benefit from the research performed for this report.

### 2.1.1: Embedded Systems versus General Purpose Systems

An embedded system is usually classified as a system that has a set of predefined, specific functions to be performed and in which the resources are constrained [46]. Take for example, a digital wrist watch. It is an embedded system, and it has several readily apparent functions: keeping the time, perhaps several stopwatch functions, and an alarm. It also has several resource constraints. The processor that is operating the watch cannot be very large, or else no one would wear it. The power consumption must be minimal; only a small battery can be contained in that watch, and that battery should last almost as long as the watch itself. And finally, it must accurately display the time, consistently, for no one wants a watch that is inaccurate. Each embedded design satisfies its own set of functions and constraints. According to [46], there are an estimated 50,000 new embedded designs a year.

This is different from general purpose systems, such as the computer that sits on a desk in an office. The processor running that computer is termed a “general purpose” processor because it was designed to perform many different tasks well, as opposed to an embedded system, that has been built to perform a few specific tasks either very well or within very strict parameters.

### 2.1.2: Design Issues

As mentioned above, embedded systems are defined by their functions and their constraints. These constraints are almost as varied as the number of embedded systems themselves, but a few of the more prevalent ones are response time accuracy, size, power consumption, and cost [46]. All of these present the embedded system designer with some difficult decisions.

Response time is a critical factor in many embedded systems. Whether it is a specific time that an embedded system tasks needs to be run, like that of the alarm on an alarm clock; or the time between tasks that is important, like the system that delivers pain medication to a burn victim; all of these are time-critical issues. The most difficult task for an embedded system designer to do is to quantify these time deadlines, decide whether these deadlines are firm, and recognize what the consequences are if these deadlines are not met.

Size, as mentioned above, is also an important decision in many embedded systems. Many embedded systems designed today are bought and sold simply because they are smaller than the last implementation of that product. Take for example, the cellular phone. Today's cellular phones are half the size of the phones available two years ago, and those phones two years ago were smaller than the phones available before them. So if the manufacturer does not take into account size when designing his cell phone, he will most likely go out of business shortly after he produces a cell phone that is two to three times the size of all of his competitor's phones.

Another design issue concerning today's embedded system designers is that of power consumption. Continuing along the same line as the above mentioned size factor, many of these devices that are very small are handheld devices that are made to be mobile and thus must have a battery. Since the designer does not want the user to be forced to plug in or recharge the device every five minutes, the designer must make important choices in his design decisions and balance a feature’s merits against the power that the feature will consume.

A final consideration that embedded designers deal with is cost. Regardless of any choice of the above issues made, an embedded product is not going to sell if its cost is exorbitant. Most end users will sacrifice a small amount of performance, or a slightly less amount of battery time, for an embedded product that is less costly than all of its competitors. So just as with all of the above considerations, the designer must consider the cost of adding a particular modification to the design and whether or not the end user will be willing to pay that additional cost for that additional feature.

### 2.1.3: Development Tools

Embedded development tools have traditionally lagged behind tools for the development of general systems [46]. Unlike general systems, the design space for embedded systems is extremely large, so it is difficult to contain all of the facilities to specify, design, and test embedded systems.

However, now that embedded systems have garnered more interest in the research community as well as there being an increased need for those embedded systems, embedded systems tools are now catching up with regular system design tools, and they have become more readily available and diverse in their area of coverage [46]. Tools that were not available 5 to 10 years ago are now available as part of common EDA development suites. Also, tools are now available for the development of embedded system application software as well as the development of real-time operating systems.

### 2.1.4: Embedded System Trends

With the increase in interest and research of embedded systems have come a flood of new design trends. It is hard to envision that five years from now embedded systems will bear much resemblance to the systems today [46], other than their basic functionalities, and even those may be replaced in the future. Two of the trends currently hot in the embedded systems world that are discussed here are that of application specific integrated circuits (ASICs) and systems on a chip (SOC).

#### 2.1.4.1:Application Specific Integrated Circuits

The best way to define an application specific integrated circuit (ASIC) is to saying what it is not: an integrated circuit designed for multiple uses. Like the title suggests, this is a IC that has been designed for a specific application. Examples of ICs that are not ASICs are standard computer parts such as RAM, ROM, multiprocessors, etc. Examples of ICs that are ASICs are a chip designed for a toy robot or a chip designed to examine sun spots from a satellite [45].

The reason for mentioning this is that since ASICs are developed for a specific purpose, they are most likely constrained with both a tight budget and a short time to market. Any and all methods that might aid in the development of these chips would be welcomed with open arms in the industry.

#### 2.1.4.2:System On A Chip

System on a chip (SOC) is exactly what it sounds like. Hardware designers have taken the normally separate pieces of a complete system; the CPU, memory controller, main memory, I/O control, and the various buses and interconnects, and placed many or all of them on a single piece of silicon. This has the added benefits of size reduction, power reduction, cost reduction, and time delay reduction.

On of the more popular forms of SOC is that of Dave Patterson's Intelligent-RAM (IRAM) [20, 29]. IRAM is the combination of a processor on a chip with a large area of DRAM instead of or in addition to cache. This concept has several advantages. Like all forms of SOC, it reduces the number of chips in a system, allowing the product to be smaller and less expensive. IRAM addresses the key bottlenecks in many systems: memory bandwidth and memory latency. Memory bandwidth on IRAM is four times as wide as that on traditional systems, and memory latency is considerably less than that of traditional systems since the signals do not need to cross a pin barrier that can have a maximum number of pins.

However, there are also several inherent difficulties with SOC and IRAM. One is that there is only so much area on a chip, and this limits what you can put on it. It puts upper limits on the amount of main memory that you can have with a system, unless you still want to rely on going off-chip occasionally for information. Another large problem is that the design team creating the system on a chip must contain all of the knowledge to create a processor, a main memory, a I/O controller, and optimize all of them together [3, 32].

### 2.1.5: The Emulator's Benefit to Embedded System Design

One of the problems with embedded systems, and more specifically ASICs and SOCs, is that it is no longer possible to obtain debug information that was readily available in systems with discrete components. Those signals that are contained only in the silicon, such as information across the memory bus, never leave the silicon. The only way to debug them is to either probe the silicon itself, or to add additional logic to the chip that brings the signal off the chip, and even that option is limited by the number of physical pins that can be put on a chip and spared for simple debug and evaluation purposes. Also, with the speeds that some of today's embedded processors are running, it becomes difficult to find a logic analyzer that can keep up with the processors, not to mention costing tens to hundreds of thousands of dollars [20, 29]. If there were another method to test these system, both valuable time and money could be saved. The emulator that has been designed for this report could be used as an additional method to test those systems without incurring additional time and cost.

## 2.2: Hardware/Software Codesign

One of the methodologies gaining wide acceptance in both the embedded world and the general purpose world is that of Hardware/Software codesign [24]. This section first defines the concept and then the methodology of Hardware/Software codesign. Then a slightly different method of codesign is described. This section is concluded with how Hardware/Software Codesign can benefit from the emulator developed in this research.

### 2.2.1: Hardware/Software Codesign: The Concept

For years, designers have partitioned systems into hardware and software components that were developed separately [16]. When this is done, the hardware designers usually make architectural choices early in the design process. These decisions are based on their knowledge of the hardware requirements and their limited knowledge and understanding of the software requirements. And they are usually hard pressed to go back and make changes to these choices [18]. The result is that often the software designers are forced to make up for problems in the hardware through additional work of the software, often leading to a less than optimal overall design of the system.

The concept of Hardware/Software Codesign is that of both hardware and software designers work together to develop a system, whether that system be an embedded one, a general purpose one, or high performance one [9,24]. From specification of the requirements to exploration of the design space, and from development of the physical design to the simulation and test of the final product, hardware and software designers work cooperatively, concurrently, and most importantly, they communicate [9].

### 2.2.2: Hardware/Software Codesign: The Methodology

In response to these problems listed above, designers as well as EDA tool manufacturers are moving towards a design methodology that has hardware and software engineers working together from the beginning of the specification phase all the way through simulation and test [12]. In hardware/software codesign, designers from both disciplines integrate their work. The process begins with a functional exploration of the project that they are undertaking. The designers define requirements and create a working specification. Then the hardware and software designers work together to map this specification on hardware and software architectures. The designers then implement these architectures onto silicon and code and come back together to simulate and test. The entire process benefits from open communication from both sides [11,12].

### 2.2.3: Model Based Codesign

Another popular method of hardware/software codesign that is gaining greater acceptance is that of model based codesign. Model based codesign includes all of the above steps, but all of the work is done using mathematical models on a computer. This gives the added benefit of being able to run the above process multiple times, i.e. iterate on the design. Each time the process runs with slight modifications, and through many of these simulations, the optimal system is found. The benefit of this methodology is that the designer does not have to wait while a physical design is being created or suffer the cost of implementing that design. Often times this model based codesign is automated, leaving the designers even more time to perform other tasks. However, the down-side of model based codesign is that it is a mathematical representation of the real world: many mathematical representations are only approximate [43].

### 2.2.4: The Emulator's Benefit to Hardware/Software Codesign

As mentioned above, one of the most difficult tasks for engineers to do is to bridge the gap of knowledge between hardware and software designers. The research that we are performing offers an aid for the software engineer in that he can test his software on a C emulator, something that he will understand, as opposed to handing the software off to a technician to go run it on the actual hardware, or for them to try to understand how to operate a VHDL model. This research offers the hardware engineer a tool that allows him to quickly evaluate architectural changes without having to re-fab the microprocessor or other pieces of the system's hardware.

## 2.3: Real-Time Operating Systems

Real-Time Operating Systems (RTOS) are commonly used in the development, productizing, and deployment of embedded systems. Unlike the world of general purpose computing, real-time systems are usually developed for a limited number of tasks and have different requirements of their operating systems [5,14,15]. This section first gives the requirements of real-time operating systems, then breaks down the internals of

RTOSs and explains them in detail. This section concludes with how the emulator developed in this research would aid in the evaluation of RTOSs.

### 2.3.1: Real-Time Operating Systems: The Requirements

According to Liu [34], a good RTOS not only offers efficient mechanisms and services to carry out real-time scheduling and resource management but also keeps its own time and resource consumption predictable and accountable. A RTOS is responsible for offering the following facilities to the user programs that will run on top of it. The first responsibility is that of scheduling: a RTOS needs to offer the user a method to schedule his tasks. The second responsibility is that of timing maintenance: the RTOS needs to be responsible in both providing and maintaining an accurate timing method. The third responsibility is to offer user tasks the ability to perform system calls: the RTOS offers facilities to perform certain tasks that the user would normally have to program himself, but the RTOS has them included in its library, and these system calls have been optimized for the hardware system that the RTOS is running on. The last thing that the RTOS needs to provide is a method of dealing with interrupts: the RTOS needs to offer a mechanism for handling interrupts efficiently, in a timely manner, and with an upper bound on the time it takes to service those interrupts [34, 46].

There are several concepts that need to be defined in any discussion of RTOSs. The first concept is that of preemption. Real-time operating systems are either preemptive or non-preemptive. If a real-time operating system is preemptive, it means that a task currently being run by the RTOS can be interrupted by another task with a higher priority or an external interrupt. The interrupted task's state is saved, and this state will be restored when it is run again, allowing it to continue along from the same point that it was interrupted.  RTOSs that are nonpreemptive cannot be interrupted.  If a task is currently running when a second task needs to run, that second task must wait for the first task to finish running before it can begin to run [46].

Another important concept is that of hard real-time versus soft real-time.  Hard real-time means that a task must always be completed by a specific time.  The integrity of the system designed with hard real-time tasks will be compromised if such a deadline is missed.  An example of this is  the communication mechanism from the cockpit of a commercial airliner to the embedded system controlling the wing flaps.  If a pilot is coming in for landing, and pulls up on his flaps to slow his descent, that communication must work — for if it doesn't, the entire plane has the possibility of crashing.  Soft real-time systems are any type of system that is not a hard real-time system, meaning that if a task is late, the system will continue to keep running.  An example of this is an Automated Teller Machine (ATM).  If the software running upon the ATM takes a little longer to process a request, other than the costumer being slightly upset, the system will be able to perform its tasks, albeit late [46].

There are several different types of task scheduling for today's real-time operating systems to choose from.  There is the endless loop scheduler, that is basically a while(1) loop that continuously runs a piece of code.  Activities within the loop are executed in sequence and as many times as possible.  The next level of task scheduling is that of the basic cyclic executive scheduler.  In a basic cyclic scheduling algorithm, the idea of the endless loop is extended in that designers can separate the code to be executed into separate tasks.  These tasks execute in a standard sequence in an infinitely repeating loop.  This type of scheduling is often called round-robin scheduling.  Like the endless loop, all of the tasks run as often as possible. Time driven cyclic scheduling, the next level of task scheduling, differs from basic cyclic in that instead of running each one of these tasks as often as possible, it introduces the idea of a time interrupt. In this scheduler, one hardware timer is used to wake up all tasks. This timer wakes up the first task in line, and as soon as that first task is finished, the next task runs. All of the tasks in line must finish before the next timer interrupt. Following the time driven cyclic scheduler is the multi-rate cyclic executive scheduler. This is an expansion of the time driven cyclic scheduler in that it allows multiple periods, so long as higher frequency tasks are a multiple of the base task's frequency. This is done by inserting a task more than one time into the chain or into multiple chains. The multi-rate executive for periodic tasks scheduler adds the ability to have multiple periods by instituting a timer that is the lowest common multiple of all of the periods of all of the tasks. At each tick of this timer, tasks can be made to execute. All of the above scheduling algorithms usually deal with interrupts by inserting tasks that poll for them, and all of the above scheduling algorithms are nonpreemptive. A multi-rate executive with interrupts allows external interrupts to break into current execution and be serviced. The task interrupted is then restarted when the interrupt is done. Finally, the priority based preemptive executive scheduler is the same as the multi-rate executive with interrupts except that it allows not only interrupts to break into the current program, but tasks with higher priority as well [27].

Scheduling algorithms are either static or dynamic. Static scheduling is performed when the execution times of all tasks to be scheduled by the scheduler are determined before any execution has taken place. Static scheduling is done when the deadlines for all of the tasks are known, and the time that it takes to execute those tasks is also known. All of the scheduling is done offline, before the execution of any tasks has begun, and is fixed. Dynamic scheduling is performed when the execution time of the tasks to be run is not fixed, is variable, and scheduling orders and priorities must be done dynamically during execution. This is done when task priority, execution time, or deadlines either change during execution, or are unknown before execution begins. The order in which tasks are scheduled and executed is decided upon during runtime, and is variable [34].

### 2.3.2: User Tasks and Threads

In RTOSs, user tasks are implemented in the form of threads. Each thread implements a computation job and is the basic unit of work handled by the scheduler. When the kernel creates a thread, it allocates memory to that thread and brings in the user code to be executed by that thread. The two different types of threads are periodic and aperiodic. Obviously, aperiodic threads run only once while periodic thread runs continuously at a given frequency [34].

Their are five major states of threads. The first is sleeping: this is when a task is set to sleep for a certain amount of time before it is to be woken up and run. The second state is ready: this is when the thread is ready to run and is simply waiting for the resources to do so. The third state is that of executing: this is when a thread is currently running on the operating system. The fourth state is that of suspended, or also known as blocked: this is when the task cannot proceed for some reason, such as waiting for another event to occur, or for some value to be brought in. The final state is terminated: this is when a thread has run, and is not to be run again [34].

### 2.3.3: The Kernel

The kernel in any RTOS, as mentioned in the introduction to this section, is responsible for four things. They are scheduling, system calls, timing maintenance, and handling interrupts. The RTOS is responsible for maintaining a schedule for all of the tasks running on it, and one of the above scheduling techniques is usually chosen. A system call is any function that the kernel might do at the request of a user thread. To perform a system call, the user task places the name or ID of the function that it wishes to run in a preset location and then traps to the kernel. After the context switch has taken place, the kernel looks up the function that it has been asked to complete, completes it, and puts the result of that function, if there is one, in a second preset location, and then returns control over to the user process. It is also possible for the user process to make a system call and continue working while the kernel is performing this system call. The kernel is also responsible for maintaining the timer. Every time that a timer interrupt is handled, the kernel must update the time as well as wake up tasks that need to be woken up and put on the ready queue. The last thing that a RTOS is responsible for is the handling of interrupts. Upon a interrupt, the hardware starts the RTOSs exception handler software. The RTOS is then responsible for saving the current state on the stack, determining the type of interrupt that has interrupted normal processing, and to know where that interrupt's service routine is. It then turns over control to that interrupt's service routine. After that routine has finished, the kernel is also responsible for transferring control back to the user process [34].

### 2.3.4: Synchronization and Communication

In addition to all of the above requirements, RTOSs are also responsible to provide methods of synchronization and communication between tasks. Mechanisms such as semaphores, mutexes, and condition variables add the ability for tasks to synchronize amongst themselves. To allow communication between the tasks, mechanisms such as message queues, mailboxes, and shared memory can be provided by the RTOS [34].

### 2.3.5: The Emulator's Benefit to Real-Time Operating Systems

One of the biggest decisions in choosing a RTOS for an embedded system is not which RTOS to choose, but whether or not to use a RTOS. Unlike the world of general purpose computing, embedded systems are usually developed for a limited number of tasks. Any facilities that these tasks might need are often built directly into the code, so many designers believe that a real-time operating system would just add unnecessary overhead [13]. What is needed, and what this research provides, is a method to test both commercially available Real-Time Operating Systems and in-house creations on the target architecture to verify which would give the best performance, without having to run the RTOS on the actual hardware, saving both time and money.

## 2.4: Evaluation of Real-Time Systems

From complex mathematical theories to full system hardware simulation, there are many different ways to evaluate real-time systems. The evaluation of these systems, like research in many fields, usually falls into two parties; theoretical and experimental. While many argue for one over the other, these two fields should not be at odds against each other. They are in fact complementary, and any evaluation cannot really be said to be complete without both having been performed. This section provides both the methods and metrics used to evaluate those real-time systems, provides examples of current research that is being done in the evaluation of real-time systems, and concludes with the benefits that the emulator that was developed for this report can give to the evaluation of real-time systems.

### 2.4.1: Methods of Evaluation

There are varying levels of real-time systems evaluation. The most prevalent ones are the use of analytical models, the simulation of scheduling algorithms, and hardware simulation. Analytical models are mathematical theorems and proofs that model the worst time performance of one or more of the aspects of real-time systems, and by changing certain inputs to these theorems, an optimum performance can be proven. Simulation takes the analytical models one step further in creating a simulation using scheduling theory to experiment with behavior of real-time systems. Finally, hardware tests take the theorems that were postulated by the analytical model and have been simulated through the use of scheduling algorithms, and run tests on the actual hardware to discover any behavior that was not determined through either of the other two methods.

### 2.4.2: Metrics of Characterization

Two of the most common metrics used to characterize real-time systems are jitter and response time. Jitter represents the minimum and maximum time separating successive iterations of periodic tasks. If this inter-arrival time is greater than the period of the task, it means that the task is running late, and this will show up as a positive jitter value. If that inter-arrival time is less than the period of the task, that means that the task is running early, and this will show up as a negative jitter value. Response time is the time that it takes for a real-time system to respond to an external interrupt and represents the reaction time of the system to an unscheduled event while under load.

### 2.4.3: Current Studies

Simulation and hardware execution of real-time software has been used in many different projects: from validating the accuracy of schedulers and analytical models [2, 8, 30, 33, 53], to measuring worst case execution time of functional blocks in dataflow graphs [17, 19, 22, 31], to measuring the effects of pipelined and superscalar processors on timer analysis [23, 35, 51], and to validating the performance of real-time databases [7, 21, 26]. However, while some simulations are accurate down to cycle behavior, most experiments model systems by using dataflow graphs to represent real-time system behavior.

### 2.4.4: The Emulator's Benefit to the Evaluation of Real-Time Systems

The analysis of these scheduling algorithms should be accompanied with experimental evaluation on the actual hardware. Unfortunately, this sometimes presents a problem when the hardware is not available, or there is a question of money or time. However, with the emulator developed for this research it is possible to run tests on an emulation of that hardware, saving both time and money.

## 2.5: SimOS

The research effort going on currently that most resembles this work is the SimOS project going on at the SimOS group at Stanford [44]. The sections following will discuss what SimOS is, describe the studies that have been performed using it, and detail the differences that exist between it and the emulator created during this research.

### 2.5.1: The SimOS Approach

SimOS is a full-system simulation environment that is capable of modeling computer hardware in enough detail to run a complete operating system, and all of the applications running on that operating system, on top of it [40]. The SimOS project started in 1992, and was built to study the execution behavior of modern workloads [25]. It is capable of studying both uniprocessor and multiprocessor systems and is used to study and evaluate the performance of high-performance and general purpose computers [42, 44].

The SimOS environment is a simulation layer that runs on top of general-purpose Unix multiprocessors such as the Silicon Graphics Inc. Challenge series [40]. On top of that general purpose multiprocessor system is the operating system running on that hardware, and in this case, it is IRIX version 5.x. On top of this software is run the SimOS environment. The SimOS environment takes in a hardware description file and is capable of modeling uniprocessors, multiprocessors, RAM, ethernet, hard disk, and other pieces of hardware associated with today's hardware platforms. On top of the SimOS is run an operating system that has been ported to the hardware platform that the SimOS environment is currently modeling. Finally, on top of that operating system is run the unaltered applications programs [40]. All of this can be seen in Figure 1.

One of the advantages of the SimOS operating systems is that it allows the user to choose which level of output detail in which to simulate. The system offers a simple trade-off of speed versus detail of simulation [40]. If the user is interested in obtaining detail simulation results of a particular program, SimOS employs slower, more detailed simulation. And when the user wishes to run an application for long periods of time

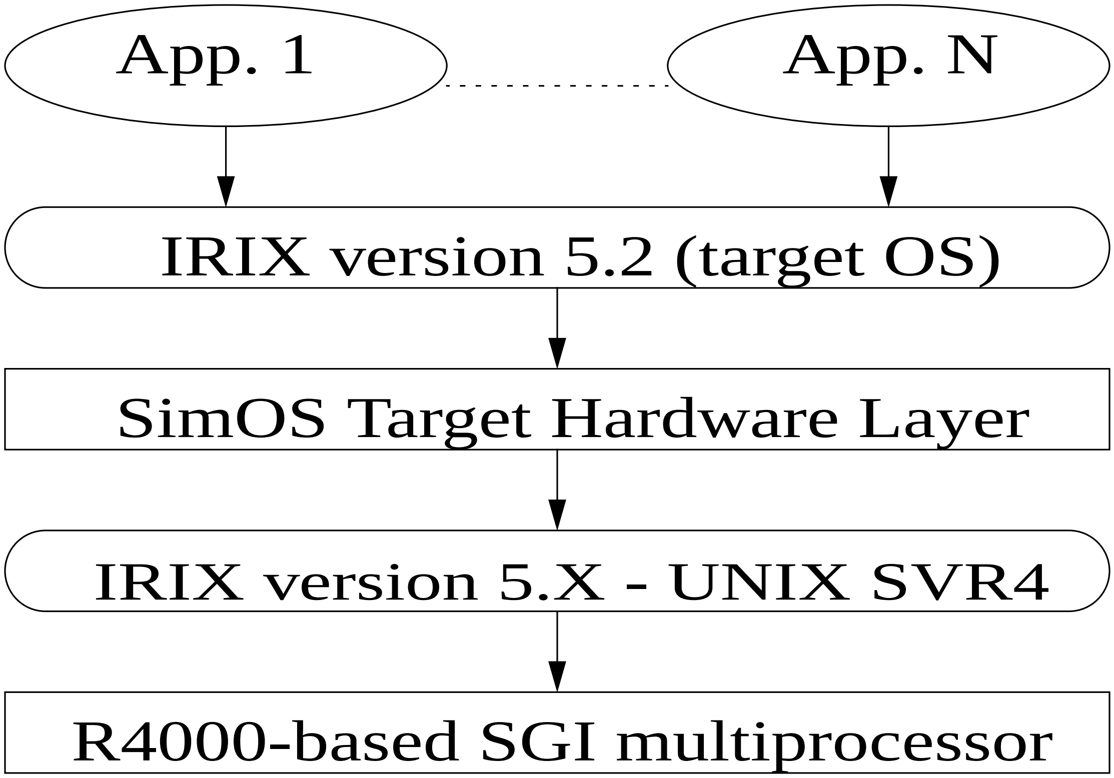

App. 1
App. N
IRIX version 5.2 (target OS)
SimOS Target Hardware Layer
IRIX version 5.X - UNIX SVR4
R4000-based SGI multiprocessor

Figure 1: The SimOS Environment.
This figure shows the layout of the SimOS development environment. The SimOS target hardware layer runs on top of an Unix Operating System running on a R4000-based SGI multiprocessor workstation. On top of the SimOS environment is run the target Operating System, and any applications that are run on top of that Operating System.

instead of for detailed simulation results, SimOS can scan over unimportant parts of the workload. Also, SimOS allows the user to modify this choice on-the-fly. The user can choose certain sections of code that he is interested in seeing the simulation results for, and scan over the rest of the code as unimportant[44].

