Experimental Scheduling Project Report

Performance Engineering of Real-Time and Embedded Systems

Christopher Wood, caw4567@rit.edu
Vineeth Vijayakumaran, vxv7555@rit.edu

1/11/2012

1. Introduction

The purpose of this project was to experiment with a variety of fixed- and dynamic-priority real-time scheduling algorithms, including Rate Monotonic Analysis (RMA), Earliest Deadline First (EDF), and Shortest Completion Time (SCT). This was done in order to determine their effectiveness at scheduling different sets of tasks in a real-time operating system (specifically, QNX) that are schedulable and unschedulable under certain conditions.

Our experimental process consisted of four steps that were completed in a linear fashion. This approach is outlined below.

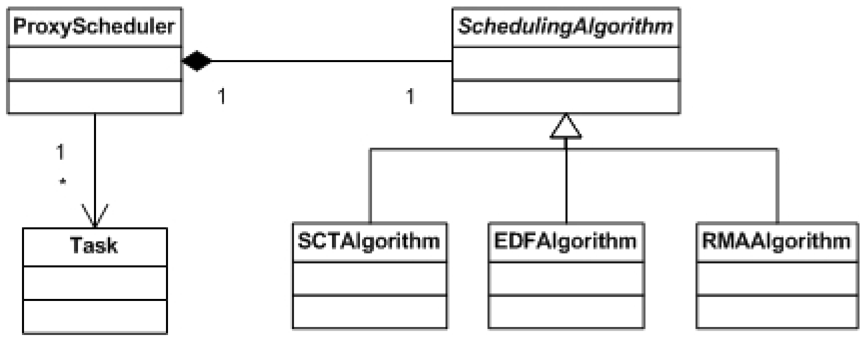
1. Designing and developing a custom test fixture that included support for an arbitrary number of tasks (active threads) and a proxy scheduler that controlled the execution of such tasks.
2. Devising an appropriate set of tasks that exercised the upper and lower bounds for creating a feasible schedule using each scheduling algorithm.
3. Executing each task set using the test fixture and recording the task and schedule data that was collected at runtime.
4. Performing a qualitative and statistical analysis on the resulting data so as to analyze the schedulability of each task set and algorithm, as well as the failures that occurred during the course of each test (i.e. those tests that yielded missed deadlines).

The rest of this document is outlined as follows. The design of our test fixture used to run the experiments is described in section 2, followed by the description of our task sets, experimental results, and analysis in sections 3, 4, and 5, respectively. Section 6 closes with a reflection and concluding remarks.

2. Test Fixture Setup

2.1 Test Fixture Design

In order to make the implementation of our test fixture as effortless as possible we chose to make use of the object-oriented paradigm of application development, which allowed us to use various inheritance and class design patterns when assembling the pieces of our test fixture. The most significant design decision we made was to make use of the Strategy pattern to encapsulate the logic of each scheduling algorithm into a single object that can be dynamically changed by the client of the algorithm, which is the proxy scheduler in this case. The class diagram for this design is shown below.



Test Fixture Class Diagram

The following lists identify the major functional responsibilities and collaborations for each of the entities shown in the above diagram.

ProxyScheduler:

- Invoke the internal scheduling algorithm to determine the priorities of the tasks to be scheduled.
- Manually change the priority of the tasks under control using the QNX threading API.
- Manually control the execution of tasks under control using thread synchronization primitives (e.g. binary semaphores).
- Preempt the currently executing task if a period expiration signal is received.
- Start and terminate a schedule test based on the user-provided runtime parameter.

Task:

- Represent an active thread in the system that is controlled by the proxy scheduler.
- Store an appropriate task compute and period time pair.
- Configure an asynchronous timer to signal events at intervals equivalent to the task period.
- Simulate computation time (e.g. non-blocking execution) when scheduled to execute.
- Signal period expiration events to the proxy scheduler when the timer expires.

SchedulingAlgorithm:

- Provide an interface that all concrete scheduling algorithms must adhere to.

Concrete SchedulingAlgorithm:

- Determine the task with the highest priority using the internally encapsulated scheduling algorithm (RMA, EDF, or SCT).

This class structure allows the proxy scheduler to control the execution of an arbitrary number of tasks with the assistance of its internal scheduling algorithm object. From an object construction perspective, the proxy scheduler will maintain an active collection of task objects and a single scheduling algorithm. The task set will be passed to the internal scheduling algorithm at each scheduling point in order to determine which task should be released next.

2.2 Test Fixture Behavior

Task Computation Cycles

In order to make sure the task computation cycles correspond to the times specified in the task parameter set, we made use of the existing nanospin() QNX primitive in order to waste CPU cycles without sleeping. Since we cannot let a single task spin with a lock on the system for an extended period of time, we decided to make the resolution of our smallest computation time unit (the time quantum) to 0.1ms. However, as is pointed out in section 4.1, this time quantum required some adjustment in order to remove some of the QNX overhead from our tests.

Also, in order to support dynamic and responsive scheduling of tasks for the scheduling algorithms, the tasks break their entire computation cycles into relatively small nanospin() blocks. In the event of a task being preempted during its computation cycle it must be able to store its current compute time so that it can resume execution from that spot when it is rescheduled. With relatively small nanospin() blocks it is possible to interrupt a task's computation cycle at the appropriate time. A rough pseudo-code description for this procedure is shown below.

Task Execution Pseudo-Code

```
While test still running
  currentComputeTime <= 0
  Wait on scheduling semaphore
  For currentComputeTime to claimedComputeTime
    If preemptedFlag != True
      Invoke nanospin(timeQuantum)
      currentComputeTime <= currentComputeTime + timeQuantum
    Else
      Break out of For loop
    End If
  End For
End While
```

Scheduling Points

For the purpose of our task scheduling structure and the requirements for each of the three scheduling algorithms, there are two main scheduling points that we need to be concerned about for the dynamic priority scheduling algorithms, as shown below.

- Upon the start of the scheduling application
- Upon a task period expiration or deadline event

For the fixed priority RMA scheduling algorithm, the first scheduling point is the only one of importance, as it is the time when each task is assigned its appropriate priority based on their period. The proxy scheduler will then rely on the standard, fixed-priority QNX scheduler to complete the rest of the scheduling for the tasks. For the dynamic priority scheduling algorithms, both scheduling points are of importance, as the priority of a task may change at any of these points. It is the responsibility of the proxy scheduler to let the current scheduling algorithm determine the appropriate task priorities to assign to each of the tasks for the QNX scheduler to handle at each of these points.

Generating events to indicate the expiration of a task's period required an asynchronous event to be generated in the foreground of the system. QNX timers fulfilled this requirement (albeit at the cost of some timing jitter). Each task configures an appropriate timer to expire at periodic intervals based on its own period value. The timer expiration event, which is generated asynchronously, then transfers control to a simple function inside the appropriate timer that resets itself for another period and signals to the proxy scheduler that it has expired. Once the proxy scheduler receives this event from the task it will determine the new task priorities based on the current scheduling algorithm (which simply takes a list of tasks and uses their properties to determine the appropriate priorities based on the specific type of algorithm), use the QNX threading primitives to assign the appropriate priority to each task thread, and then release each task in their scheduled order before blocking on its own semaphore again.