2.5.2: Studies Performed with SimOS

This SimOS simulator has been used in a variety of different research studies. SimOS has aided studies in the areas of architectural evaluation, such as the study of the Stanford FLASH multiprocessor. With the detailed model provided by SimOS, researchers within this project have examined the performance impacts of several of their design decisions. SimOS has helped studies of system software development, such as in the development for an operating system for the above mentioned FLASH multiprocessor. Having the ability to run on the operating system on the SimOS simulated hardware has provided a more complete set of debug information for those researchers than they could obtain from the actual hardware. SimOS has also been used in the research of workload characterization, such as the characterization of the Sybase database [40]. Other studies performed with the aid of SimOS include the breakdown of operating system execution time on today's processors as opposed to tomorrow processors [41], in the evaluation of different organizations of chip multiprocessors [39], and the evaluation of the system performance on large commercial workloads [6].

### 2.5.3: SimOS versus our Emulator.

The SimOS simulation environment differs from the emulation tool developed during this research in two different ways. The first is that the two have very different target application domains. The SimOS system studies both high performance and general purpose systems. The emulator developed for this research was created to study embedded systems.

The second difference is the system characteristics that each studies. Being created to study high-performance and general purpose systems, SimOS studies mainly end-to-end performance. The emulator developed during this research was created to evaluate the real-time performance of low power embedded processors, such as measuring the difference between the invocations of periodic tasks, measuring if they are either early or late, and measuring the reaction time from external interrupts or stimuli.

# Chapter 3: The Emulator

For this project, Motorola's M-CORE architecture was used as the model architecture for our emulator. This architecture was chosen because the M-CORE architecture is one of the cutting edge embedded processors on the market today, and the M-CORE was designed for high performance and low power operation [37]. In this chapter, first the M-CORE architecture is described, followed by the specifics of the emulator, how it works, what information it takes as an input, how it processes that information, and what information it outputs during the emulation. Finally, the method used to validate the emulator is described. Figure 2 shows a system view of the emulator and both the hardware and operating system that it is running on and the Real-Time Operating System and applications that are running on it.

## 3.1: M-CORE Architecture

The Motorola M-CORE architecture is a 32-bit Load/Store architecture with a fixed 16-bit instruction length and 32-bit data length. Figure 3 shows all of the available instruction formats in the M-CORE architecture. It has a 16 entry 32-bit general register file, a 16 entry 32-bit alternate register file to allow fast interrupt support, and a 13 entry control register file accessible only by the supervisor mode. Its execution pipeline's four stages are completely hidden from the application software. Most instructions execute in a single cycle with two cycle execution for loads, stores, and taken branches and

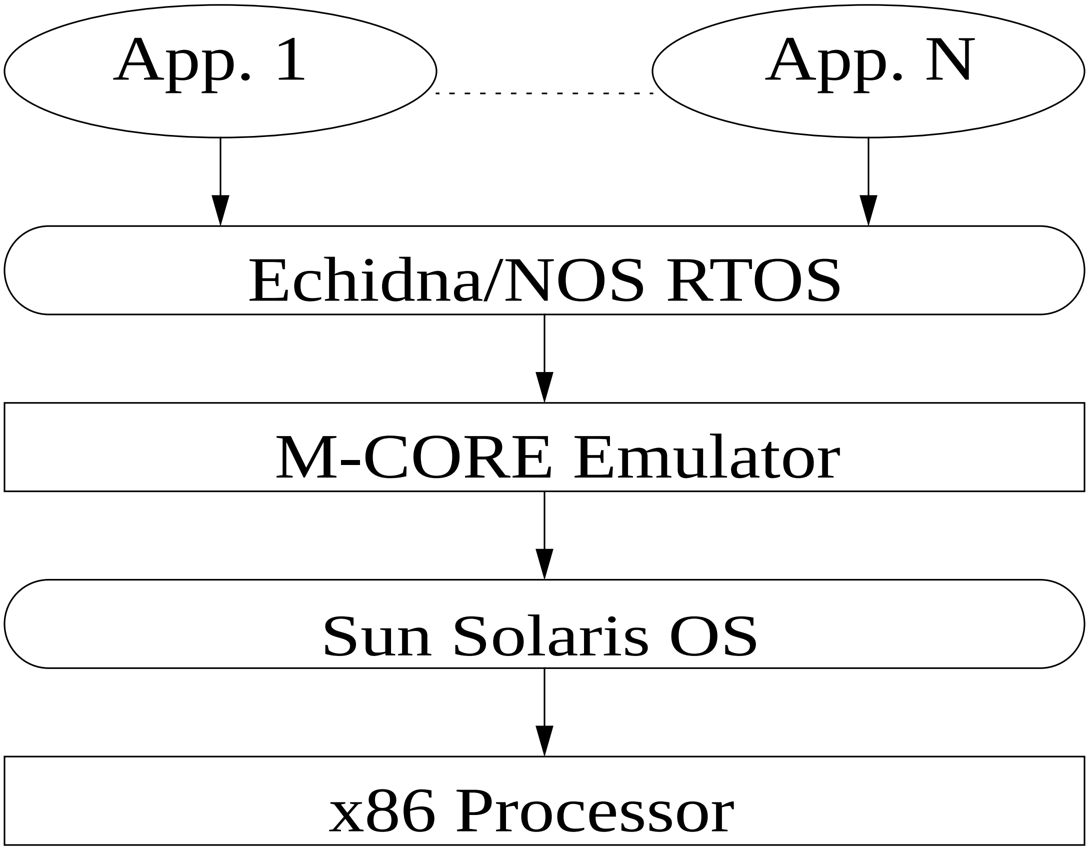


Figure 2: The Emulation Environment.
This figure shows the emulation environment developed in this work for real-time system evaluation. The emulator runs on top of the Sun Solaris Operating System, running on top of a x86 Processor. On top of the emulator is run the target Real-Time Operating System, either Echidna or NOS. On top of the RTOS, the benchmark applications are run.

jumps. The address space is byte, halfword, and word addressable, and allows both fast and normal interrupts, allowing those interrupts to be either vectored or autovectored interrupts.

The pipeline for the M-CORE consists of four stages: instruction fetch, instruction decode/register file read, execute, and writeback. All of these stages operate simultaneously, making single cycle instructions possible. All sixteen general purpose

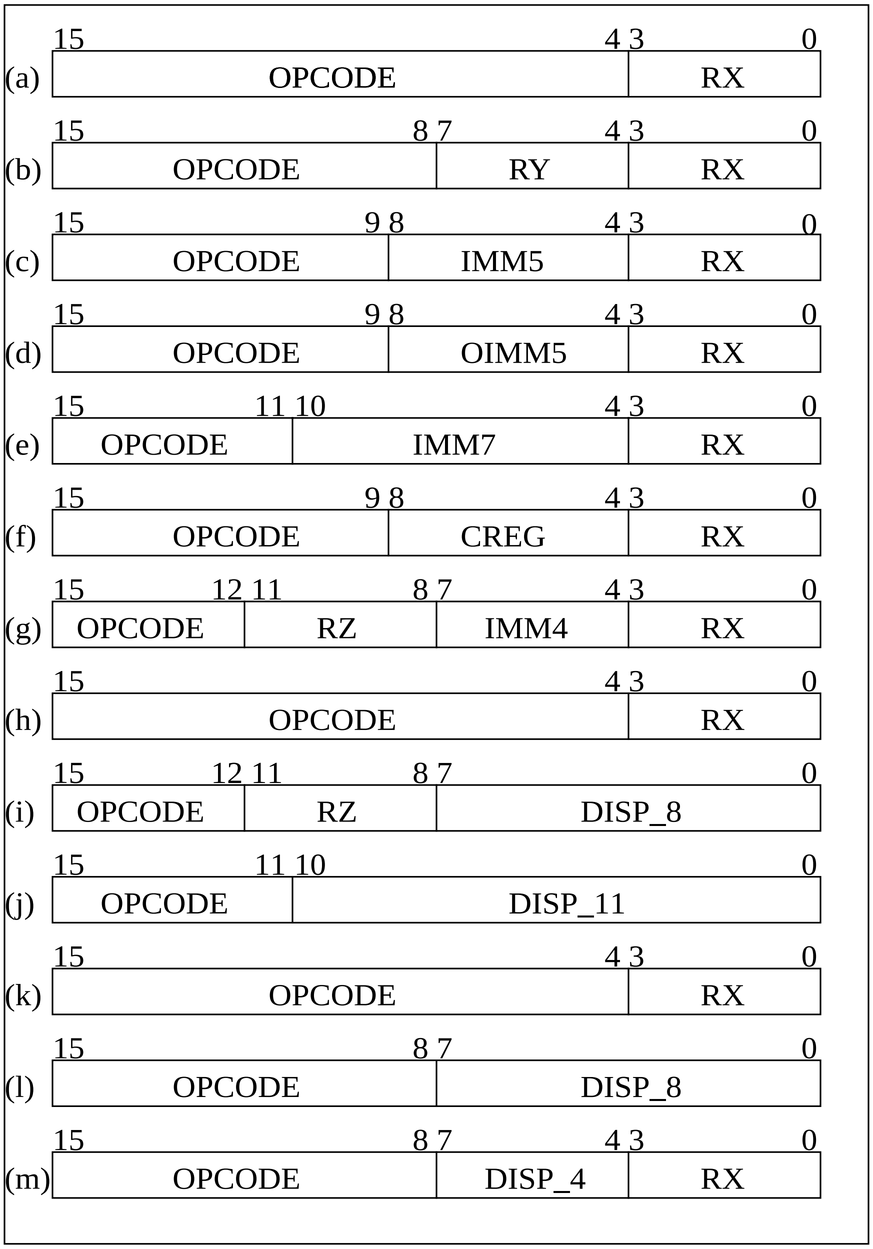


Figure 3. Instruction Format.

The above figure shows all 13 of the possible instruction formats for the M-CORE architecture. The first six instruction formats are register to register instructions, and they are: (a) Monadic Register Addressing, (b) Dyadic Register Addressing, (c) Register with 5-Bit Immediate, (d) Register with 5-Bit Offset Immediate, (e) Register with 7-Bit Immediate, and (f) Control Register Addressing. The next three are data memory access instructions, and they are: (g) Scaled 4-Bit Immediate Addressing, (h) Load/Store Register Quadrant and Multiple Register, and (i) Load Relative Word. The last four formats are flow control instructions, and they are: (j) Scaled 11-Bit Displacement, (k) Register Addressing, (l) Indirect, and (m) Register with 4-Bit Negative Displacement.

registers can be used as source operands and instruction results (i.e. it is an orthogonal register file).

The architecture's execution unit contains the following sub-units:  A 32-bit ALU, a 32-bit barrel shifter, a find-first-one unit, a multiplication/division unit, and result feeding forward hardware.  All arithmetic instructions are single cycle instructions with the exception of the multiply and signed and unsigned divide instructions.  The multiply is implemented with a 2-bit per cycle Booth algorithm, and the divide instruction's timing is also data dependent.

The program counter (PC) unit has a PC incrementer and a dedicated branch address adder.  This minimizes the change of instruction flow delays to only a single pipeline bubble delay, for the branch target addresses are calculated during the instruction decode phase.  If a branch is not taken, no delay is incurred.

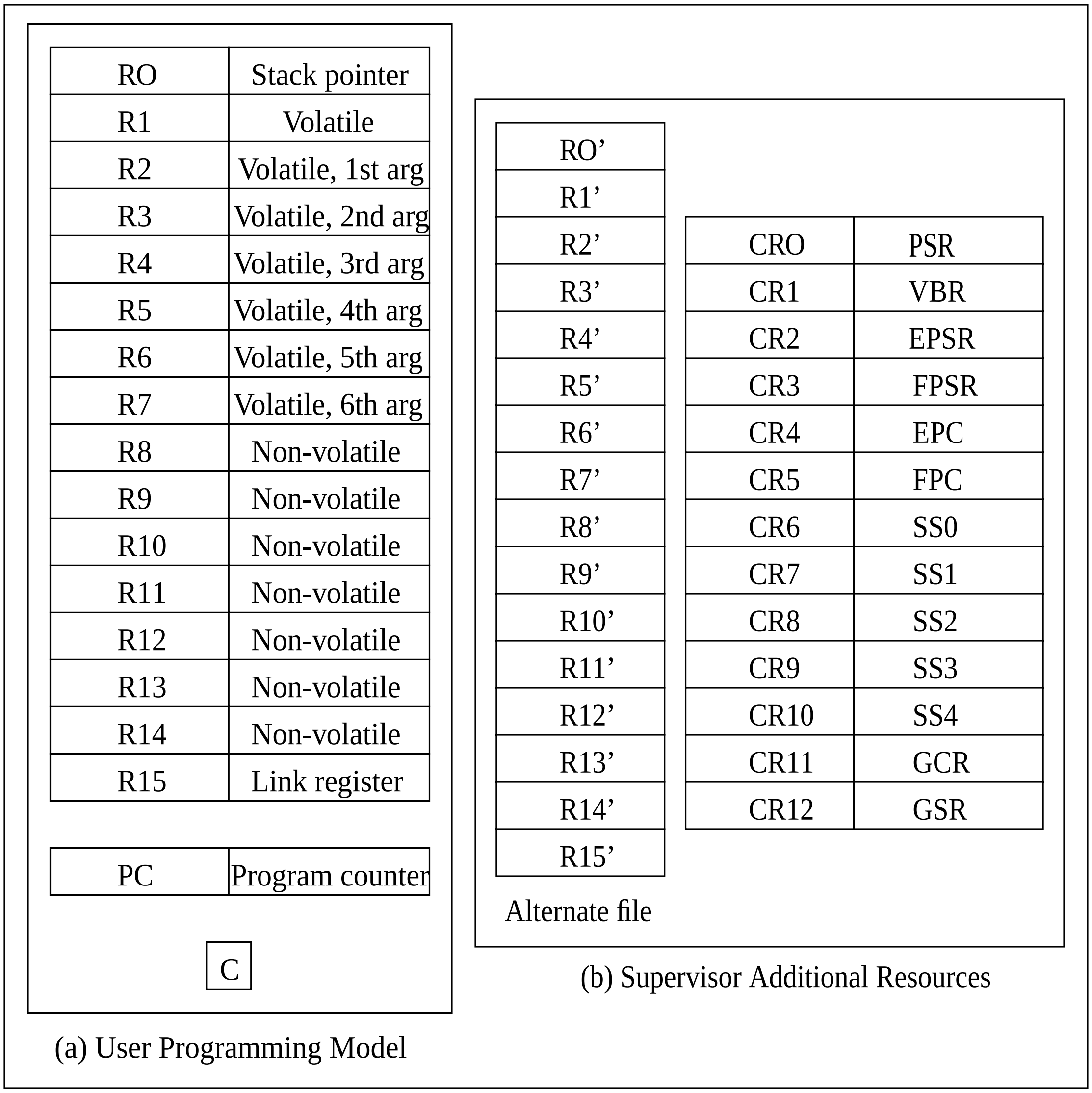
Byte, halfword, and word memory accesses are provided with this architecture, with an automatic zero-extension of bytes and halfwords.  Single memory accesses, independent of size, execute in two cycles.  Multiple memory accesses, such as the Load Multiple instruction, or the Store Quadrant instruction, can execute in a number of cycles equal to the number of words transferred plus one.

The M-CORE programming model is defined for two privilege modes:  supervisor and user mode.  There are certain operations not available in user mode.  User programs can only access registers in the general register file, whereas supervisor mode programs can access all registers, using control registers to perform supervisory functions.  User programs are prohibited from accessing privileged information (the control registers, vector offset table, settings for I/O, etc.).  If a user program tries to access privileged information or tries to execute a privileged instruction, a privilege violation exception occurs.  A single bit (the S bit) in the Program Status Register(PSR) determines which mode the architecture is currently running in.

This architecture uses the user programming model during normal user mode operation.  During exception and interrupt processing, the processor changes over from user mode to privileged mode.  Exception processing saves the current values of the PC and the PSR, and then sets the S bit in the PSR, and loads the new PC from the exception vector table.  During the return from exception (rfe) instruction (or return from fast interrupt (rfi) instruction for fast interrupts) the original PC and PSR values from before the exception are restored, and execution continues in the user mode.

There are thirteen control registers in the M-CORE architecture that can only be accessed during the supervisor mode.  These include the PSR as mentioned above, the Vector Base Register (VBR) holding the base address used in the calculation of exception handler PCs, an Exception Program Status Register (EPSR) and Exception Program Counter (EPC), to store the PSR and PC during exceptions, a Fast Interrupt Program Status Register (FPSR) and Fast Interrupt Program Counter (FPC), to store the PSR and PC during fast interrupts, five scratch registers for supervisor software to use during the handling of exceptions, and two registers are used for global control and status.  Both the user programming model as well as those resources that are available during supervisor mode can be seen in Figure 4.

The M-CORE supports two’s-complement data formats, and instructions either explicitly encode the operand size in the instruction (load/store) or implicitly define it for the instruction operation (index operations, byte extraction).  Memory is viewed



| | |
|---|---|
| RO | Stack pointer |
| R1 | Volatile |
| R2 | Volatile, 1st arg |
| R3 | Volatile, 2nd arg |
| R4 | Volatile, 3rd arg |
| R5 | Volatile, 4th arg |
| R6 | Volatile, 5th arg |
| R7 | Volatile, 6th arg |
| R8 | Non-volatile |
| R9 | Non-volatile |
| R10 | Non-volatile |
| R11 | Non-volatile |
| R12 | Non-volatile |
| R13 | Non-volatile |
| R14 | Non-volatile |
| R15 | Link register |

| | |
|---|---|
| PC | Program counter |

| |
|---|
| C |

(a) User Programming Model

| |
|---|
| RO’ |
| R1’ |
| R2’ |
| R3’ |
| R4’ |
| R5’ |
| R6’ |
| R7’ |
| R8’ |
| R9’ |
| R10’ |
| R11’ |
| R12’ |
| R13’ |
| R14’ |
| R15’ |

Alternate file

| | |
|---|---|
| CRO | PSR |
| CR1 | VBR |
| CR2 | EPSR |
| CR3 | FPSR |
| CR4 | EPC |
| CR5 | FPC |
| CR6 | SS0 |
| CR7 | SS1 |
| CR8 | SS2 |
| CR9 | SS3 |
| CR10 | SS4 |
| CR11 | GCR |
| CR12 | GSR |

(b) Supervisor Additional Resources

Figure 4: User Program Model and Supervisor Additional Resources. The above figure shows both the limited resources available during user mode execution as well as the additional resources available during supervisor mode execution. In (a), it can be seen that the user mode programs can use any of the sixteen general purpose registers, access the Program Counter, and the Carry bit. In supervisor mode (b), the alternate register file can be accessed, as well as all of the control registers.

from big-endian perspective, meaning the most significant byte (byte zero) of word zero is located at address zero.

There are ninety-eight instructions for the M-CORE architecture. A table of these can be found in Appendix A (Table A-1).

All of the information in the above section has been obtained from [36, 37, 38, 52].

## 3.2: Emulator Parts

The emulator for the Motorola M-CORE was written entirely using the C programming language and can be broken down into several distinguishable parts. The sections following look at each of those points in detail. Section 3.2.1 describes the method for bringing in the program to be run on the emulator using the ELF file format. Section 3.2.2 describes the mechanisms used to store and retrieve information in the main memory and the general purpose registers. Sections 3.2.3-3.2.6 describe the pipeline. In the emulator, the pipeline is instantiated in reverse order, that is, write back first, then execution, then decode, and finally fetch. The order does not matter since all stages are taking place simultaneously, but ordering the phases this way in C allows for easier handling of data transfer as well as for exception processing. Section 3.2.7 describes all of the post stage maintenance that needs to be maintained every cycle, specifically maintaining the timer, checking for interrupts, and handling exceptions. Section 3.2.8 describes the optional output of the simulator. Finally, Section 3.3 describes the method used to validate the emulator.

### 3.2.1: ELF Input

The emulator takes as its input the executable and linking format (ELF) file that is produced by the compiler. An ELF file, like any other compiler output file, contains all of the information necessary to run a program on its target hardware. The ELF file contains sections of data called program segments that are blocks of data to be placed directly into the M-CORE's memory at locations also given in the file as physical

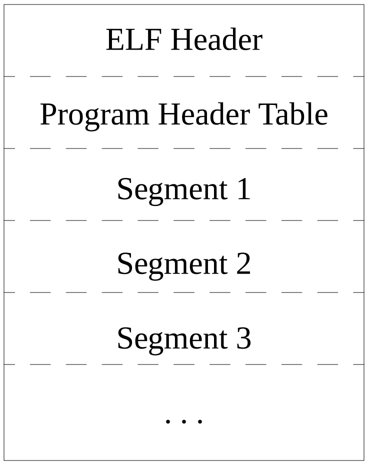


Figure 5: Format of an ELF file.
This figure shows the breakdown of an ELF file. The first group of information is the ELF Header. This data will contain the location of the Program Header Table. The Program Header Table gives information about the programs segments: How many are there, where are they located in this file, and were do they need to be placed in the target systems memory map.

addresses. Each one of these areas are given as either read-only (code), or read-write (such as the stack and the heap). Writing to either an undefined area of memory, or to a read-only section causes an exception. See Figure 5 for an overview of the ELF file format. The emulator uses a separate executable C program that brings the ELF file in, parses it, and creates a file the emulator can read. This file contains a translation map consisting of the start address, final address, and offset for each block of memory — and the actual data itself. During emulator start up, this file is brought in, the translation map is loaded, and the data is placed into the memory storage device (described in 3.2.2).

### 3.2.2: Main Memory and Registers

Given the four gigabytes of addressable memory space in the M-CORE architecture, it is both impossible and unnecessary to store the entire addressable memory space in an array. Instead, the emulator stores smaller segments of that addressable space, located in arrays, and institutes a method of accessing locations in memory through a translation map. For example: A load instruction wishes to load a word from memory location ‘x’. From looking in the translation map, the emulator finds that address ‘x’ falls between locations ‘w’ and ‘y’. The associated offset to values that fall between ‘w’ and ‘y’ is ‘z’. Therefore the data at that memory location can be found in the memory array using an index of ‘x’-‘z’.

Since the M-CORE architecture uses memory mapped devices, the emulator uses two main memory structures, one for values sent to and from the physical memory and one for values sent to and from the memory mapped devices such as I/O and the timer. The first structure, OnChipRam, is the physical memory. This is where the data brought in from the ELF file is placed. The second structure, OnChipMapRam, is an array containing all locations in the memory map that correspond to memory mapped devices, such as the exception vector table, interrupt registers, timer registers, and I/O devices.

Other values that need to be stored, beside the above mentioned memory arrays, the sixteen member 32-bit register file, its shadow register file, and the control register file, are the PC value and instruction before and after each stage. Therefore the emulator has five pairs of registers: MAIN and MAINpc store the values about to enter the pipeline, IFID and IFIDpc store the values between the Instruction Fetch and Instruction Decode stages, IDEX and IDEXpc store the values between the Instruction Decode and Execution Stages, EXWB and EXWBpc store the values between the Execution and Write Back stages, and WBEnd and WBEndpc stores the values leaving the pipeline.

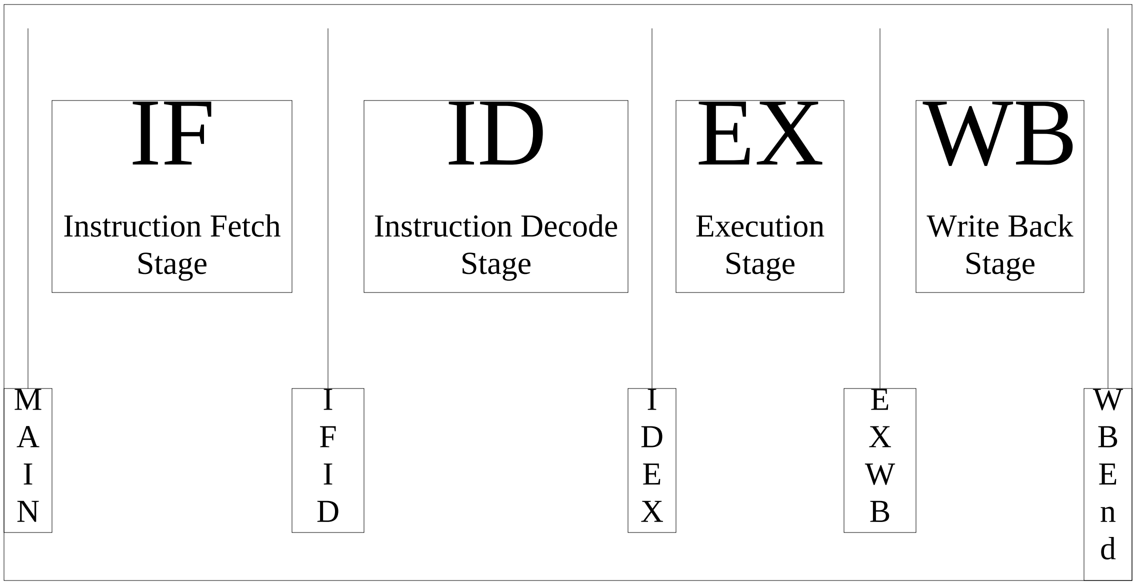


Figure 6: The M-CORE pipeline.

The above diagram shows the M-CORE pipeline. It consists of four stages: Instruction Fetch, Instruction Decode, Execution, and Write Back. Between each of the stages and at the beginning and end of the pipeline there are pairs of registers that hold information between cycles. These registers are MAIN & MAINpc, IFID & IFIDpc, IDEX & IDEXpc, EXWB & EXWBpc, and WBEnd &WBEndpc.

### 3.2.3: Write Back Stage

In the emulator, since the majority of the assignments are done in the execution stage, the only operation that occurs during this stage is the moving of PC and instruction values from the EXWB registers before this stage to the WBEnd registers following this stage. This stage's purpose is to show which instruction is currently in the write back stage of the pipeline.

### 3.2.4: Execution Stage

Both instruction execution and assignment take place during this stage. All of the instruction execution code for each instruction is placed into a function of the same name as that instruction. All of these instructions are then mapped into an array of functions, indexed by the opcode. The first level of the function array is indexed by the most significant four bits in the opcode. If these first four bits are enough to determine which instruction is to be executed, then that instruction's function is immediately called. If not, a second level function array is indexed with the second four most significant bits of the opcode, and so on. On the average, only the first two function pointers need to be indexed before the instruction has been found, greatly improving upon the worst case performance of a 98 element if-then-else-if-then-else/switch statement.

Once the function is called, the instruction is executed. It is also important to note here that during this phase the divide instructions check for possible divide-by-zero exceptions and the load and store instructions check for misaligned access to memory exceptions. Another important point to make here is that there are instructions that take more than one cycle to execute, such as loads and stores, branches, and multiplies and divides. If the instruction executing is one that takes more then one cycle, a stall variable is set with the number of bubbles that need to be inserted into the pipeline.

After executing the instruction, if the stall variable is still zero, the PC and instruction values for the execute phase are moved to the EXWB registers. However, if the stall variable is not zero, the PC and instruction values are not passed on, since they do not leave this stage (they are still executing). It is also important to note that if the stall variable is greater than zero during the beginning of this stage, no instruction will execute since the instruction currently in the stage is the one that caused the stall and has already executed.

### 3.2.5: Instruction Decode Stage

The instruction decode phase is almost as important as the execution phase in that it also checks for a number of exceptions that can happen during the decode stage.

These exceptions are the illegal instruction exception, the privilege violation exception, and the four different trap exceptions.  It is important to catch these exceptions here because they must not be allowed to reach the execution phase.  If the exception occurs here, the instruction in this stage must be flushed and exception handling must begin.

At the end of this stage, like the end of the execution stage, the PC and instruction values are moved to the IDEX registers if the stall variable is set to zero.  If not, it is assumed that this instruction is stalled here while waiting for the instruction executing in the execution stage to finish.  Also, the exception check is not made if the stall variable is greater than zero (since this check already occurred when the instruction originally got to this stage).

### 3.2.6:  Instruction Fetch Stage

During this stage, the instruction is fetched from memory, and passed to the IFID registers.  Again, however, this only occurs when the stall variable is set to zero.

### 3.2.7:  Post Stage Maintenance

After all of the stages have completed, there are a few tasks that need to be performed before the emulator can continue onto the next cycle.  Specifically, if the timer needs to be incremented, that needs to be completed.  Also, a check must be made to verify that no interrupts are waiting to happen.  Finally, if an exception has occurred, state needs to be saved, and the exception handler needs to be started.

#### 3.2.7.1:  The Timer

The timer described in the M-CORE documentation sets off a timer interrupt every 0.1 seconds.  Since the emulator is modeling a 20MHz processor, that means that every 2 million cycles there needs to be a timer interrupt.  A timer tick variable (two bytes) is set, and every time that the variable overflows, an interrupt occurs. The variable flows from a start value of 0x0bdb and continues to 0xffff. This calculates to 62,500 “ticks”. Dividing this value into 2 million cycles determine that every 32 cycles, the timer tick value needs to be incremented. Therefore, every cycle, a check is made. If it has been 32 cycles since the last timer tick increment, the timer tick is incremented. If when this occurs, the timer tick overflows, the timer tick is set back to 0x0bdb (0xFFFF-62,500), and a timer interrupt is signaled by the emulator [36]. Since this description of the timer is very straight forward and easy to program in C, this is how the emulator implements the timer.

#### 3.2.7.2: Interrupts

The Motorola M-CORE architecture allows for both fast and regular interrupts to occur. The difference between a fast and a regular interrupt is that a regular interrupt can be interrupted (by a fast interrupt) while a fast interrupt cannot. To determine whether an interrupt has occurred, every cycle the M-CORE architecture does a bitwise AND between two registers: the interrupt source register (the fast interrupt source register for fast interrupt and the regular interrupt source register for regular interrupts), which will contain all zeros, unless there is an interrupt pending on one of the interrupt lines; and the interrupt enable register (the fast interrupt enable register for fast interrupts, and the regular interrupt enable register for regular interrupts), which is set by the supervisor, determining which interrupts will be serviced, and which will be ignored.

If the ANDing of these two registers returns any value other than zero, an interrupt occurs. The interrupt number is then determined and from that number a vector offset is then determined and sent to the exception handler. Interrupts are handled by the exception handler. The only difference is the method in which they are detected, and that the interrupt vector offset is sent to the exception handler [36, 37, 38]. Being that this is very straight forward and easy to program in C, this is how the emulator implements interrupts.

### 3.2.7.3: Exceptions

In the M-CORE architecture, if either an exception or an interrupt has occurred, a flag is set showing that an exception (or interrupt) has happened and needs to be handled. The first thing that occurs is that both the current PC and PSR values are saved to either the EPC and EPSR registers (for exceptions and regular interrupts) or the FPC and FPSR registers (for fast interrupts). The PC value that is stored is determined by what type of interrupt or exception that is happening. If it is an exception that occurred during the instruction decode stage, the PC for the instruction in the decode stage is stored. Likewise, if the exception was a divide-by-zero or misaligned access in the execution stage, the PC of that instruction is saved. However, if a misaligned access occurred during the calculation of either a jump to subroutine immediate (jsri) or jump immediate (jmpi) instruction, the PC value fetched is the value that is stored. All of the instructions in the pipeline behind the instruction that caused the exception are flushed. However, all the instructions further along in the pipe are allowed to complete.

If an interrupt caused the exception handler to start, the PC value stored is the PC after the instruction that is in the instruction fetch stage. So whenever an interrupt occurs, it allows all instructions in the pipe to complete before beginning the exception handler.

After the PC and PSR values have been saved, certain bits are set in the new PSR to disable further exceptions and interrupts (except fast interrupts, unless the exception was caused by a fast interrupt — in that case, no further exceptions or interrupts are allowed), and then the location of the new PC for the exception handler is calculated. If the exception occurring is an interrupt, then the vector offset is already known. If not, then the vector offset is calculated by multiplying the exception number by four. This value is then added to the value in the vector base register (control register one), and then the new PC value is loaded from this location in memory. Then, after the appropriate number of cycles to emulate the delay that all of the above actions take, execution is started at this new PC.

Upon completion of the exception handling, either a return from exception (rte) or return from fast interrupt (rfi) instruction is reached, which will reset the PC back to the value that it was before the exception, as well as reset the PSR [37, 38]. Again, this is very straightforward and is how exceptions are implemented in the emulator.

### 3.2.8: Output

During the calling of the emulator, the user has the option of turning on detailed output. If this option is set, the emulator prints out information at the end of every cycle. The information that it prints out is: what instruction is currently in the execution stage of the pipeline, the current cycle number, a listing of the pc address and opcode for the instruction in each stage of the pipeline, the condition bit, and values contained in the register file, the shadow register file, and the control registers.

For use during the experiment stage of this research, the emulator outputs a statement whenever a particular I/O address has been written to, giving the I/O address written to and the time in microseconds.  This shows when processes have executed and whether or not they are executing at their set frequency.

Finally, at the end of execution, the emulator prints out the number of microseconds that it has spent in each section of the code.  This is performed by having the process running on the emulator store a value to a particular memory address denoting what section of the code it is in:  Kernel code, Application code, Interrupt Handling code, or Idle code.  The emulator reads this memory location every microsecond and increments the time value for whatever section it is in.

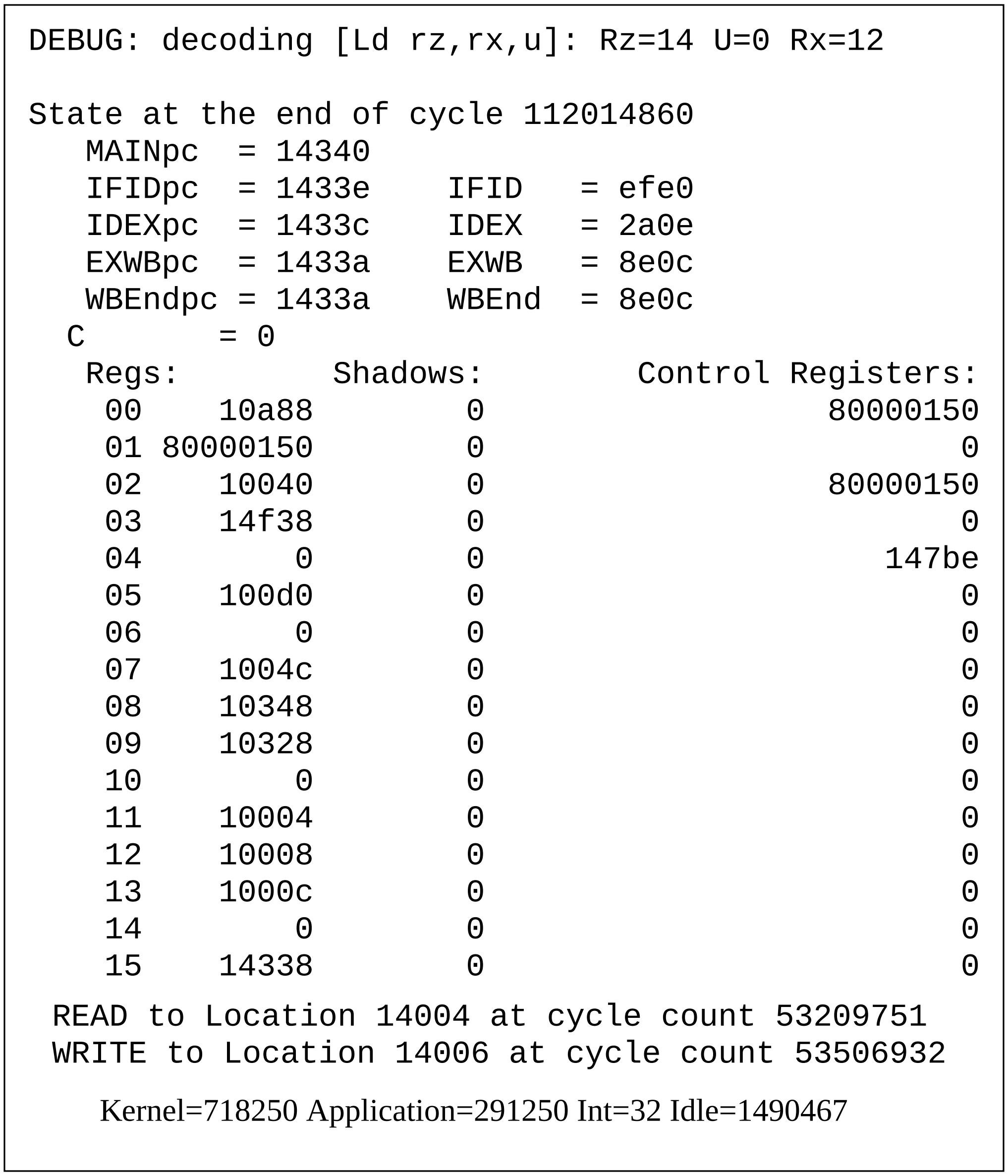
An example of all of this output can be found in Figure 7.

### 3.3:  Validating the Emulator

There were two steps used to test and validate the M-CORE Emulator.  The first was the creation of a simple suite of 8 programs.  These applications and what operations they perform can be seen in Table 1.  This test suite was first run on the actual M-CORE hardware and then was run on the emulator.  These applications were used to make sure that all of the basic instructions of the M-CORE architecture were working.

By running these applications, several small bugs were found and eliminated in the code.  Also, a disparity with the M-CORE Reference Manual [37] was found.  The opcode for one of the instructions, move c bit to register (mvc), was listed differently in one of the areas of the manual than the actual opcode.  This problem was quickly found and fixed, for when the emulator was supposed to be performing the mvc instruction, it instead ran across an invalid instruction exception.

The second validation that was performed was the running of Echidna [10] (described in more detail in Chapter 4) on both the Emulator and the actual hardware.



```
DEBUG: decoding [Ld rz,rx,u]: Rz=14 U=0 Rx=12

State at the end of cycle 112014860
   MAINpc  = 14340
   IFIDpc  = 1433e    IFID   = efe0
   IDEXpc  = 1433c    IDEX   = 2a0e
   EXWBpc  = 1433a    EXWB   = 8e0c
   WBEndpc = 1433a    WBEnd  = 8e0c
  C       = 0
   Regs:         Shadows:        Control Registers:
    00    10a88        0                   80000150
    01 80000150        0                          0
    02    10040        0                   80000150
    03    14f38        0                          0
    04        0        0                      147be
    05    100d0        0                          0
    06        0        0                          0
    07    1004c        0                          0
    08    10348        0                          0
    09    10328        0                          0
    10        0        0                          0
    11    10004        0                          0
    12    10008        0                          0
    13    1000c        0                          0
    14        0        0                          0
    15    14338        0                          0

 READ to Location 14004 at cycle count 53209751
 WRITE to Location 14006 at cycle count 53506932

    Kernel=718250 Application=291250 Int=32 Idle=1490467
```

Figure 7: Output Example.

Shown above is the output that the emulator can provide the user. The first group shows the status of all of the registers at the end of a cycle. This prints every cycle if the user turns on the debug mode. The second group shows reads and writes to the various I/O ports used for the experiment stage of this research. These values printout whenever there is a read or write to one of the selected I/O ports. The last group shows the CPU breakdown for the program run on the emulator. This prints at the end of emulation.

| | |
|---|---|
| Application 1 | Basic Arithmetic |
| Application 2 | Basic Loop |
| Application 3 | Array Manipulation |
| Application 4 | Basic Pointer Manipulation |
| Application 5 | Complex Pointer Manipulation |
| Application 6 | Quick Sort Algorithm and Function Calls |
| Application 7 | Linear Sort Algorithm |
| Application 8 | Search and Return Algorithm |

Table 1: Test Applications.

The above listed applications were used to test the functionality of the emulator.

This was done for several reasons. The first reason was to test out the interrupt and exceptions portions of the Emulator code, which had not been tested. The second reason was to determine the speed at which the emulator could operate. The number that was achieved was 500,000 cycles per second, or 500KHz. Compared with the hardware speed of 20MHz, this is only a 40 times slow down — very respectable for a software emulation. The final reason for this test to determine how accurately the emulator was running against the hardware. Comparing the cycle counts between the emulator and hardware, it was determined that the emulator experienced less than 100 cycle difference per million cycles, or less than 0.01% error, this is exceptionally good, considering the SimOS simulator is typically 5-10% off and is considered acceptably accurate at that level[44]. This disparity is due to several factors. For the actual hardware to give information about the processor (register state, etc.) it must perform a breakpoint instruction in order to transfer the information off chip. Our emulator does not need to do this to return processor state. Also, the amount of time before an interrupt occurs and is serviced is variable, so we approximate it by using the average value. If the interrupt in our emulator starts at a different point in execution than the hardware does, the time spent servicing that interrupt can vary, due to the asynchronous behavior of interrupts.

At this point, the emulator was ready for the experiment.

# Chapter 4: Real-Time Performance Evaluation

The experiment carried out on the emulator was a real-time performance evaluation and comparison of an experimental RTOS and a bare-bones scheduler representing a minimal RTOS, as well as the theoretical performance. Both the experimental RTOS that was used and the bare-bones operating system that was created are described below. Following that is a description of the benchmarks run on those RTOSs to evaluate performance and the types of background noise added to simulate non-determinism in real-time systems. This chapter ends with a description of the experiment that was performed.

## 4.1: Echidna RTOS

Echidna is a cooperative multitasking Real-Time Operating System that is based on the Chimera [48] operating system developed at the Advanced Manipulators Laboratory at Carnegie Mellon University. A smaller version of Chimera (~6KB footprint), Echidna swaps Chimera's POSIX-like threads in the microkernel for port-based objects and supports reconfigurable component-based software for microcontrollers and digital signal processors [10].

The traditional coding method used by most of today's real-time operating systems is that processes are created, each with their own main(). Each of these processes executes their own user code and controls the flow of the program. This process calls upon the operating system whenever an operating system service is needed. These services include communication, time control, the creation of new processes, and synchronization. The port-based object method, on the other hand, gives a consistent structure for every process, and thus operating system services as listed above are performed in a predictable manner. Only when necessary, the operating system calls a port-based object's method to perform user-defined functions [49].

In this port-based object model, each independent object does not need to explicitly communicate or synchronize with any other component in the system, making integration very easy. When an object needs information, it obtains that information from its input ports. When that object generates informations that needs to be passed on to either another process, or to a future invocation of itself, it sends that information to its output ports. The information on these ports is stored in shared memory so data can be sent between objects [47, 50].

Echidna was designed to support dynamically reconfigurable real-time software and was targeted to run on 8 to 32 bit microcontrollers as well as DSPs [49]. Like Chimera, Echidna provides cooperative multitasking, but unlike Chimera, it offers a good deal of functionality in a relatively small footprint, and therefore it is a good candidate for this study in real-time performance in embedded systems.

## 4.2: NOS

The Non-Operating System (NOS) is a bare-bones, fixed priority, multi-rate executive [27] similar to a real-time operating system that an embedded-systems designer might create on the fly [14]. Although not a full operating system, just a task scheduler, NOS represents the attainable performance limit of a non-preemptive RTOS.

A multi-rate executive scheduler was chosen over something simpler, such as a basic cyclic executive or a time-driven cyclic executive scheduler [27], because we needed the ability to have several jobs that had different frequencies, and the timing of the multi-rate executive is less independent on the code size of the jobs run upon it, which is important because each of the benchmarks to be run is contains a different amount of code. NOS's main control loop can be seen in Figure 8.

When a job is created, it is set onto a callout queue similar to the callout table in UNIX [4]. Jobs added to this queue are assigned a location in this queue by the time that they are to occur. Upon entry to this queue, each job is assigned a delta value, which is the time difference between when this job is to run and the time the job ahead of it is to run. If a task is periodic, the last thing that it does during its runtime is to reinstantiate itself on the queue a time in the future equal to its period.

NOS is capable of handling two different priority levels of tasks (HARD and SOFT deadline tasks), and two different priority levels of interrupts (HIGH and LOW priority interrupts). Interrupts in NOS are handled via masking the interrupts and polling the interrupt status register. However, HIGH priority interrupts are not polled until all HARD deadline tasks scheduled to run have executed. Likewise, SOFT deadline tasks are not executed until all HARD deadline tasks scheduled to run have finished executing and all HIGH priority interrupts have been handled, and so on. For this experiment all tasks are run as HARD deadline tasks and all interrupts are HIGH priority interrupts. This means that if the NOS is overloaded with HARD deadline jobs, such that as soon as one job finishes another must run, interrupts will be handled much later than when they occur, if ever.

```
struct event {
     struct event *next;
     time_t delta;
     void (*execute)();
     char *data;
     int priority;       // HARD_DEADLINE or SOFT_DEADLINE
};

struct event *calloutq;
struct event *freelist;

time_t time = now();
while (1) {
     for (entryp = calloutq; time_to_execute(time, entryp); entryp = entryp->next) {
          if (entryp->priority == HARD_DEADLINE) {
               entryp->execute(entryp->data);
               entryp = free_entry(entryp);
               time = update_calloutq(now(), time);
          }
     }

     if (HIGH_PRIORITY(interrupt_status())) {
          handle_interrupt(HIGH_PRIORITY(interrupt_status()));
          time = update_calloutq(now(), time);
          continue;
     }

     if (calloutq && calloutq->delta <= 0) {
          calloutq->execute(calloutq->data);
          free_entry(calloutq);
          time = update_calloutq(now(), time);
          continue;
     }

     if (LOW_PRIORITY(interrupt_status())) {
          handle_interrupt(LOW_PRIORITY(interrupt_status()));
          time = update_calloutq(now(), time);
          continue;
     }

     if (calloutq) {
          delta = calloutq->delta;
     } else {
          delta = INDEFINITE;
     }

     sleep(delta);      // wakes up only for interrupt or timeout

     time = update_calloutq(now(), time);
}
```

Figure 8: NOS main loop.

The above figure shows the main control loop for the simple multi-rate executive RTOS [27] based on descriptions of designers in the industry [14]. There are two levels of priority for tasks (HARD and SOFT deadline tasks), and two levels of interrupts (HIGH and LOW priority interrupts). All HARD deadline tasks schedule to run now must be completed before any HIGH priority interrupts can be serviced. All HIGH priority interrupts must be completed before any SOFT deadline tasks are serviced, etc.

## 4.3: Benchmarks

To aid in the real-time performance evaluation of these two RTOSs, four benchmarks were created. Each one of these benchmarks executes two separate jobs. The first job grabs a value from an input I/O port. The second job performs some operation on the data gathered by the read job, and then writes to an output I/O port. The four benchmarks are periodic inter-process communication, up sampling, down sampling, and a finite impulse response filter.

### 4.3.1: Periodic Inter-Process Communication

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

### 4.3.2: Up Sampling

With up sampling (UP), the second job runs at a higher frequency than that of the first job. Only a fraction of times that the second job has run will there be any new information. Therefore the second job carries out a basic form of interpolation.

### 4.3.3: Down Sampling

As with up sampling, the frequencies of the first and second jobs in down sampling (DOWN) are not the same. However, as opposed to up sampling, the first job runs at a higher frequency than that of the second job. The second job takes all of the values that have been brought in by the read job since last time that second job has run, averages them, and then outputs that average to the output I/O port.

### 4.3.4: Finite Impulse Response Filter

The finite impulse response (FIR) filter is the most computation intensive of the four benchmarks. The second job runs a 128-tap filter on the data that has been collected by the first task. For each run of the second job, the last 128 values to be inputted by the first job are used in a dot product, and that value is outputted to the I/O port.

### 4.4: Background Load

To add some non-determinism to the evaluation of these two operating systems, and to offer more realistic simulations indicative of real-world systems, two different additional tasks were created. These tasks can be run concurrently with the above listed benchmarks to provide a background load. These two tasks are a periodic control loop and an aperiodic inter-process communication process.

#### 4.4.1: Control Loop

The control loop was created to run in the background at a period of 32ms to simulate the background load that many embedded systems have running while they are performing other tasks, such as a cell phone that has a task that runs every so often to refresh its LCD display. This control loop performs several RAM lookups with an index that is randomly generated.

#### 4.4.2: Aperiodic Inter-Process Communication

The aperiodic inter-process communication (AP-IPC) task is run only when an interrupt is generated by the hardware. The interrupt inter-arrival times obeys a geometric distribution: the emulator generates an interrupt every 100µs with a probability of 0.01, giving an average of 100 interrupts a second.

Because the Echidna RTOS can only have periodically scheduled tasks, a process with the smallest period possible on the Echidna RTOS (1ms) checks to see if an AP-IPC interrupt has occurred. If such is the case, then the AP-IPC code will run. It is important to note that since an interrupt is possible every 100µs, and the interrupt is checked only every 1ms, it is possible for several interrupts to happen before any of them are serviced.

For the NOS, as mentioned above, if processes with higher priorities are constantly running, interrupts from the AP-IPC may not be serviced at all.

## 4.5: The Experiment

For the experiment, all four periodic benchmarks (P-IPC, UP, DOWN, and FIR) were executed on both the Echidna RTOS and NOS. The background load that is run to simulate non-determinism was varied (none, a periodic control loop running every 32ms, an interrupt driven aperiodic IPC, or both the control loop and the aperiodic IPC). For each our the runs, the number of each of those benchmarks ran was varied (1, 2, 4, or 8 tasks), as well as the period at which each of the individual tasks were run (16ms, 8ms, 4ms, 2ms, 1ms, 0.5ms, 0.25ms, 0.125ms, and 0.064ms). For UP and DOWN, the sampling ratios that were run are 2:1, 4:1, and 8:1. The effective cross product of the above variations was run, but it is important to note that, as mentioned above, Echidna cannot schedule a task with a period less than 1ms, therefore periods of less than 1ms were only run on NOS. Also, as mentioned above, each benchmark task is actually two jobs: a reading from I/O job, and a writing to I/O job.

In this study, three things are studied: jitter, delay, and CPU breakdown. Jitter is found by measuring the time between periodic I/O writes, and comparing that time to the goal period of that task. If that inter-arrival time is greater than the period, it means that the task is late, and this will show up as a positive jitter value. If that inter-arrival time is less than the period, then that task is early, and this will show up as a negative jitter value. Delay is measured by keeping track of the time between when an aperiodic

IPC interrupt occurs and when the corresponding I/O write occurs. This represents the response time of the system under varying loads. CPU breakdown is a breakdown of time spent in either the application, kernel, interrupt handler, or idle compilation.

# Chapter 5: Results and Analysis

As mentioned earlier, three things were evaluated during this research: jitter, delay, and CPU breakdown. Jitter is the offset in the goal arrival-time between periodic I/O writes, delay is the time between an aperiodic IPC interrupt and its corresponding I/O write, and CPU breakdown is a breakdown of the time spent in the application, kernel, interrupt handler, or idle sections of computation. This chapter is broken down into four sections. The first section examines the characteristics found in the jitter graphs, the second section looks at the traits found in the delay graphs, the third section goes over the trends in CPU breakdown, and the last section summarizes the characteristics found. Each of the first three sections is further broken down by benchmark (note: delay and CPU breakdown results were obtained for UP and DOWN, but due to their similarity to those of IPC, they will not be shown in this report).

## 5.1: JITTER

As described above, jitter measurements represent the time deltas between successive output seen at the I/O ports for a given task. When more than one task is running, each task is assigned a separate I/O port to write to, enabling the distinction between tasks. On those runs, the jitter information for each of the tasks is combined into a single set of data points.

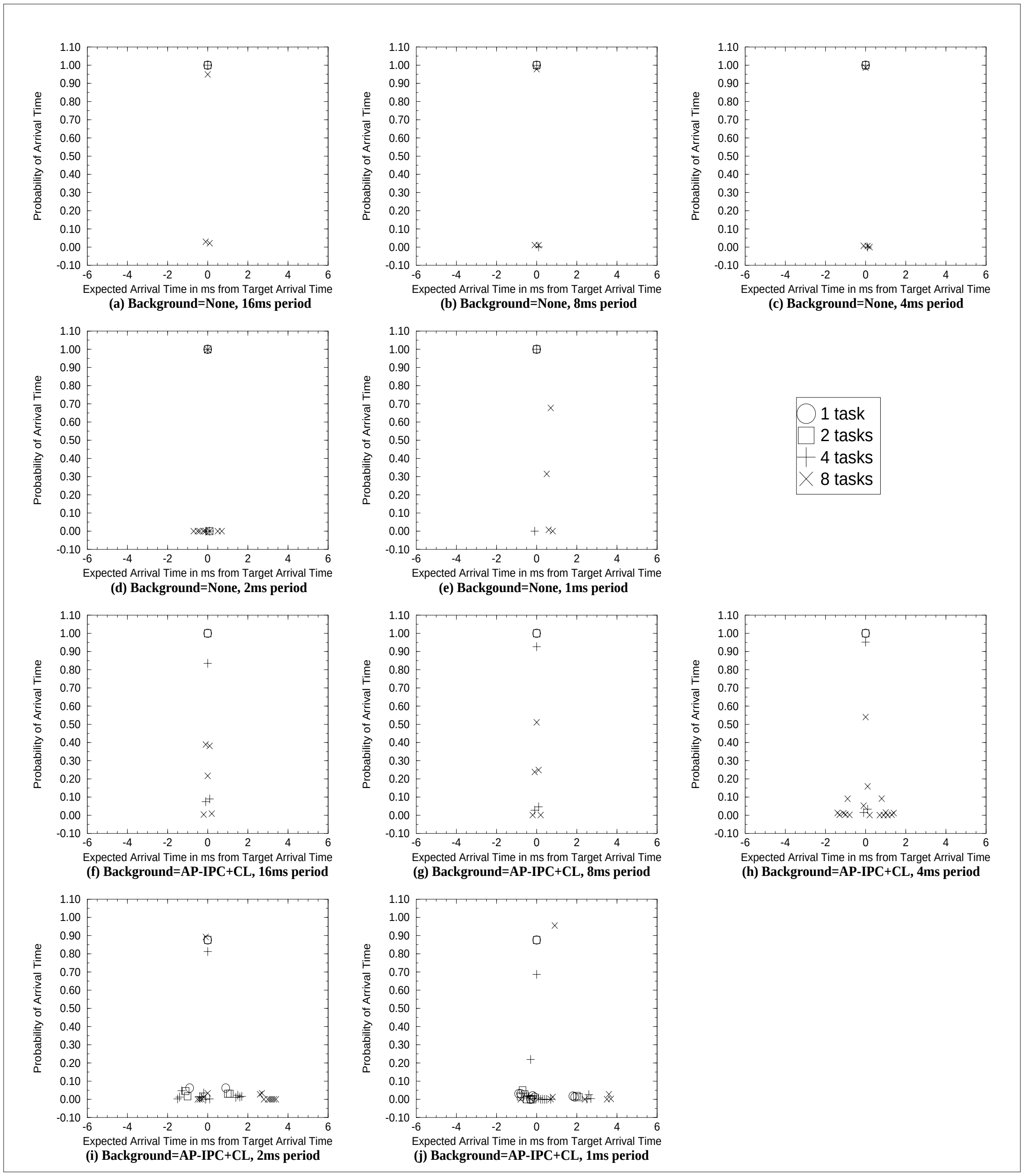
All of the graphs shown are probability density graphs, centered on the desired period.  Negative numbers along the x-axis represent tasks that have run early, and positive numbers represent tasks that have run late, in relation to the previous task.  To maintain readable graphs, only non-zero y-values have been shown, and all of the values have been grouped into 100µs intervals.

### 5.1.1:  Periodic Inter-Process Communication

In this section the Jitter characteristics found in the periodic inter-process communication benchmark runs are examined.  Figures 9 and 10 show the Jitter characteristics for P-IPC.  Figure 9 shows runs on the Echidna RTOS, and Figure 10 shows runs on NOS.  The periodic IPC task represents the simplest possible case of two interacting jobs.  There is no computation performed other than the movement of data between processes; IPC thus represents the smallest workload that a realistic application would schedule on a RTOS.

Figure 9 shows the runs on the Echidna RTOS.  On Figure 9, graphs (a)-(e) represent individual tasks running at periods of 16ms down to 1ms with no background load, while graphs (f)-(j) represent individual tasks running at periods of 16ms down to 1ms with a background load of a control loop running at a period of 32ms and an aperiodic interrupt-driven IPC.

The first five graphs in Figure 9 ((a)-(e)), those runs with no background load, show spikes of data points that are for the most part centered at zero, indicating that the tasks are executing at the given period.  As mentioned above, the heights of the data points indicate the probability of seeing that time delta.  For example, in Figure 9(a), when only 1, 2, or 4 tasks are running, they always execute on time.  However, when 8



(a) Background=None, 16ms period

(b) Background=None, 8ms period

(c) Background=None, 4ms period

(d) Background=None, 2ms period

(e) Backgound=None, 1ms period

(f) Background=AP-IPC+CL, 16ms period

(g) Background=AP-IPC+CL, 8ms period

(h) Background=AP-IPC+CL, 4ms period

(i) Background=AP-IPC+CL, 2ms period

(j) Background=AP-IPC+CL, 1ms period

Figure 9: JITTER probability density graphs for P-IPC on Echidna.
The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(e) represent individual tasks running at periods of 16ms down to 1ms with no background load. Graphs (f)-(j) represent individual tasks running at periods of 16ms down to 1ms with a background load of a control loop running every 32ms, and an aperiodic interrupt driven IPC.

tasks are running, only 95% of the tasks occur on time, while 5% occur 100μs off in either direction. As the period decreases, simulations of 1, 2, and 4 tasks still tend to execute on time, with only 2% falling either 100μs too late or too early for 4 tasks. With 8 tasks, as the period decreases, the values that fall either before or after the expected period start to move away from the origin, (all the way to 1ms off in Figure 9(d)). These values are balanced on both sides of the origin because when a task runs late in Echidna, Echidna schedules the task to run early the next time to make up for the difference and thus return to running on period. In Figure 9(e), with 8 tasks running, the average case no longer falls on 0; instead 70% of the time the task falls at 700μs past the period, meaning that upon reaching this point, the tasks are always executing late almost by an additional period of time. This appears to be the workload level at which the RTOS becomes overloaded.

The second five graphs in Figure 9 ((f)-(j)) were runs with a background load of both the 32ms control process and the aperiodic interrupt driven IPC. Like those with no load, these graphs also show spikes of data mostly centered at zero when only 1 or 2 tasks are running (see Figure 9(f), (g), and (h)). However, as the number of tasks is increased, or the period at which those tasks are running is decreased, several interesting characteristics start to show.

In Figure 9(f), when 8 tasks are running, the data points appear to form a V shape (Other than the two points and +/-200μs that amount to less than 1%, 40% of the data falls at -100μs, 40% falls at +100μs, and remaining 20% falls at 0.). Among all of the IPC graphs running on Echidna, this only appears in this graph, when 8 tasks are running at 16ms each. The reason for this occurrence is the control loop. Because the control loop runs once for every two times the IPC tasks do, half of the time that the IPC tasks are running, the control loop is also scheduled, pushing the average I/O write to 100μs past period instead of at 0. And to counteract that late arrival, Echidna schedules the next task earlier which accounts for the negative peak. This V characteristic only appears when there are 8 tasks (which is 16 jobs: 8 I/O reads, and 8 I/O writes), because Echidna first schedules the control tasks, then each pair of reads and writes. This explains why this characteristic does not appear with only 4 tasks, when only 5% of the tasks are pushed out to 100μs: the additional task overhead with 4 tasks is only large enough to make the last write occasionally late.

As the period decreases to 8ms and 4ms (Figure 9 (g), (h)), when 4 or 8 tasks are running, the peaks are still centered on zero, but data points on both sides of the origin start to increase in probability and distance from the origin, showing that the control loop and the aperiodic interrupt-driven IPC are having more of an affect on I/O write output times.

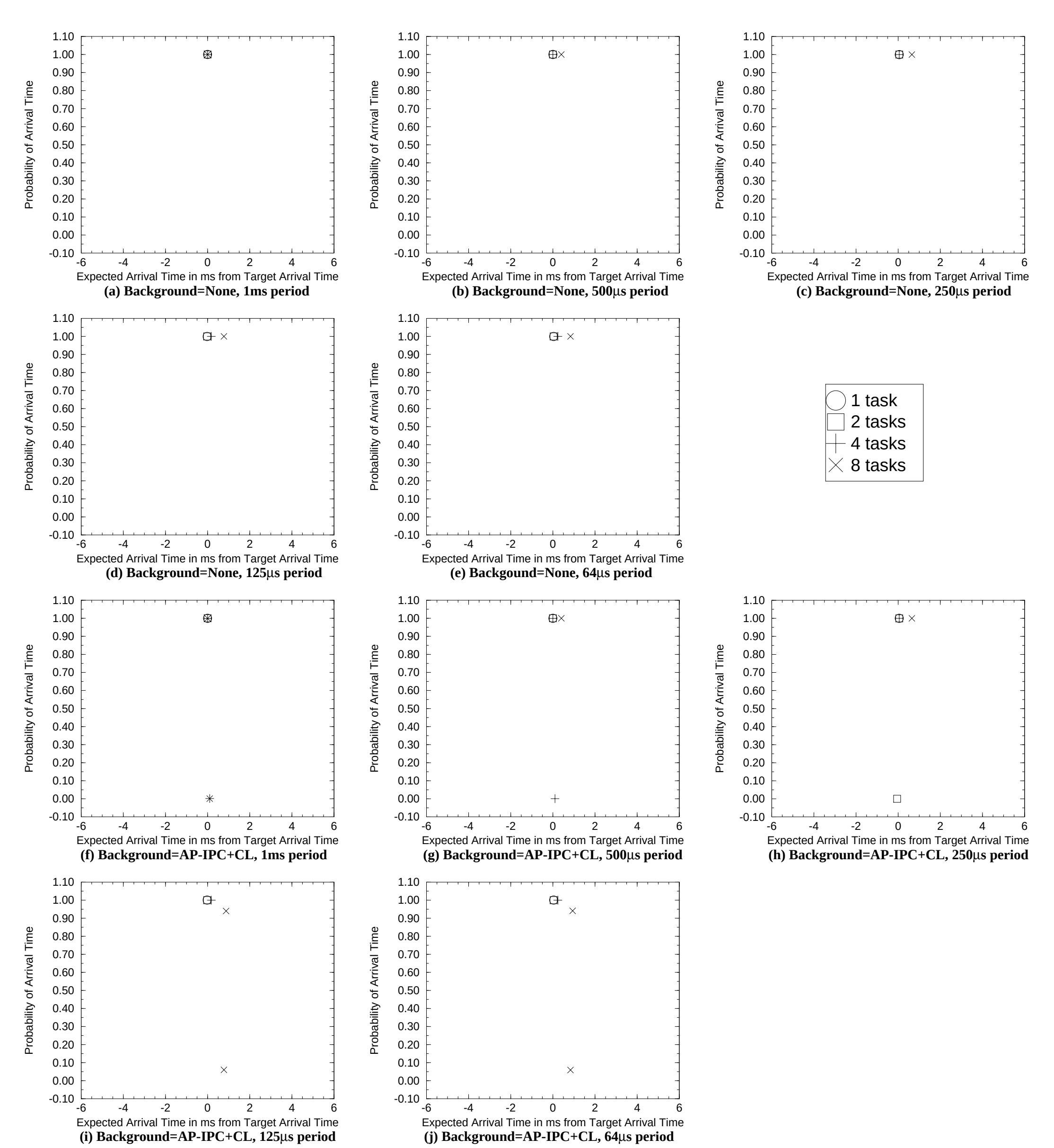
At a period of 2ms (Figure 9 (i)), another interesting trend presents itself. The values are all still centered on zero, however the values are no longer balanced at equal probability on both sides of the origin for runs of 8 tasks (1,2, and 4 task runs are still balanced). Along with the equally balanced data points on each side of the origin, there are also higher probability data points just a fraction of a millisecond early, and several data points much further away from the origin on the positive side of the graph with lesser probability. What this is showing is that the background load is causing tasks to run late, and Echidna is trying to make up for those late runs by scheduling the next task to run early, but it occasionally takes more than one run to fix that late arrival. For example, a task runs 2ms past deadline (causing a delta value of +2ms). Therefore, to fix this problem, Echidna schedules the next run of this task to run the period minus 2ms from now to make up for the late run. However, the task runs late again (this time only 1ms past when it was originally scheduled, causing a delta value of -1ms), and so Echidna has to schedule the next run to also run early to still make up for the remainder of the original 2ms late for the first task (setting the next run to occur at the period minus 1ms from now). This time, the task runs when scheduled, causing a delta value of -1ms. In this example, the data point at -1ms would have a probability twice that of the data point at +2ms. However, if all of the points' delta values are multiplied times their probabilities and summed, the result of 0, meaning that the average case still falls at the origin.

In the final graph (Figure 9(j)), runs of 1, 2, and 4 tasks start to show some of the same characteristics mentioned in the above paragraph for 8 tasks running at 2ms, (higher probability data points of negative deltas close to the origin balanced out by lower probability data points of positive deltas further away from the origin). Also, as was seen in Figure 9(e), for 8 tasks, the values are no longer centered on zero, but at 900µs, with values also landing as far as 3.7 ms off, meaning that during certain runs, the task ran almost three periods late. This is due to the control process running every 32 ms, as well as the aperiodic IPC, which in Echidna is performed by a periodic process that polls the interrupt every 1ms. So every 1ms there is the AP-IPC polling job, 16 jobs associated to the 8 reads and 8 writes, and all of the scheduling overhead involved with maintaining those tasks. And every 32 times that this occurs, a control loop is also scheduled to run.

It appears that for IPC running on Echidna, when there is any background load, 1 to 2 tasks running at 8 to 16ms periods is the limit in order to always have tasks running on time. If it only matters to run tasks on time for the average case, 1, 2, or 4 tasks at speeds of 16-1ms, or 8 tasks from 16-2ms can be run. Beyond that, the RTOS becomes overloaded and unpredictable.

Figure 10 shows the runs on NOS. On Figure 10, graphs (a)-(e) represent individual tasks running at periods of 1ms down to 0.064ms with no background load, while graphs (f)-(j) represent individual tasks running at periods of 1ms down to 0.064ms with a background load of a control loop running at a period of 32ms and an aperiodic interrupt driven IPC. Runs with periods of greater than 1ms are not shown because all of the runs at those periods, both with and without background load, always run on period.

For the first five graphs in Figure 10 ((a)-(e)), those runs with no background load, regardless of the number of tasks or what the delta of those points are, the probability arrival time for all of the data points fall at 1. The only trait that varies is where the data points fall. For 1 or 2 tasks and for all of the periods from 1ms to 0.064ms, the tasks run on time. Only when 4 or 8 tasks are run do the tasks start running late, and even the 4 task runs still meet their period down to 125µs. Unlike Echidna, NOS does not attempt to fix late arrival times, so a task that occurs late will still be rescheduled with its original period. However, NOS would not benefit from this correction, because for periods of 250µs or slower with runs of 8 tasks, the delta is always greater than the period, so this delay is not caused by application computation time; instead it is caused by scheduler overhead, which is independent of the period at which the tasks are run, but instead is a function of the number of tasks that are being run.



(a) Background=None, 1ms period

(b) Background=None, 500µs period

(c) Background=None, 250µs period

(d) Background=None, 125µs period

(e) Backgound=None, 64µs period

(f) Background=AP-IPC+CL, 1ms period

(g) Background=AP-IPC+CL, 500µs period

(h) Background=AP-IPC+CL, 250µs period

(i) Background=AP-IPC+CL, 125µs period

(j) Background=AP-IPC+CL, 64µs period

Figure 10: JITTER probability density graphs for P-IPC on NOS.

The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(e) represent individual tasks running at periods of 1ms down to 0.064ms with no background load. Graphs (f)-(j) represent individual tasks running at periods of 1ms down to 0.064ms with a background load of a control loop running at a period of 32ms, and an aperiodic interrupt driven IPC. Graphs with periods of higher than 1ms have been omitted since they always meet their deadlines.

By examining the data points for 8 tasks running at decreasing periods, the minimum period that 8 tasks can be run at before the kernel has maxed out can be determined. In Figure 10(b), the data point for 8 tasks is 400µs past the goal period of 500µs. In Figure 10(c), 8 tasks are running 650µs past their goal period of 250µs. In Figure 10(d), 8 tasks are 800µs off, and in Figure 10(e), 8 tasks are 800µs off. So by adding the goal period to the amount of time that the tasks are late in each case gives the minimum period that 8 tasks can successfully run at on NOS as ~900µs.

The second five graphs in Figure 10 ((f)-(j)), those runs with a background load of both a 32ms control loop and the aperiodic interrupt driven IPC, match almost identically with those with no load with the exception of a data point occasionally falling 100µs away from the origin with a probability of less than 0.01. Also, the data points for 8 tasks have shifted a little further from the origin, changing the minimum period from being ~900µs to falling between 900µs and 1ms.

This shows that the control loop and the AP-IPC have very little effect on the arrival time of our IPC runs on NOS. This makes sense for two reasons. The first is that the control loop runs at a period of 32 ms, while our processes are running at periods of 1ms to 64µs, so the control loop has very little effect. Second, since in NOS the AP-IPC interrupt is only serviced after all of the tasks that need to run have run, the AP-IPC has very little effect.

So it appears that for IPC runs on NOS, the limiting factor is not how fast the tasks are run, but instead it is the number of tasks that are run at that speed. So, if a design specification calls for 8 tasks to be run at a period of 500µs each, it would be more optimal to run four tasks at 250µs instead, and all of those tasks will run on time, where 8 tasks at 500µs will always run 400µs late.

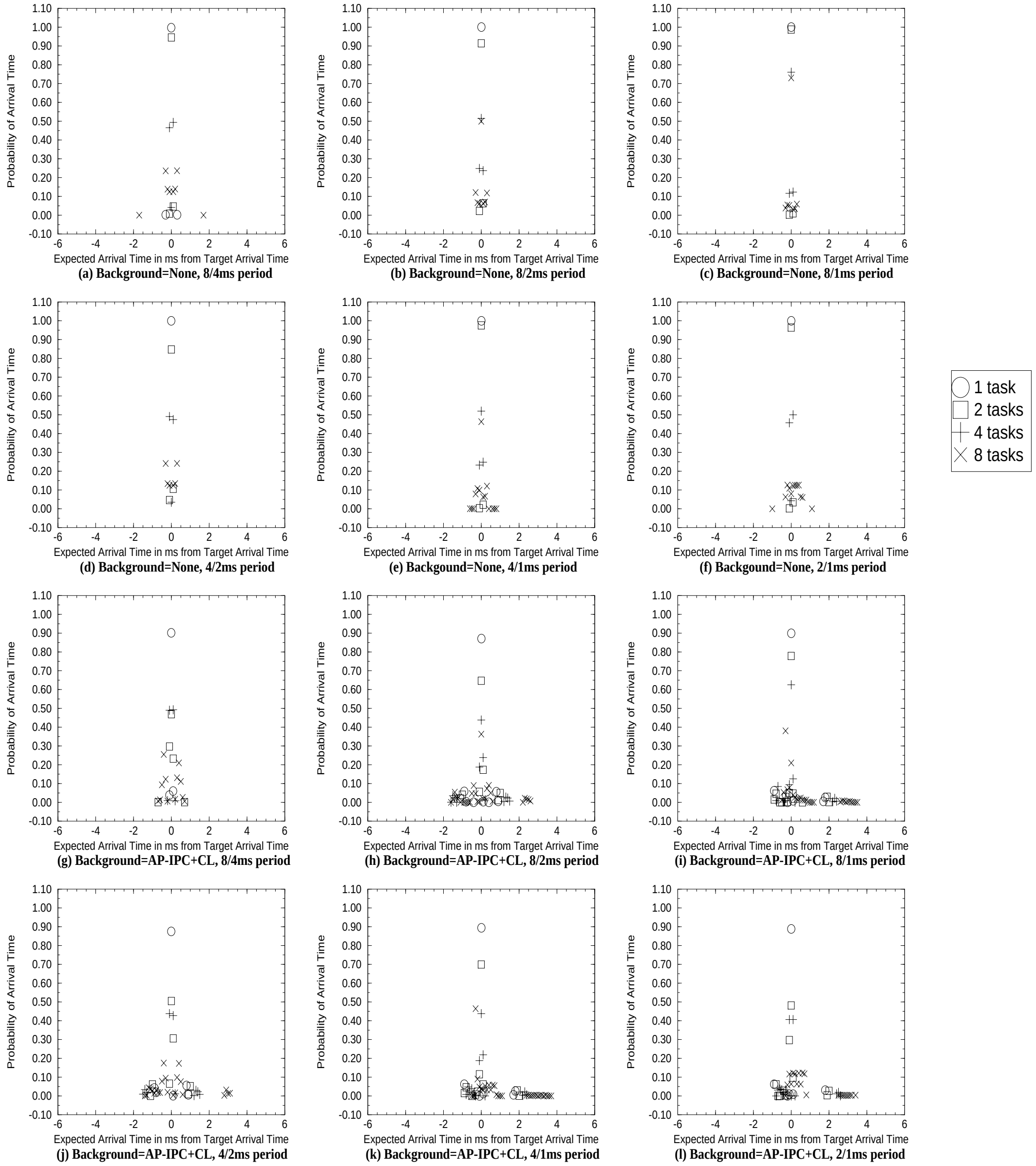
Comparing Echidna to NOS, we find that workload level that Echidna begins to fail at is around a period of 1ms with 8 tasks running, while NOS, which is a much simpler, bare-bones scheduler, can operate successfully even below the 1ms limit of Echinda for 4 tasks or less.

### 5.1.2: Up Sampling

This section examines the Jitter characteristics found in the up sampling benchmark runs. Figures 11 and 12 show the Jitter characteristics for UP. Figure 11 shows runs on the Echidna RTOS, and Figure 12 shows runs on NOS. The up sampling benchmark is not as simple a benchmark as IPC. The first job still only reads an I/O input value, but the second job performs some basic computation in addition to writing to the I/O output port. With up sampling, the second job runs at a faster period than that of the first job. The second job takes the last input value brought in and spreads that value amongst all of the write jobs until the next input job is set to run.

Figure 11 shows the runs on the Echidna RTOS. On Figure 11, graphs (a)-(f) represent individual tasks running at increasing loads (8/4, 8/2, 4/2, 8/1, 4/1, and 2/1, all periods in ms) with no background load, while graphs (g)-(l) represent individual tasks running at increasing loads (8/4, 8/2, 4/2, 8/1, 4/1, and 2/1, all periods in ms) with a background load of a control loop running at a period of 32ms and an aperiodic interrupt driven IPC.

The first six graphs in Figure 11 ((a)-(f)) are the runs that operate with no background load. However, these graphs show characteristics that we saw with the IPC runs



**(a) Background=None, 8/4ms period**

**(b) Background=None, 8/2ms period**

**(c) Background=None, 8/1ms period**

**(d) Background=None, 4/2ms period**

**(e) Backgound=None, 4/1ms period**

**(f) Backgound=None, 2/1ms period**

**(g) Background=AP-IPC+CL, 8/4ms period**

**(h) Background=AP-IPC+CL, 8/2ms period**

**(i) Background=AP-IPC+CL, 8/1ms period**

**(j) Background=AP-IPC+CL, 4/2ms period**

**(k) Background=AP-IPC+CL, 4/1ms period**

**(l) Background=AP-IPC+CL, 2/1ms period**

Figure 11: JITTER probability density graphs for UP on Echidna.

The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(f) represent individual tasks running at increasing loads (8/4, 8/2, 4/2, 8/1, 4/1, and 2/1, all in ms) with no background load. Graphs (g)-(l) represent individual tasks running at increasing loads (8/4, 8/2, 4/2, 8/1, 4/1, and 2/1, all in ms) with a background load of a control loop running at a period of 32ms, and an aperiodic interrupt driven IPC.

on Echidna with background load because they are multi-rate tasks. For example, Figure 11(a) shows the run where the first job is running at a period of 8ms while the write job runs at a rate of 4ms. Half of the time when a write job wants to run, it has to also compete with a read job, and half the time it doesn't, just like when the IPC write job occasionally had to deal with a control loop, thus the V shape.

Like the IPC graphs for Echidna, these graphs also show spikes for data points, centered around zero, until the RTOS becomes overloaded (see the 8 tasks data point in Figure 7(f)). Those values that do not fall on the origin are balanced, meaning that each point on the negative portion of the graph is matched by a corresponding data point on the positive portion of the graph. An interesting observation is that those runs with the read/write ratio of 2:1 (Figure 11 (a), (c), and (f)) show V characteristics for runs of 4 to 8 tasks, like those mentioned in the IPC section. These occur for same reasons as listed above; half of the time, the writes have to deal with an accompanied read, pushing the execution time for those last few writes later, pushing the average away from 0, causing the positive half of the V, and the Echidna correction causes the negative half. This is not seen in the 8/2, 8/1, or 4/1 graphs (Figure 11 (b), (d), and (e)) because the read jobs happen only a quarter or an eighth as often as the write does.

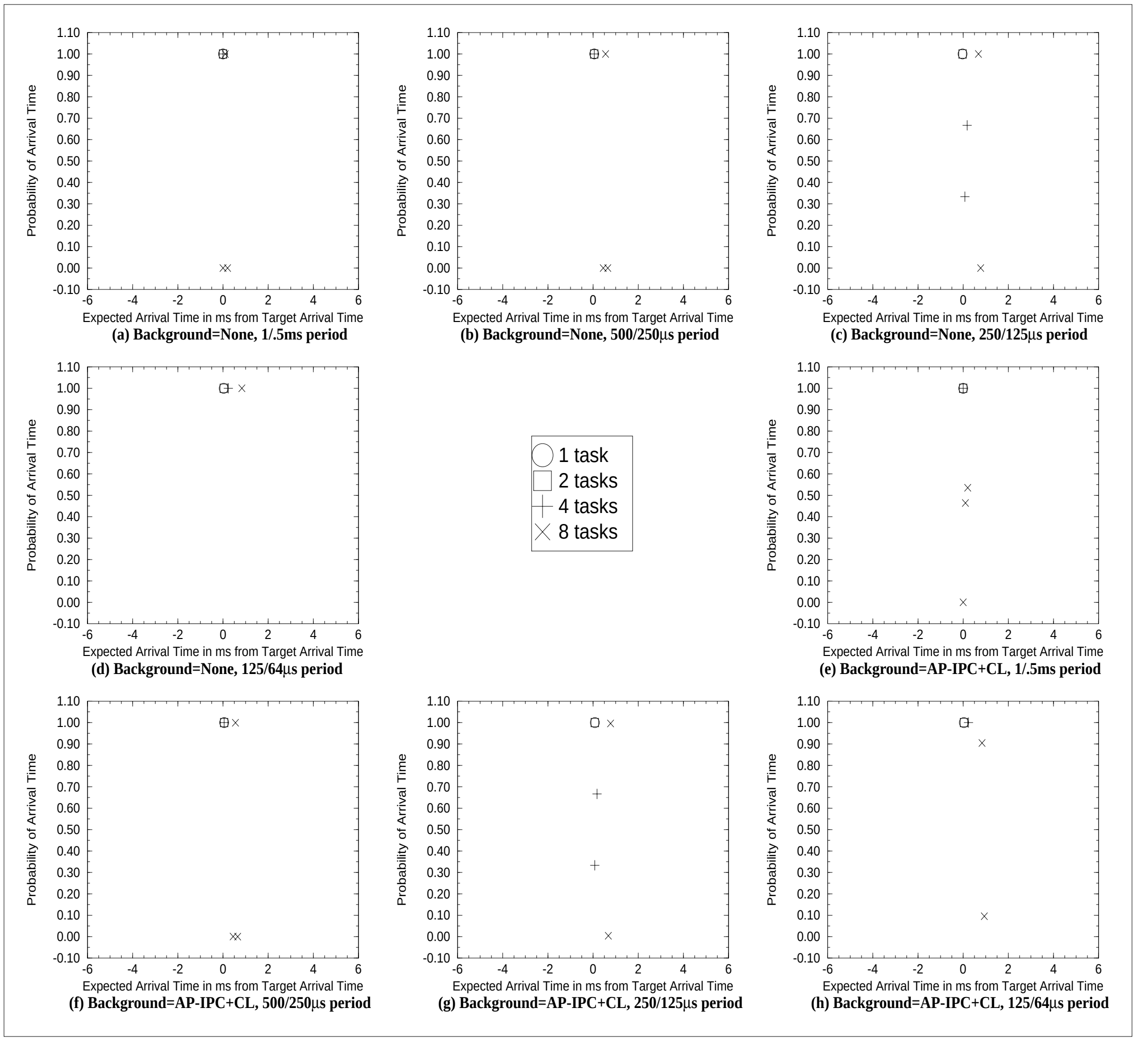
As before, as load increases, more and more of the data values move away from the origin, until the point at which the system is overloaded. This occurs in Figure 11(f) with 8 tasks; the values are no longer centered at 0, they are instead centered at a value of 100-200μs.

The second six graphs in Figure 11 ((g)-(l)), those runs with a background load of both the 32ms control process and the aperiodic interrupt driven IPC, show the same characteristics as those with no background load, except the values tend to spread even further away from the origin and with a greater probability. Also, values that are not balanced around the origin can be seen. This occurs for the same reason as was described in section 5.1.1, that as the speed of the jobs running is increased, the effect of the control loop as well as the aperiodic interrupt IPC becomes more prevalent and more difficult to recover from.

An important difference to note between the runs with background load and those without background load is that, when there is no background load, when only 1 task is running, that task always runs on time, regardless of period or the ratio of the read and write job. However, when the background load is added, that is no longer the case. For all of the runs with background load, the runs with 1 task never reach the goal period more than 90% of the time. For the runs of UP on Echidna with a background load, tasks are no longer guaranteed to run on period, they only run on period on average.

Figure 12 shows the UP runs on NOS. In Figure 12, graphs (a)-(d) represent tasks running at increasing loads (1/.5, .5/.25, .25/.125, and .125/.065, all periods in ms), with no background load, while graphs (e)-(h) represent tasks running at increasing loads (1/.5, .5/.25, .25/.125, and .125/.064, all periods in ms) with a background load of a control loop running at a period 32ms and an aperiodic interrupt driven IPC. Runs with periods greater than 1/.5 are not shown because all of the runs at those periods, both with and without the background load, always run on period.

The first four graphs in Figure 12 ((a)-(d)), those runs with no background load, for the most part, fall at the origin. In Figure 12(a), all data points are at the origin with a probability of 1 with the exception of the run with 8 tasks. The reason that the 8 task run



(a) Background=None, 1/.5ms period

(b) Background=None, 500/250µs period

(c) Background=None, 250/125µs period

(d) Background=None, 125/64µs period

(e) Background=AP-IPC+CL, 1/.5ms period

(f) Background=AP-IPC+CL, 500/250µs period

(g) Background=AP-IPC+CL, 250/125µs period

(h) Background=AP-IPC+CL, 125/64µs period

Figure 12: JITTER probability density graphs for UP on NOS.
The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(d) represent individual tasks running at increasing loads (1/.5, .5/.25, .25/.125, and .125/.64 all in ms) with no background load. Graphs (e)-(l) represent individual tasks running at increasing loads (1/.5, .5/.25, .25/.125, and .125/.65 all in ms) with a background load of a control loop running at a period of 32ms, and an aperiodic interrupt driven IPC. Graphs with periods of higher than 1/.5ms have been omitted since they always meet their deadlines.

is off slightly is due only to the number of tasks running, and like mentioned above, occasionally the read jobs push the last write back a bit, but only very rarely. When we decrease the period, the values once again start moving away from the origin. As mentioned above, the tasks are running late because too many tasks are trying to be scheduled within a short amount of time and the kernel cannot handle it. This is shown by the fact that only runs of 4 to 8 tasks are running off period. Runs of 1 or 2 tasks are right on period, down to 64µs.

As shown above, by examining the data points for 8 tasks running at decreasing periods, the minimum period that 8 tasks can be run at can be determined. From Figure 12(a), the 8 tasks data point is late by 100µs. In Figure 12(b), the 8 tasks data point is late by 550µs. With Figure 8(c), the data point is off by 700µs, and in Figure 12(d) the data point is off by 800µs. Thus, the minimum periods for UP on NOS is between 1.8ms/900µs and 1.2ms/600µs.

The second four graphs in Figure 12 ((e)-(h)), those runs with a background load of a 32ms control loop and the aperiodic interrupt driven IPC, match almost identically with those with no load, with occasionally a small shift of 100µs for a part of the data (as seen in Figures 12 (e), (h)), caused by the control loop. As mentioned above, this is both because the control loop runs much more slowly than these jobs, and that the AP-IPC is not serviced unless there are no waiting jobs. Like with the IPC data, the 8 task minimum is pushed out100µs by the background load, making the minimum period between 2/1ms and 1.4ms/700µs.

As seen with IPC, it again appears for UP that with NOS, the limiting factor is not how fast we are running the tasks, but instead it is the number of tasks that are running. Also seen with IPC, when comparing Echidna to NOS, it is found that the workload level that Echidna begins to fail is at 2/1ms periods with 8 tasks running, while

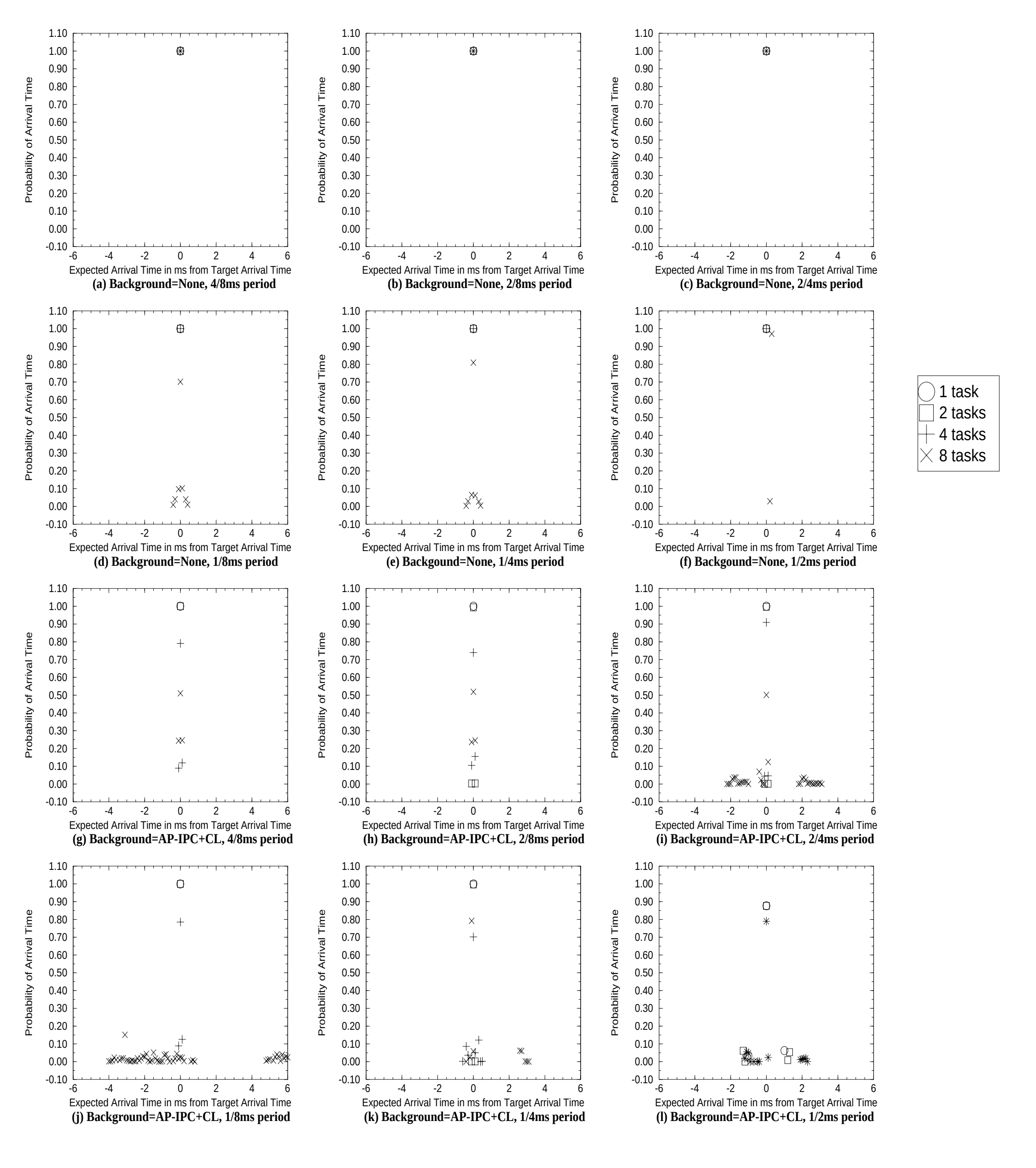
NOS appears be able to run well below that.  However, as opposed to IPC, UP has a much greater variance in the data points than IPC did on Echidna.

### 5.1.3:  Down Sampling

This section examines the Jitter characteristics found in the down sampling benchmark runs.  Figures 13 and 14 show the Jitter characteristics for DOWN.  Figure 13 shows runs on the Echidna RTOS, and Figure 14 shows runs on NOS.  Like up sampling, the down sampling benchmark is not a simple benchmark like IPC, but it is not as complicated a benchmark as FIR.  While the first job still only reads an I/O input value, the second job does perform some basic computation.  Since the second job runs at a slower period than the first job, it takes all of the input values brought in since the last run of the write job and averages them, and then outputs that value to the write I/O port.

Figure 13 shows the runs on the Echidna RTOS.  On Figure 13, graphs (a)-(f) represent individual tasks running at  increasing loads (4/8, 2/8, 2/4, 1/8, 1/4, and 1/2, all periods in ms) with no background load, while graphs (g)-(l) represent individual tasks running at increasing loads (4/8, 2/8, 2/4, 1/8, 1/4, and 1/2, all periods in ms) with a background load of a control loop running at a period of 32ms and an aperiodic interrupt driven IPC.

The first six graphs in Figure 13 ((a)-(f)) are the runs that operate with no background load.  For the most part, all of the spikes are centering around the origin, and the only runs that are not 100% at zero are those of 8 tasks.  And those runs of 8 tasks that are having their output move away from the origin, while not being a problem of kernel overhead (see the 8 tasks data point in  Figure 13(f)), are Figure 13 (d) and (e).  The reason for this is because unlike the up sampling write job, which performs a simple



(a) Background=None, 4/8ms period

(b) Background=None, 2/8ms period

(c) Background=None, 2/4ms period

(d) Background=None, 1/8ms period

(e) Backgound=None, 1/4ms period

(f) Backgound=None, 1/2ms period

(g) Background=AP-IPC+CL, 4/8ms period

(h) Background=AP-IPC+CL, 2/8ms period

(i) Background=AP-IPC+CL, 2/4ms period

(j) Background=AP-IPC+CL, 1/8ms period

(k) Background=AP-IPC+CL, 1/4ms period

(l) Background=AP-IPC+CL, 1/2ms period

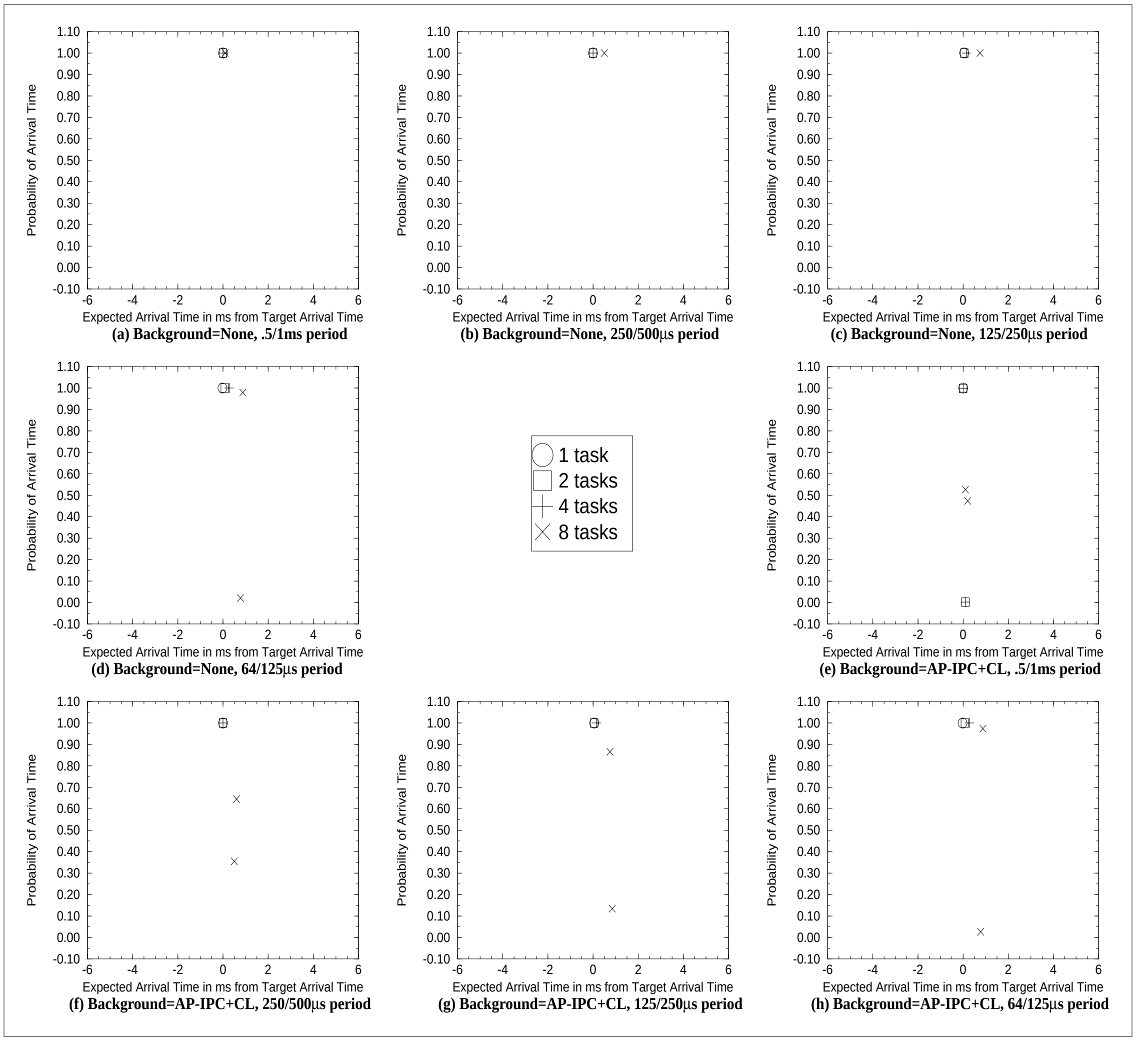
Figure 13: JITTER probability density graphs for DOWN on Echidna. The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(f) represent individual tasks running at increasing loads (4/8, 2/8, 2/4, 1/8, 1/4, and 1/2, all in ms) with no background load. Graphs (g)-(l) represent individual tasks running at increasing loads (4/8, 2/8, 2/4, 1/8, 1/4, and 1/2, all in ms) with a background load of a control loop running at a period of 32ms, and an aperiodic interrupt driven IPC.

divide, the down sampling write job executes a loop that sums all of the values inputted since its last iteration, and then performs a divide.  Also, this loop runs longer if the ratio between the read and the write job is greater; as it is in the 1/8ms and 1/4ms graphs.

The second six graphs in Figure 13 ((g)-(l)), those with a background load of both the 32ms control process and the aperiodic interrupt driven IPC, show characteristics as those with no background load, except some of data points start to spread away from the origin.  We also can see the added trend of values that are not balanced around the origin (Figure 13 (i), (j), (k), and (l)), as mentioned above, and for the same reason as mentioned above, that as you increase the speeds at which the jobs are running, the effect of the control loop as well as the aperiodic interrupt IPC becomes more prevalent and also becomes more difficult to recover from.  This is very clear in Figure 13(j), because in this instance, with 8 tasks, each one of the write tasks performs the most amount of work possible(due to the 1:8 ratio), and the control loop runs once for every four times that the write job does.

Figure 14 shows the DOWN runs on NOS.  On Figure 14, graphs (a)-(d) represent tasks running at increasing loads (.5/1, .25/.5, .125/.25, and .065/.125, all periods in ms), with no background load, while graphs (e)-(h) represent tasks running at increasing loads (.5/1, .25/.5, .125/.25, and .065/.125, all periods in ms) with a background load of a control loop running at a period of 32ms and an aperiodic interrupt driven IPC.  Runs with periods greater than .5/1ms are not shown because all of the runs at those periods, both with and without the background load, always run on period.

The first four graphs in Figure 14 ((a)-(d)), those runs with no background load, when the system is not being overloaded, all of the data points fall at the origin with a



(a) Background=None, .5/1ms period

(b) Background=None, 250/500μs period

(c) Background=None, 125/250μs period

(d) Background=None, 64/125μs period

(e) Background=AP-IPC+CL, .5/1ms period

(f) Background=AP-IPC+CL, 250/500μs period

(g) Background=AP-IPC+CL, 125/250μs period

(h) Background=AP-IPC+CL, 64/125μs period

Figure 14: JITTER probability density graphs for DOWN on NOS.
The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(d) represent individual tasks running at increasing loads (.5/1, .25/.5, .125/.25, and .64/.125 all in ms) with no background load. Graphs (e)-(l) represent individual tasks running at increasing loads (.5/1, .25/.5, .125/.25, and .64/.125 all in ms) with a background load of a control loop running at a period of 32ms, and an aperiodic interrupt driven IPC. Graphs with periods of higher than .5/1ms have been omitted since they always meet their deadlines.

probability of 1. However, when the system begins to be overloaded (2 tasks at 64/125μs, 4 tasks at 125/250μs, and 8 tasks at 250/500μs), the values slowly start drifting to the positive side of the origin.  Once again the minimum period for 8 tasks running can be derived by examining the data in the graphs.  The minimums derived from these figures is 1ms/500µs.

The second four graphs in Figure 14 ((e)-(h)), those runs with a background load of a 32ms control loop and the aperiodic interrupt driven IPC, match almost identically with those with no load, with the exception of an occasional small shift of 100µs for a part of the data (as seen in Figures 14 (e), (f), and  (g)), caused by the control loop when 8 tasks are running.  As mentioned above, this lack of effect of the background load is both because the control loop runs much slower than these jobs and because the AP-IPC is not serviced unless there are no waiting jobs.  Again, by examining these graphs, the minimum period can be derived to be 1.2ms/600µs.

As seen with UP and IPC, it appears that for NOS the limiting factor is not how fast we are running the tasks, but instead it is the number of those tasks that we are running.  Also seen with UP, when comparing Echidna to NOS it is found that the workload level that Echidna begins to fail at is 1/2ms periods with 8 tasks, where NOS appears be able to run normally below that limit.  However, as opposed to UP, the amount of variance in the data values with Echidna is more dependent on the ratio of periods than the periods that the jobs are running at.  This is easily explained by the fact that the second job’s workload is proportional to the number of times the first job has run since the last time the second job has run.

### 5.1.4:  Finite Impulse Response Filter

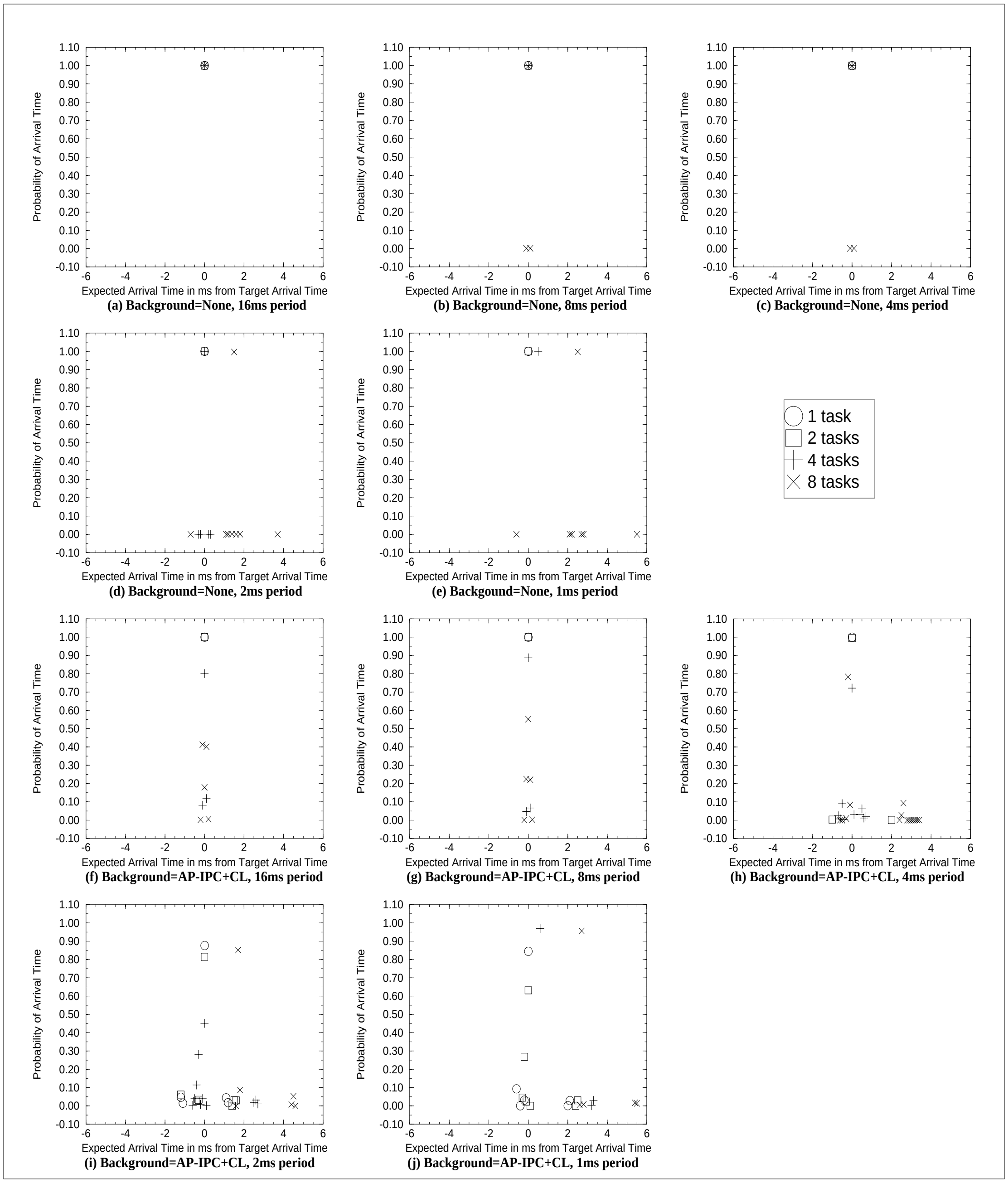
This section examines the Jitter characteristics found in the finite impulse response filter benchmark runs.  Figures 15 and 16 show the Jitter characteristics for

FIR. Figure 15 shows runs on the Echidna RTOS, and Figure 16 shows runs on NOS. Of the four benchmarks that are being used for this experiment, FIR is the most computationally intensive. The first job reads an I/O input value, and the second job runs a 128-tap filter on the data that has been collected by the first job in the past. For each run of the second job, the last 128 values inputted by the first job are used in a dot product and that value is outputted to the I/O port.

Figure 15 shows the runs on the Echidna RTOS. On Figure 15, graphs (a)-(e) represent individual tasks running at periods of periods of 16ms down to 1ms with no background load, while graphs (f)-(j) represent individual tasks running at periods of 16ms down to 1ms with a background load of a control loop running at a period of 32ms and an aperiodic interrupt driven IPC.

The first five graphs in Figure 15 ((a)-(e)), those runs with no background load, show spikes of data points, for the most part centered at the origin, indicating that the tasks are executing at the given period. As the period decreases, data points start to show up on both sides of the origin, starting with 8 tasks in Figure 15(b), then with both 4 and 8 tasks in Figure 15(c). However, as opposed to all of the other benchmarks, in Figure 15(d), 8 task runs are no longer centered at 0 for the 2ms period. And in Figure 15(e) both 8 and 4 task runs are no longer centered at 0. This shows that now, in addition to delays caused by the kernel being maxed out due to scheduling issues, the runs are also experiencing problems due to applications interfering with each other because of the computation-intensive nature of the FIR benchmark.

The second five graphs in Figure 15 ((f)-(j)), those runs with a background load of both the 32ms control process and the aperiodic interrupt driven IPC, like those with



(a) Background=None, 16ms period

(b) Background=None, 8ms period

(c) Background=None, 4ms period

(d) Background=None, 2ms period

(e) Backgound=None, 1ms period

(f) Background=AP-IPC+CL, 16ms period

(g) Background=AP-IPC+CL, 8ms period

(h) Background=AP-IPC+CL, 4ms period

(i) Background=AP-IPC+CL, 2ms period

(j) Background=AP-IPC+CL, 1ms period

Figure 15: JITTER probability density graphs for FIR on Echidna.

The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(e) represent individual tasks running at periods of 16ms down to 1ms with no background load. Graphs (f)-(j) represent individual tasks running at periods of 16ms down to 1ms with a background load of a control loop running at a period of 32ms, and an aperiodic interrupt driven IPC.

no load, also show spikes of data, again mostly centered at zero, when only 1 or 2 tasks are running (see Figure 15(f),(g), and (h)). However, as number of tasks is increased, or the period of those tasks is decreased, several interesting characteristics start to appear.

In Figure 15(f), when 8 tasks are running, the data points appear to form a V shape (Other than the two points and +/-200µs that amount to less than 1%, 40% of the data falls at -100µs, 40% falls at +100µs, and remaining 20% falls at 0.). Among all of the FIR graphs running on Echidna, this only appears in this graph, when 8 tasks are running at 16ms each. As mentioned in the IPC section, the reason for this occurrence is the control loop. Because the control loop runs once for every two times the FIR tasks do, half of the times that the FIR tasks are running, the control loop is also scheduled, pushing the average I/O write to 100µs past period instead of at 0. And to counteract that late arrival, Echidna schedules the next task earlier, which accounts for the negative peak.

As the period decreases, when 4 or 8 tasks running, the peaks are still centered on zero, but data points on both sides of 0 start to both increase in probability as well as distance from the origin, showing that the control loop and the aperiodic interrupt-driven IPC are having more of an affect on I/O write output times.

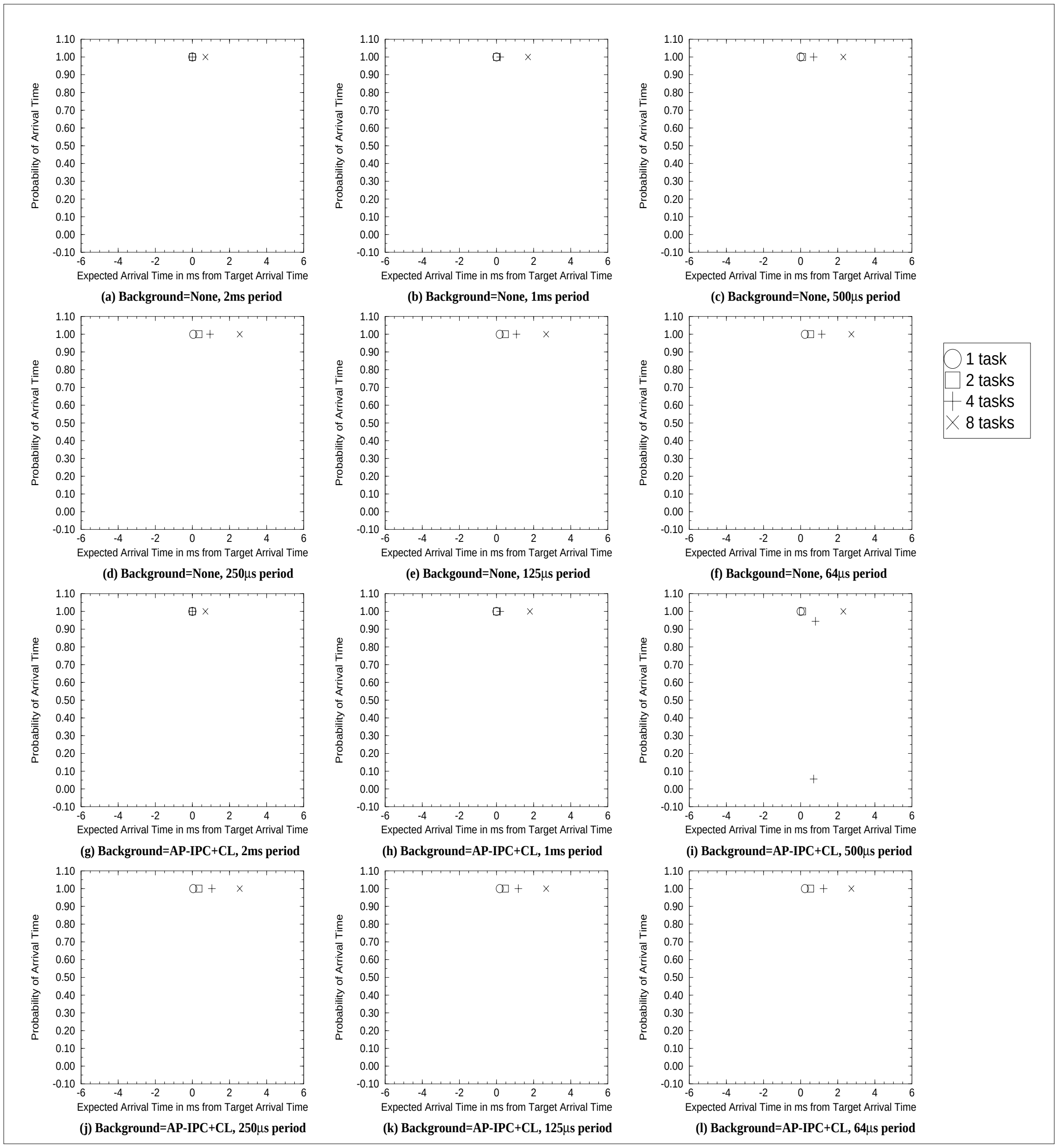
When the period of the tasks approaches 4ms (Figure 15 (h)), the trait explained in the IPC section of values no longer being balanced at equal probability on both sides of the origin is once again seen. The values are all still centered on zero (until the period decreases to 2ms and 1ms with 4 or 8 tasks running), but when a task executes late, it takes more than a single early scheduled task to return to the set period.

With the 2ms and 1ms graphs, (Figure 15 (i), (j)), 1 and 2 tasks continue to show the same traits as listed above, but first the 8 tasks running at a 2ms period run and the 4 tasks running at 1ms period run are no longer centered at the origin, but at 500µs for the 4 task run, and at 2.5ms for the 8 task run. As mentioned above, this shows that in addition to delays caused by the kernel being maxed out, the runs also experiencing the problem of applications interfering with each other due to the computation intensive nature of the FIR benchmark.

It appears that with FIR running on Echidna, when any background load is running, only the one or two task runs are achieving their goal period, and only at periods of 16ms to 8ms. If the number of tasks is increased, or the period at which they run is decreased, the task only arrives on period on average, with 4 tasks down to a 2ms period, and with 8ms down to a 4ms period. Beyond this point, the system has become overloaded, and will never achieve its goal period.

Figure 16 shows the runs on NOS. On Figure 16, graphs (a)-(f) represent individual tasks running at periods of periods of 2ms down to 0.064ms with no background load, while graphs (g)-(l) represent individual tasks running at periods of 2ms down to 0.064ms with a background load of a control loop running at a period of 32ms and an aperiodic interrupt driven IPC. Runs with periods of greater than 2ms are not shown because all of the runs at those periods, both with and without background load, always run on period (with the exceptions of 8 tasks being 0.7ms off in graphs with and without background load).

The first six graphs in Figure 16 ((a)-(f)), those runs with no background load, regardless of the number of tasks, or what the delta of those points are, the probability



(a) Background=None, 2ms period

(b) Background=None, 1ms period

(c) Background=None, 500µs period

(d) Background=None, 250µs period

(e) Backgound=None, 125µs period

(f) Backgound=None, 64µs period

(g) Background=AP-IPC+CL, 2ms period

(h) Background=AP-IPC+CL, 1ms period

(i) Background=AP-IPC+CL, 500µs period

(j) Background=AP-IPC+CL, 250µs period

(k) Background=AP-IPC+CL, 125µs period

(l) Background=AP-IPC+CL, 64µs period

Figure 16: JITTER probability density graphs for FIR on NOS.
The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks. Graphs (a)-(f) represent individual tasks running at periods of 2ms down to 0.064ms with no background load. Graphs (g)-(l) represent individual tasks running at periods of 2ms down to 0.064ms with a background load of a control loop running at a period of 32ms, and an aperiodic interrupt driven IPC. Graphs with periods of higher than 2ms have been omitted since they always meet their deadlines.

arrival time for all of the data points fall at 1. The only trait that varies is where the data points fall. For a 2ms period, 8 task runs are late. For a 1ms period, only runs of 1 or 2 tasks run on period, while 4 and 8 tasks are late. For a 500μs period, only runs of 1 task run on period, and all other runs with periods of less than 500μs all of the tasks run late. From these graphs, the minimum frequency for 1, 2, 4, and 8 tasks can be determined. For 8 tasks, the minimum period that can be run is 2.7ms. For 4 tasks, the minimum period that can be run is 1.2ms. For 2 tasks, the minimum period that can be run is 600μs. And for 1 task, the minimum period is 300μs.

The second six graphs in Figure 16 ((g)-(l)), those runs with a background load of both a 32ms control loop and the aperiodic interrupt driven IPC, match almost identically with those with no load, with the exception of Figure 16(g), in which the data point are clustered around one of the 100μs boundaries, with 90% falling on one side, and 10% falling on the other. Once again this shows that the control loop and the AP-IPC have very little effect on the jitter time of the FIR runs. This makes sense for two reasons. The first is that the control loop runs at a period of 32ms, while the FIR tasks are running at speeds of 1Kz to 16KHz, so the control loop has very little effect. Second, since in the NOS the AP-IPC interrupt is only serviced after all of the tasks that need to run now have run, the AP-IPC has very little effect.

It appears that for FIR runs on NOS, the minimum period that a task can run is a function of both the number of tasks running as well as the amount of computation that FIR needs to perform in order to run.

Comparing Echidna with NOS, it can be seen that the average case from both operating systems is the same. The reason for this is that the minimum period possible is no longer determined by scheduler overhead but by how many iterations of FIR can be run in a specific period of time (theorectical limit).

### 5.1.5: JITTER Summary

For the first three benchmarks (IPC, UP, and DOWN), each with relatively small computational workloads, it appears that the scheduler overhead for multiple jobs is the most limiting factor. When running on Echidna, in order to always run on time, the period must be limited to 16 to 8ms. To only run on period on the average case, any periods with any number of tasks can be run, with the exception of 8 tasks running at a period of 1ms, which is when the output will become unpredictable and almost always late. NOS on the other hand, is capable of going below a 1ms period and still arrive on time, with 4 tasks or less.

With FIR, we found that the limiting factor with the workload level was a combination of both application time and kernel overhead. Both Echidna and NOS begin to experience late tasks when 8 tasks are running at a period of 2ms.

## 5.2: DELAY

The delay measurements represent the time between an external interrupt generating an aperiodic IPC and the corresponding output to an I/O port from the responding thread. Therefore, this delay measures the response time of the system in terms of when the first reaction to an interrupt could take place.

Neither Echidna nor NOS handles interrupts preemptively; both use a polling technique. The difference is that Echidna has a periodic thread that is scheduled to run every 1ms to check for an interrupt, and if one is found, respond to it; NOS checks to see if an interrupt has occurred only when the system is idle: If the system is either busy or overloaded, an interrupt will be ignored, perhaps indefinitely, unless the system returns to an idle state and checks to see if an interrupt is waiting.

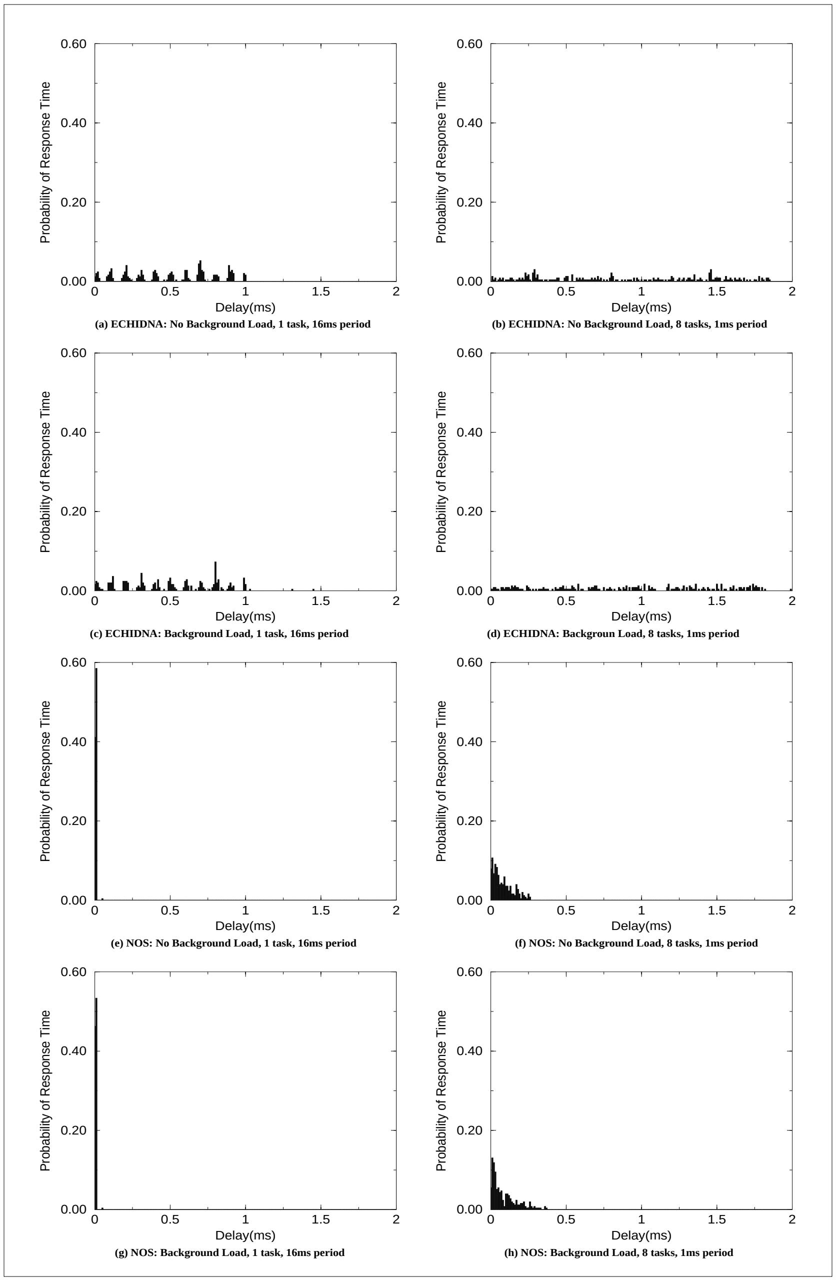
An important difference to note between the values obtained for the Jitter graphs and the values obtained for the Delay graphs is that the values on the delay graphs are grouped into intervals of 10μs, instead of 100μs like in the Jitter graphs. This is done because in many of the Delay graphs, all of the values would fit into the first 100μs, but would give several points in a 10μs interval graph.

In this section, we take a look at the effect of the AP-IPC on the IPC benchmarks and the FIR benchmarks. Both the UP and the DOWN benchmarks have results similar to those of P-IPC.

### 5.2.1: Periodic Inter-Process Communication

This section examines the Delay characteristics found in the periodic inter-process communication benchmark runs. Figure 17 shows the graphs selected to show the trends in delay for IPC. Eight graphs were chosen, four from runs on Echidna, and four from runs on NOS. For each of the two above, the first two runs were run without the control loop running in the background, and the second two runs were run with the control loop. For each of these sets of two, the run with the lightest load (1 task running at a period of 16ms), and the run with the heavier load (8 tasks running at periods of 1 ms each) are shown.

Figure 17(a) shows the run when Echidna is the least loaded and when there is no control loop running. The figure shows that the delay probability times are equally spread out from 0 to 1ms. This make sense. Since this system is not overloaded, the interrupts occur with equal probability over time, and Echidna checks for interrupts



(a) ECHIDNA: No Background Load, 1 task, 16ms period

(b) ECHIDNA: No Background Load, 8 tasks, 1ms period

(c) ECHIDNA: Background Load, 1 task, 16ms period

(d) ECHIDNA: Backgroun Load, 8 tasks, 1ms period

(e) NOS: No Background Load, 1 task, 16ms period

(f) NOS: No Background Load, 8 tasks, 1ms period

(g) NOS: Background Load, 1 task, 16ms period

(h) NOS: Background Load, 8 tasks, 1ms period

Figure 17: Delay probability density graphs for P-IPC.

The x-axis represents time between an interrupt being generated by an I/O device and the corresponding output to an I/O port of the responding thread. The y-axis indicates the probability of each delta. (a)-(d) are graphs from runs on Echidna. (a) and (b) were run without the control loop, while (c) and (d) were run with a background load of a control loop running at a period of 32ms. (e)-(h) are graphs from runs on NOS. (e) and (f) were run without the control loop, while (g) and (h) were run with a background load of a control loop running at a period of 32ms. (a), (c), (e), and (g) were runs with only 1 task running at a period of 16ms (minimal load), while (b), (d), (f), and (h) were runs with 8 tasks running each at a period of 1ms.(Note: x-axis is 0 to 2ms).

every 1ms. Figure 17(b) shows when Echidna is more heavily loaded. In this graph, the response times span from 0 to almost 2ms. These measurements show that the process checking for interrupts is not running every 1ms, instead it is running much more slowly, almost twice as slow. This is caused by the overhead involved in managing 8 tasks (16 jobs) and the job checking the interrupt, at periods of 1ms each. Figure 17 (c) and (d) show the same situations as Figure 17 (a) and (b), except that a control process is also running. The only noticeable difference in these runs is that there is an occasional point past 1 and 2ms (not shown) respectively. These are caused by interrupts that occur during one of the control loops runs, and the response time is affected.

Figure 17(e) shows the run when NOS is the least loaded and when there is no control loop running. From the figure it can be seen that 100% of the interrupts are handled within the first 15µs (note, the first bin is centered at 0, and contains any values that occur from 0-4µs). This makes sense since this is the least loaded run on NOS, and the interrupt is serviced during idle times. Figure 13(f) shows the run when NOS is loaded with 8 tasks each running at 1ms. In this graph it can be seen that the values of response time vary from 0 to 275µs. This additional time is caused by the wait for the NOS to finish all currently scheduled tasks, so that it can check for interrupts. Figure 17 (g) and (h) show the same situations that Figure 17 (e) and (f) show, except that a control loop is also running at the same time. The only noticeable difference is that there is a very small bar at 50µs in Figure 17(g), and the values in Figure 17(f) continue out to 375µs. These additions are caused by the running of the control loop, and the additional time that it takes up in the scheduler before the system becomes idle and interrupts are polled.

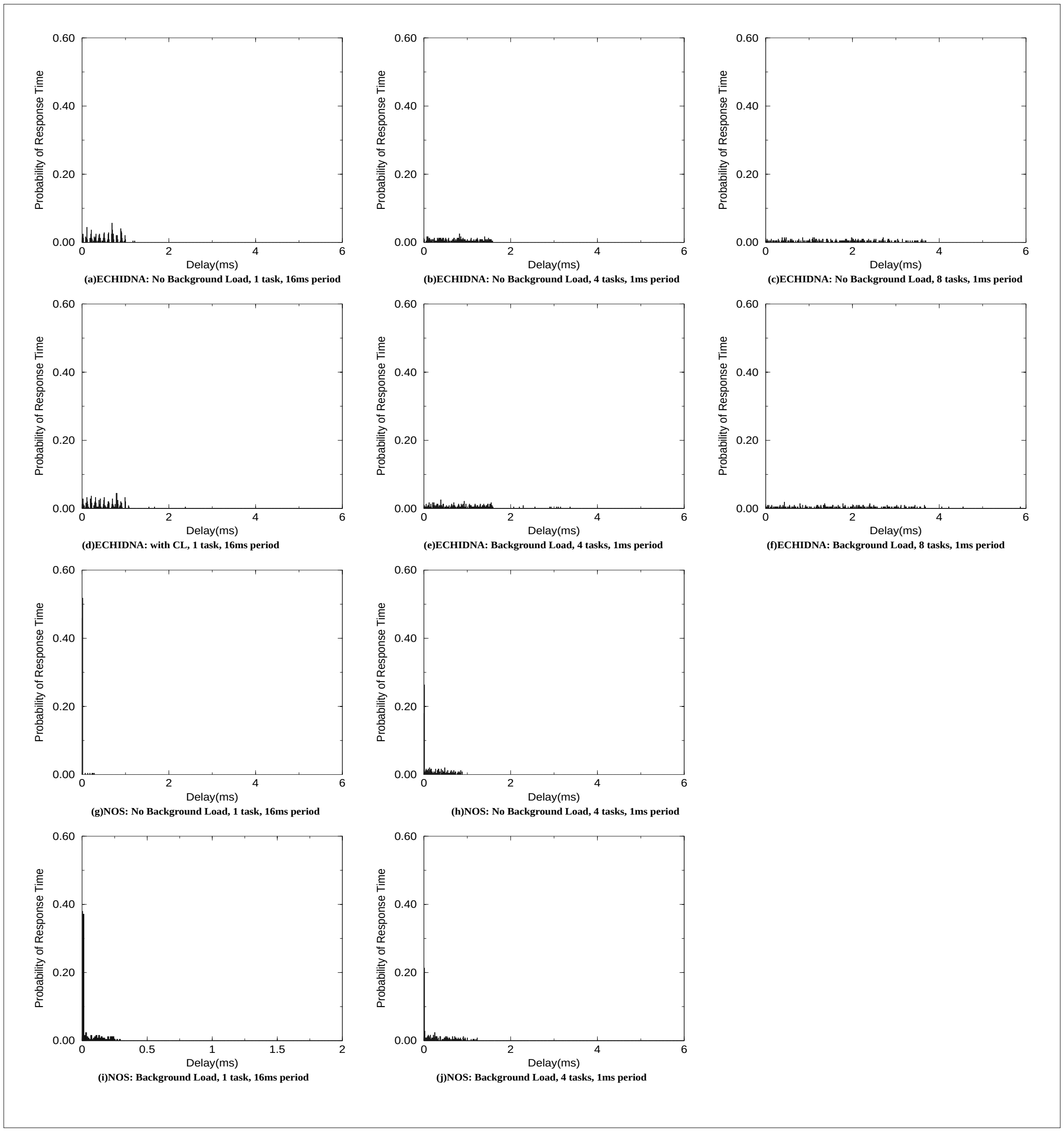
An additional point to note is that as NOS becomes overloaded interrupts are not checked.  For runs of IPC on NOS for periods less than 1ms, no interrupts at all were serviced for runs of 8 tasks at 500μs and faster, for runs of 4 tasks at 250μs and faster, and for runs of 2 tasks at 125μs and slower.  Only the runs of 1 task were successfully able to service the I/O interrupt down to 64μs.

As seen in Figure 17, for Echidna, when the load is low, an interrupt is serviced within 1ms (since the interrupt polling routine runs at 1ms), and when the load is high, it can take twice as long to service an interrupt.  The control loop seems to have a very minimal affect on the response time.  For NOS, when the load is low, the response time is almost immediate.  When the load is greater, the response is slower, but it is still quicker than that of Echidna.  When the system is overloaded NOS does not respond to the interrupt at all.

### 5.2.2:  Finite Impulse Response Filter

This section examines the Delay characteristics found in the finite impulse response filter benchmark runs.  Figure 18 shows the graphs selected to show the trends in delay for FIR.  FIR is more computation-intensive than any of the other benchmarks, so it is expected to have the worst delay times.  Ten graphs have been chosen, six from runs on Echidna, four from runs on NOS.  For Echidna, there is a light load run (1 task running at 16ms), a heavy load run (4 tasks running at 1ms), and an overloaded run (8tasks running at 1ms) both with and without the control loop.  For NOS, only the light and medium loads with and without the control loop are shown.

Figure 18(a) shows the run when Echidna is the least loaded and when there is no control loop running.  From this figure, it can be seen that the delay probability times are



(a)ECHIDNA: No Background Load, 1 task, 16ms period

(b)ECHIDNA: No Background Load, 4 tasks, 1ms period

(c)ECHIDNA: No Background Load, 8 tasks, 1ms period

(d)ECHIDNA: with CL, 1 task, 16ms period

(e)ECHIDNA: Background Load, 4 tasks, 1ms period

(f)ECHIDNA: Background Load, 8 tasks, 1ms period

(g)NOS: No Background Load, 1 task, 16ms period

(h)NOS: No Background Load, 4 tasks, 1ms period

(i)NOS: Background Load, 1 task, 16ms period

(j)NOS: Background Load, 4 tasks, 1ms period

Figure 18: Delay probability density graphs for FIR.

The x-axis represents time between an interrupt being generated by an I/O device and the corresponding output to an I/O port of the responding thread. The y-axis indicates the probability of each delta. (a)-(f) are graphs from runs on Echidna. (a), (b) and (c) were run without the control loop, while (d), (e), and (f) were run with a background load of a control loop running at a period of 32ms. (g)-(j) are graphs from runs on NOS. (g) and (h) were run without the control loop, while (i) and (j) were run with a background load of a control loop running at a period of 32ms. (a), (d), (g), and (i) were runs with only 1 task running at a period of 16ms (minimal load), (b), (e), (h), and (j) were runs with 4 tasks at a period of 1ms, and (c) and (f) were runs with 8 tasks running each at a period of 1ms. Graphs of NOS with 8 tasks running at a period of 1ms are not shown because no interrupts are serviced during these runs. (Note: x-axis is 0 to 6ms).

equally spread out from 0 to 1ms.  This make sense, since this system is not overloaded, the interrupts occur with equal probability over time, and Echidna checks for interrupts every 1ms.  Figure 18(b) shows when Echidna is sustaining a heavy load.  In this graph, the response times span from 0 to almost 1.75ms.  These measurements show that the process checking for interrupts is not running every 1ms, instead it is running much slower.  Figure 18(c) shows when the system is overloaded.  As seen in the graph, the response times go from 0 to almost 4ms.  This means that the interrupt polling task that is supposed to be running every 1ms is instead only running every 4ms.  This is caused by the overloaded state of the system, when the system is trying to run more applications than possible.  Figure 18 (d), (e), and (f) show the same situations as Figure 18 (a), (b) and (c), except that a control process is also running.  The difference between these two sets of  graphs is that in the control graphs, there are some additional points beyond the areas listed above for the runs without the control loop.  These are caused by interrupts that occur during one of the control loop runs.

Figure 18(g) shows the run when NOS is the least loaded and when there is no control loop running.  This figure shows that 95% of the interrupts are handled within the first 15µs.  However, unlike the other benchmarks, even at this light load, we have delay values all the way until 300µs.  This is due to the more computational nature of FIR: it takes longer to run the job.  Figure 18(h) shows the run when NOS is heavily loaded with 4 tasks running at a period of 1ms.  This graph shows the response time varying from 0 to 1ms, and the values appear more frequently at higher delays than that of Figure 18(g).  This additional time is caused by the wait for the NOS RTOS to finish all currently scheduled benchmark tasks, and become idle, when it checks for interrupts.

Figure 18 (i) and (j) show the same situations that Figure 18 (h) and (g) show, except that a control loop is also running at the same time. In Figure 18(i), the response times vary from 0 to 250µs, and in Figure 18(j), the values vary from 0 to 1.25ms.

The graphs that were chosen for this section were chosen differently than the other sections because the graphs of 8 tasks running at periods of 1ms on NOS contained no information. This is because at that point the system is overloaded, and the system is never idle, and therefore never checks for interrupts. For FIR on NOS, no interrupts at all were serviced for runs of 8 tasks at 1ms and faster, for runs of 2 and 4 tasks at 500µs and faster, and for runs of 1 task at 250µs and slower.

### 5.2.3: DELAY Summary

Both delay sections show similar traits. For Echidna, when the load is low, an interrupt is serviced within 1ms (since the interrupt polling routine runs at 1ms), and when the load is high, it can take between two to four times as long to service an interrupt. The control loop seems to have minimal affect on this response time. For NOS, when the load is low, the response time is almost immediate. When the load is greater, the response is slower, but it is still quicker than that of Echidna When the system is overloaded, the response to interrupts in NOS is non-existent.

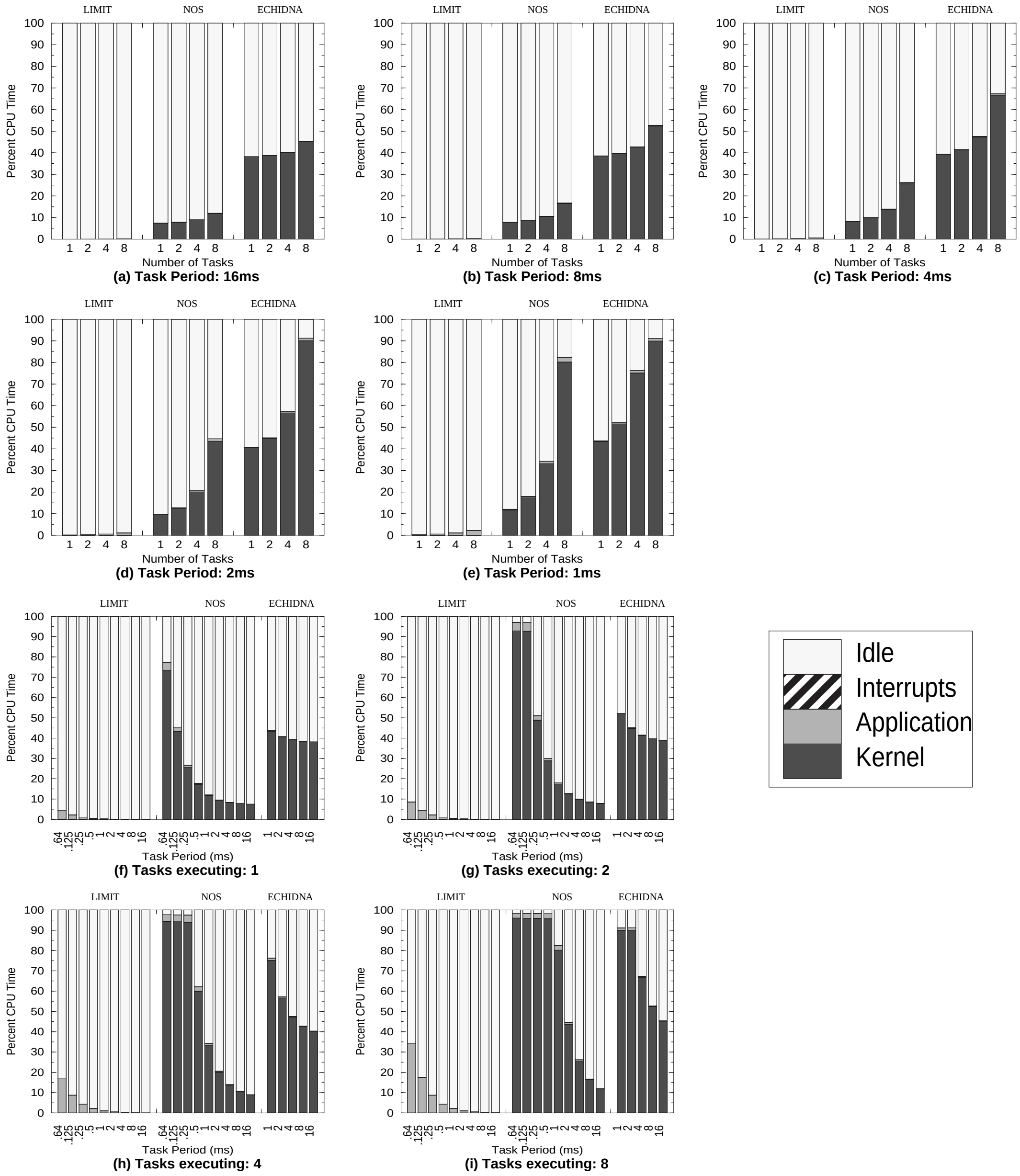
## 5.3: CPU Breakdown

The CPU breakdown graphs show the amount of time spent by the system in kernel, application, interrupt handling, and idle portions of the code. Two different types of graphs are shown: (1) Those with a constant task period, while varying the number of tasks that are being run; and (2) Those with a constant number of tasks running, while varying the frequency that those tasks are running at. On each graph, there are three distinct groups of data. The first group is the calculated theoretical limit of a system running the application code. This group only contains application and idle segments, and the values are calculated by multiplying the number of tasks to be run at that frequency by the time it takes to run a single task. The second group of bar graphs show the CPU breakdowns for the runs on NOS. The final group of bar graphs are the CPU breakdowns for those runs on the Echidna RTOS. As with the Delay graphs, only the results from simulations of IPC and FIR are shown, as the results from UP and DOWN fall in between them.

### 5.3.1: Periodic Inter-Process Communication

This section examines the CPU breakdown characteristics found in the periodic inter-process communication benchmark runs. The periodic IPC benchmark represents the simplest case of two interacting jobs. There is no computation performed other than the movement of data, and this represents the least amount of workload that an application would schedule on an RTOS, therefore it is most likely to exhibit the highest kernel overhead. Figure 19 contains all of the graphs used to evaluate CPU breakdown for the IPC runs. All of the graphs shown are executions with background load, for there is little difference between those runs with and without background load.

The first five graphs (Figure 19 (a)-(e)), show the runs in which, for each graph, the period that the tasks are running at is constant, and the number tasks run at that period is varied. The second four graphs (Figure 19 (f)-(i)), show the runs in which, for each graph, the number of tasks is constant, and the period is varied. The first thing seen is that the percentage of time spent in application execution is very small for all of the runs. For NOS, kernel overhead is 95% of the non-idle CPU time, and the Echidna



(a) Task Period: 16ms

(b) Task Period: 8ms

(c) Task Period: 4ms

(d) Task Period: 2ms

(e) Task Period: 1ms

(f) Tasks executing: 1

(g) Tasks executing: 2

(h) Tasks executing: 4

(i) Tasks executing: 8

Figure 19: CPU-BREAKDOWN graphs for P-IPC.

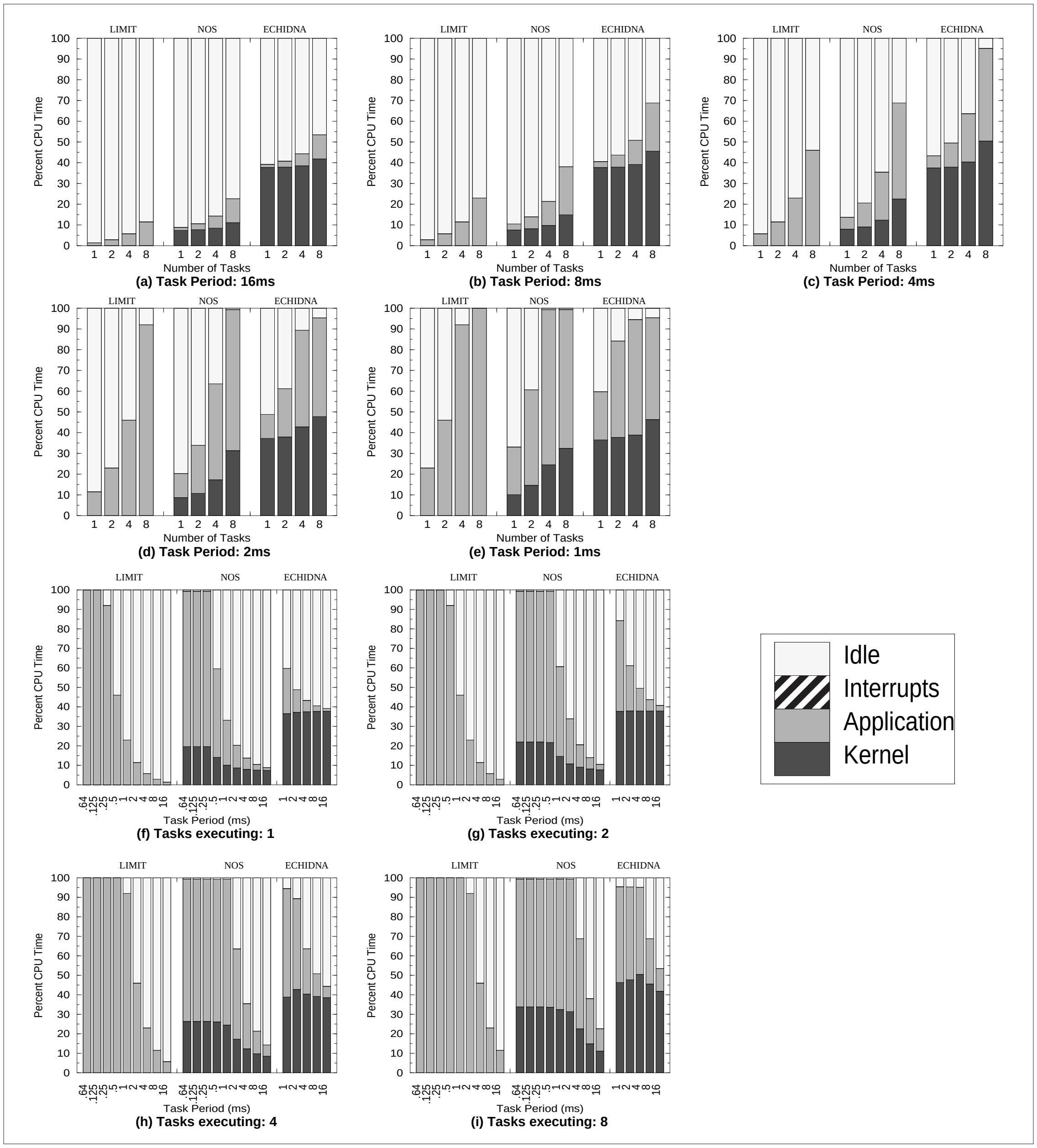
Along the x-axis there are three distinct groups of bar graphs. The first group are calculations of theoretical limit of user application execution time versus idle time. The second group are the CPU breakdowns of NOS runs. The final group are the CPU breakdowns of Echidna runs. In graphs (a)-(e) the x-axis represents increasing number of tasks for periods 16ms down to 1ms. In graphs (f)-(i) the x-axis represents decreasing periods for 1, 2, 4, and 8 tasks. The y-axis represents the total CPU breakdowns for how much time is spent executing kernel code, executing user application code, handling interrupts, and sitting idle. Idle includes both time sleeping as well as some loop overhead in the main loop and parts of the timekeeping code for Echidna.

kernel time dominates even more.  One can also see where the runs reach their limit of execution before they become overloaded.  For Echidna, that limit is at 8 tasks running at 1ms (as seen in Figure 19(i)); for in the transition from 2 to 1ms, no more additional application time is spent.  For NOS, the limit is at 500µs for 8tasks (see Figure 19(i)), at 250µs for 4 tasks (see 19(h)), and the limit for 2 tasks is 125µs.  Runs of 1 task on NOS do not appear to reach a limit, even at a period of 64µs.

### 5.3.2:  Finite Impulse Response Filter

This section examines the CPU breakdown characteristics found in the finite impulse response filter benchmark runs.  Of the four benchmarks FIR is the most computation-intensive.  Figure 20 contains all of the graphs used to evaluate CPU breakdown for the FIR runs.  All of the graphs shown are executions with background load, for there is little difference between those runs with and without background load.

The first five graphs (Figure 20 (a)-(e)), show the runs in which, for each graph, the period that the tasks are running at is constant, and the number of tasks run at that period is varied.  The second four graphs (Figure 20 (f)-(i)), show the runs in which, for each graph, the number of tasks is constant, and the period is varied.  The first thing that is seen is that in contrast to the IPC runs, the time spent in the application code is much greater than anything else once the number of tasks has increased to 4 or 8, and once the period of those tasks drops below 4ms.  For NOS, the kernel overhead has dropped down to 20-50% of the non-idle CPU time while Echidna takes 50-95% of the non-idle time.  In the FIR runs, the system becomes overloaded much sooner than that of the other benchmarks, and in this case it is a combination of too much application work and kernel scheduling overhead that is causing the overload.  For Echidna, that limit for 8 tasks



(a) Task Period: 16ms

(b) Task Period: 8ms

(c) Task Period: 4ms

(d) Task Period: 2ms

(e) Task Period: 1ms

(f) Tasks executing: 1

(g) Tasks executing: 2

(h) Tasks executing: 4

(i) Tasks executing: 8

Figure 20: CPU-BREAKDOWN graphs for FIR.

Along the x-axis there are three distinct groups of bar graphs. The first group are calculations of theoretical limit of user application execution time versus idle time. The second group are the CPU breakdowns of NOS runs. The final group are the CPU breakdowns of Echidna runs. In graphs (a)-(e) the x-axis represents increasing number of tasks for periods 16ms down to 1ms. In graphs (f)-(i) the x-axis represents decreasing periods for 1, 2, 4, and 8 tasks. The y-axis represents the total CPU breakdowns for how much time is spent executing kernel code, executing user application code, handling interrupts, and sitting idle. Idle includes both time sleeping as well as some loop overhead in the main loop and parts of the timekeeping code for Echidna.

running is at 4ms (as seen in Figure 20(i)) and for 4 tasks running it is at 2ms (see Figure 20(h)). For NOS, those limits are at 2ms for 8tasks (see Figure 20(i)), at 1ms for 4 tasks (see Figure 20(h)), at 500µs for 2 tasks (see Figure 20(g)), and at 250µs for runs of a single task (see Figure 20(f)). Clearly, at these faster periods, the limit is caused not by kernel overhead alone, but by a combination of limit of how much one can run on a given system and the maximum number of tasks that a RTOS can schedule in a given period of time.

### 5.3.5: CPU Breakdown Summary

Several things were seen from the CPU breakdown graphs. The first was not really talked about, but interrupt handling overhead is insignificant. This makes sense because both of the RTOSs that we are looking at are polled systems, so no state needs to be saved or restored, so interrupt overhead is non-existent.

The second thing that was seen was that on the systems where the applications are not computationally intensive, as mentioned in the jitter and delay sections, it is cheaper to run fewer applications at a faster period than to run more applications at a slower period.

Finally, once the system is finally overloaded, it gravitates to an optimal ratio of kernel versus user time. This ratio is a function of benchmark and configuration. This characteristic is best seen in Figure 20(h) and (i).

## 5.4: Analysis Summary

This experiment has evaluated three aspects of real-time behavior: jitter, delay, and CPU breakdown. With Jitter, it was observed that as the number of tasks is increased, the amount of scheduling overhead incurred is increased, more so with

Echidna than NOS. With IPC, UP, and DOWN, the limit for Echidna is reached when 8 tasks are running at periods of 1ms(or 2/1ms, or 1/2ms), while NOS can continue to run on time for periods lower than that of Echidna's limit. For FIR, scheduling overhead is only one factor in calculating the limit, and both NOS and Echidna reach a limit of 8 tasks running at 2ms, or 4 tasks running at 1ms. For Echidna runs, if there is any background load, the data points start to move away from the origin, but the average run is still on time. For NOS, the background has very little affect.

With delay, when a system has a light load, both Echidna and NOS are able to service the interrupt immediately (within 1ms is as fast as Echidna can check the interrupt). However, if the system is running with a significant load, Echidna can take up to four times as long to service the interrupt, and NOS has the possibility of dropping the interrupt entirely. Addition of the control loop has very little affect on these characteristics.

With CPU breakdown, several things were seen. Interrupt handling overhead was insignificant because both RTOSs use polling. On the systems where applications are not computationally intensive it is cheaper to run fewer applications at a faster period than to run more applications at a slower period. And once the system is overloaded it gravitates to an optimal ratio of kernel versus user time.

# Chapter 6: Conclusions

This report has presented a method of using full-system emulation to evaluate the real-time performance of an embedded system. An embedded architecture emulator was created, using the C programming language, that emulates the Motorola M-CORE embedded processor down to the register level and is accurate to within 100 cycles per million as compared to actual hardware. With tests and experiments run on this emulator, the goal of this report was to show that this method can be successfully used in the evaluation of embedded systems.

A study of non-preemptive real-time operating systems was presented, focusing on Echidna, a small, public domain RTOS, and comparing it to NOS, a bare-bones scheduler that represents the performance limit for non-preemptive RTOSs. Three different real-time performance characteristics were measured: JITTER, DELAY, and CPU USAGE.

With Jitter, it was observed that as the number of tasks was increased, the amount of scheduling overhead incurred was increased, more so with Echidna than NOS. With IPC, UP, and DOWN, the limit for Echidna is reached when 8 tasks are running at periods of 1ms(or 2/1ms, or 1/2ms), while NOS can continue to run on time for periods lower than that of Echidna's limit. For FIR, scheduling overhead is only one factor in calculating the limit, and both NOS and Echidna reach a limit of 8 tasks running at 2ms, or 4 tasks running at 1ms. For Echidna runs, if there is any background load, the data points start to move away from the origin, but the average run is still on time. For NOS, the background has very little affect. With delay, when a system has a light load, both Echidna and NOS are able to service the interrupt immediately (within 1ms is as fast as Echidna can check the interrupt). However, if the system is running with a significant load, Echidna can take up to four times as long to service the interrupt, and NOS has the possibility of dropping the interrupt entirely. The addition of background load has very little affect on these characteristics. With CPU breakdown, several things were seen. Interrupt handling overhead was insignificant because both RTOSs use polling. On the systems where applications are not computational-intensive, it is cheaper to run fewer applications at a faster period than to run more applications at a slower period. Once the system is overloaded it gravitates to an optimal ratio of kernel versus user time.

All of the results obtained in this report could have been obtained using other methods, such as using a logic analyzer to obtain those signals that leave the chip (i/o signals) or using breakpoint instructions to bring off-chip those signals that do not normally leave the chip (register contents). However, those signals that could be obtained with the logic analyzer can only be obtained in this particular instance because an evaluation board of the M-CORE was used in which the components were discrete parts on a printed circuit board, rather than logic blocks on an integrated circuit. The M-CORE processors used in industry are systems on a chip, and therefore those signals would not leave the chip. For those signals that are brought off-chip using the breakpoint instruction, this incurs its own penalty, both slowing the system down, as well as modifying some of the register values.  This report is a tool thesis.  It presents the emulator, describes how it works, and then provides an experiment to validate it.

With the tests and experiments run on this emulator, the report and the research that has lead up to it has shown that this method can be successfully used as an additional method in the evaluation of embedded systems.

# Chapter 7: Future Work

There are several different directions that future work in this area can continue in. Adding more emulator output would be very beneficial. In addition to the cycle count, register contents, access to I/O, and CPU usage that the emulator already outputs, memory write and read frequencies, memory access localities, instruction frequencies, power estimates, and cost estimates could be added. This information would lend further insight to given architecture and would allow speculation on possible changes or modifications to improve on those values. The memory frequencies and locality information might show whether an on chip cache would be beneficial, where as the instruction frequencies would show where improvements could be made to the architecture method for carrying out those instructions.

Running more experiments and more diverse benchmarks on the emulator would also be beneficial. Observing the performance of varying benchmarks and programs would allow the determination of which applications would benefit from this particular architecture, and which applications might benefit from changes to the current architecture. If a class of applications seem to be running slowly, and it is determined that the applications use a large number of multiply and divide instructions, this would lead to consideration of improving the multiply/divide unit.

Porting other Real-Time operating systems to this emulator also might prove beneficial. The SERTS Laboratory Echidna is a operating system that is currently in development, and porting a system such as MicroC/OS-II, Linux, or Windows CE that has wide spread use in industry might provide additional information.

Creating other architectural emulators is another possible route of research. Once several other architectures have been implemented (such as StrongARM, Coldfire, etc.), comparisons could be made between the output for each of the architectures.

After an emulator has been created, and tested with various benchmarks and programs, modifications to that emulator is the next logical step. As mentioned above, embedded systems without caches might benefit from the addition of them. The emulator allows this investigation to proceed at a minimum of cost, where as in the real world, adding an on chip cache to an existing embedded chip would be both costly and time consuming.

Finally, a method of dynamic emulator creation would be a worthy direction of further research. Creating some method, perhaps using Perl scripting, to dynamically create, from an instruction set architecture file, an emulator capable of accurately simulating the architecture, would both be greatly useful for testing multiple architectures accurately as well as cutting the generation time for each architecture down to nearly nothing.

# Appendix A: M-CORE Instruction Set

Table A-1: M-CORE Instructions

Below is a complete listing of all of the M-CORE instructions with a short description of what each instruction does.

| Mnemonic | Description |
|---|---|
| ABS | Absolute Value |
| ADDC | Add with C bit |
| ADDI | Add Immediate |
| ADDU | Add Unsigned |
| AND | Logical AND |
| ANDI | Logical AND Immediate |
| ANDN | AND NOT |
| ASR | Arithmetic Shift Right |
| ASRC | Arithmetic Shift Right, Update C Bit |
| BCLRI | Clear Bit |
| BF | Branch on Condition False |
| BGENI | Bit Generate Immediate |
| BGENR | Bit Generate Register |
| BKPT | Breakpoint |
| BMASKI | Bit Mask Immediate |
| BR | Branch |
| BREV | Bit Reverse |
| BSETI | Bit Set Immediate |
| BSR | Branch to Subroutine |
| BT | Branch on Condition True |
| BTSTI | Bit Test Immediate |
| CLRF | Clear Register on Condition False |
| CLRT | Clear Register on Condition True |
| CMPHS | Compare Higher or Same |

| Mnemonic | Description |
|---|---|
| CMPLT | Compare Less-Than |
| CMPLTI | Compare Less-Than Immediate |
| CMPNE | Compare Not Equal |
| CMPNEI | Compare Not Equal Immediate |
| DECF | Decrement on Condition False |
| DECGT | Decrement Register and Set Condition if Result Greater-Than Zero |
| DECLT | Decrement Register and Set Condition if Result Less-Than Zero |
| DECNE | Decrement Register and Set Condition if Result Not Equal to Zero |
| DECT | Decrement on Condition True |
| DIVS | Divide (Signed) |
| DIVU | Divide (Unsigned) |
| DOZE | Doze |
| FF1 | Find First One |
| INCF | Increment on Condition False |
| INCT | Increment on Condition True |
| IXH | Index Halfword |
| IXW | Index Word |
| JMP | Jump |
| JMPI | Jump Indirect |
| JSR | Jump to Subroutine |
| JSRI | Jump to Subroutine Indirect |
| LDB | Load Byte |
| LDH | Load Halfword |
| LDW | Load Word |
| LDM | Load Multiple Registers |
| LDQ | Load Register Quadrant |
| LOOPT | Decrement with C-Bit Update and Branch if Condition True |
| LRW | Load Relative Word |

| Mnemonic | Description |
|---|---|
| LSL | Logical Shift Left |
| LSLC | Logical Shift Left, Update C Bit |
| LSLI | Logical Shift Left by Immediate |
| LSR | Logical Shift Right |
| LSRC | Logical Shift Right, Update C Bit |
| LSLI | Logical Shift Right by Immediate |
| MFCR | Move from Control Register |
| MOV | Move |
| MOVI | Move Immediate |
| MOVF | Move on Condition False |
| MOVT | Move on Condition True |
| MTCR | Move to Control Register |
| MULT | Multiply |
| MVC | Move C Bit to Register |
| MVCV | Move Inverted C Bit to Register |
| NOT | Logical Complement |
| OR | Logical Inclusive-OR |
| ROTLI | Rotate Left by Immediate |
| RSUB | Reverse Subtract |
| RSUBI | Reverse Subtract Immediate |
| RTE | Return from Exception |
| RFI | Return from Fast Interrupt |
| SEXTB | Sign-Extend Byte |
| SEXTH | Sign-Extend Halfword |
| STB | Store Byte |
| STH | Store Halfword |
| STW | Store Word |
| STM | Store Multiple Registers |

| Mnemonic | Description |
|---|---|
| STQ | Store Register Quadrant |
| STOP | Stop |
| SUBC | Subtract with C Bit |
| SUBI | Subtract Immediate |
| SUBU | Subtract Unsigned |
| SYNC | Synchronize |
| TRAP | Trap |
| TST | Test Operands |
| TSTNBZ | Test for No Byte Equal to Zero |
| WAIT | Wait |
| XOR | Exclusive OR |
| XSR | Extended Shift Right |
| XTRB0 | Extract Byte 0 |
| XTRB1 | Extract Byte 1 |
| XTRB2 | Extract Byte 2 |
| XTRB3 | Extract Byte 3 |
| ZEXTB | Zero-Extend Byte |
| ZEXTH | Zero-Extend Halfword |

# REFERENCES

[1] L. Abeni and G. Buttazzo. "Integrating multimedia applications into hard real-time systems." In Proc. IEEE Real-Time Systems Symposium(RTSS), 1998.

[2] J. H. Anderson, et al. "Efficient object sharing in quatum-based real-time systems." In Proc. IEEE Real-Time Systems Symposium(RTSS), 1998.

[3] T. Anderson. "System-on-chip design with virtual components." Circuit Cellar, No. 109, pp. 12-19, August 1999.

[4] M. J. Bach. The Design of the Unix Operating System. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[5] S. R. Ball. Embedded Microprocessor Systems: Real-World Design. Newnes, Butterworth-Heinemann, Boston MA, 1996.

[6] L. A. Barroso, et al. "Memory system characterization of commercial workloads." In Proc. 25th Annual International Symposium on Computer Architecture (ISCA ’98), Barcelona, Spain, June 1998, pp. 3-14.

[7] A. Bestavros and S. Nagy. "Value-cognizant admission control for RTDB systems." In Proc. IEEE Real-Time Systems Symposium(RTSS), 1996.

[8] M. Brockmeyer, et al. "A flexible, extensible simulation environment for testing real-time specifications." In Proc. IEEE Real-Time Systems Symposium(RTSS), 1997.

[9] D&T Roundtable. "Hardware-Software codesign." IEEE Design and Test of Computers, Vol. 14, No. 1, pp 75-83, 1997.

[10] Echidna. Echidna: A Real-Time Operating System to Support Reconfigurable Software on Microcontrollers and Digital Signal Processors. Software Engineering for Real-Time Systems Laboratory, University of Maryland, http://www.ece.umd.edu/serts/research/echidna/index.shtml, 2000.

[11] R. Enrst. "Codesign of Embedded Systems: Status and Trends." IEEE Design and Test of Computers, Vol. 15, No. 2, pp. 45-54, 1998.

[12] D. D. Gajski and F. Vahid. "Specification and Design of Embedded Hardware-Software Systems." IEEE Design and Test of Computers, Vol. 12, No. 1, pp. 53-67, 1995.

[13] J. Ganssle. "Conspiracy Theory, take 2." The Embedded Muse newsletter, no. 47, March 22, 2000.

[14] J. G. Ganssle. "An OS in a can." Embedded Systems Programming, January 1994.

[15] J. G. Ganssle. "The challenges of real-time programming." Embedded Systems Programming, vol. 11, no. 7, pp. 20-26, July 1997.

[16] L. Garber and D. Sims. "In Pursuit of Hardware-Software Codesign." IEEE Computer, Vol. 31, No. 6, pp. 12-14, 1998.

[17] R. K. Gupta. "A framework for interactive analysis of timing constraints in embedded systems." In Workshop on Hardware-Software Co-Design (CODES), 1996.

[18] R. K. Gupta and G. D. Micheli. "Hardware-Software Co-synthesis for Digital Systems." IEEE Design and Test of Computers, Vol. 10, No. 3, pp. 29-41, 1993.

[19] R. K. Gupta and G. D. Micheli. "Specification and analysis of timing constraints for embedded systems." IEEE Transactions on Computer-Aided Design, vol. 16, no. 3, pp. 240–256, March 1997.

[20] L. Gwennap. "New Processor Paradigm: V-IRAM." Microprocessor Report, Vol. 12, No. 3, pp. 17-19, 1998.

[21] J. R. Haritsa, et al. "Earliest deadline scheduling for real-time database systems." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1991.

[22] M. G. Harmon, et al. "A retargetable technique for predicting execution time." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1992.

[23] C. A. Healy, et al. "Integrating the timing analysis of pipelining and instruction caching." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1995.

[24] J. Hennessy and M. Heinrich. “Hardware/Software codesign of processors: Concepts and examples.” Presented at Advanced Study Institute (ASI). Temezzo Italy, June 1995.

[25] S. A. Herrod. Using Complete Machine Simulation to Understand Computer System Behavior, Ph.D. Dissertation, Department of Computer Science, Stanford University, February 1998.

[26] J. Huang, et al. "On using priority inheritance in real-time databases." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1991.

[27] D. Kalinsky. "A survey of task schedulers." In Embedded Systems Conference 1999, San Jose CA, September 1999.

[28] D. I. Katcher, et al. "Modeling DSP operating systems for multimedia applications." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1994.

[29] C. E. Kozyrakis, et al. "Scalable Processors in the Billion-Transistor Era: IRAM." IEEE Computer, Vol. 30, No. 9, pp. 75-78, 1997.

[30] C.-G. Lee, et al. "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1996.

[31] Y.-T. S. Li, et al. "Efficient microarchitecture modeling and path analysis for real-time software." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1995.

[32] C. Liem, et al. "System-on-a-Chip Cosimulation and Compilation." IEEE Design and Test of Computers, Vol. 14, No. 2, pp. 16-25, 1997.

[33] J. W. S. Liu, et al. "PERTS: A prototyping environment for real-time systems." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1993.

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

[35] T. Lundqvist and P. Stenstrom. "Timing anomalies in dynamically scheduled microprocessors." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1999.

[36] Mcore. M-CORE Evaluation System User’s Manual. Motorola Literature Distribution, Denver CO, 1997

[37] Mcore. M-CORE Reference Manual. Motorola Literature Distribution, Denver CO, 1997.

[38] Mcore. M-CORE MMC2001 Reference Manual. Motorola Literature Distribution, Denver CO, 1998.

[39] B. Nayfeh, et al. "Evaluation of design alternatives for a multiprocessor microprocessor." In Proc. 23rd Annual International Symposium on Computer Architecture (ISCA'96), Philadelphia PA, May 1996, pp. 67–77.

[40] M. Rosenblum, et al. "Complete Computer System Simulation: The SimOS Approach." IEEE Parallel and Distributed Technology, Fall 1995.

[41] M. Rosenblum, et al. "The impact of architectural trends on operating system performance." In Proc. 15th ACM Symposium on Operating Systems Principles (SOSP'95), December 1995.

[42] M. Rosenblum, et al. "Using the SimOS machine simulator to study complex computer systems." ACM Trans. on Modeling and Computer Simulation, vol. 7, no. 1, pp. 78–103, January 1997.

[43] S. Schulz, et al. "Model-Based Codesign." IEEE Computer, Vol. 31, No. 8, pp. 60-67, 1998.

[44] SimOS. SimOS: The Complete Machine Simulator. Stanford University, http://simos.stanford.edu/, 1998.

[45] M. J. S. Smith. Application-Specific Integrated Circuits.Addison-Wesley, Reading, Massachusetts, 1997.

[46] D. Stepner, et al. "Embedded application design using a real-time OS." DAC 99, New Orleans LA, 1999.

[47] D. B. Stewart. "Designing Software Components for Real-Time Applications." Embedded Systems Conference, San Jose, CA, September 1999.

[48] D. B. Stewart, et al. "The Chimera II real-time operating system for advanced sensor-based applications." IEEE Transactions on Systems, Man, and Cybernetics, vol. 22, no. 6, pp. 1282–1295, November/December 1992.

[49] D. B. Stewart, et al. "A Run-Time Environment to Support Reconfigurable Real-Time Software on Embedded Microcontrollers and Digital Signal Processors." Submitted to IEEE Real-Time Application Symposium(RTAS), 1999.

[50] D. B. Stewart, et al. "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.

[51] H. Theiling and C. Ferdinand. "Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1998.

[52] J. Vaglica. "Efficient Interrupt Processing on the M-CORE Architecture." Tech. Report. Motorola, Inc.

[53] M. Xiong, et al. "Scheduling transactions with temporal constraints: Exploiting data semantics." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1996.

[54] H. Zhou, et al. "Performance effects of information sharing in a distributed multiprocessor real-time scheduler." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1992.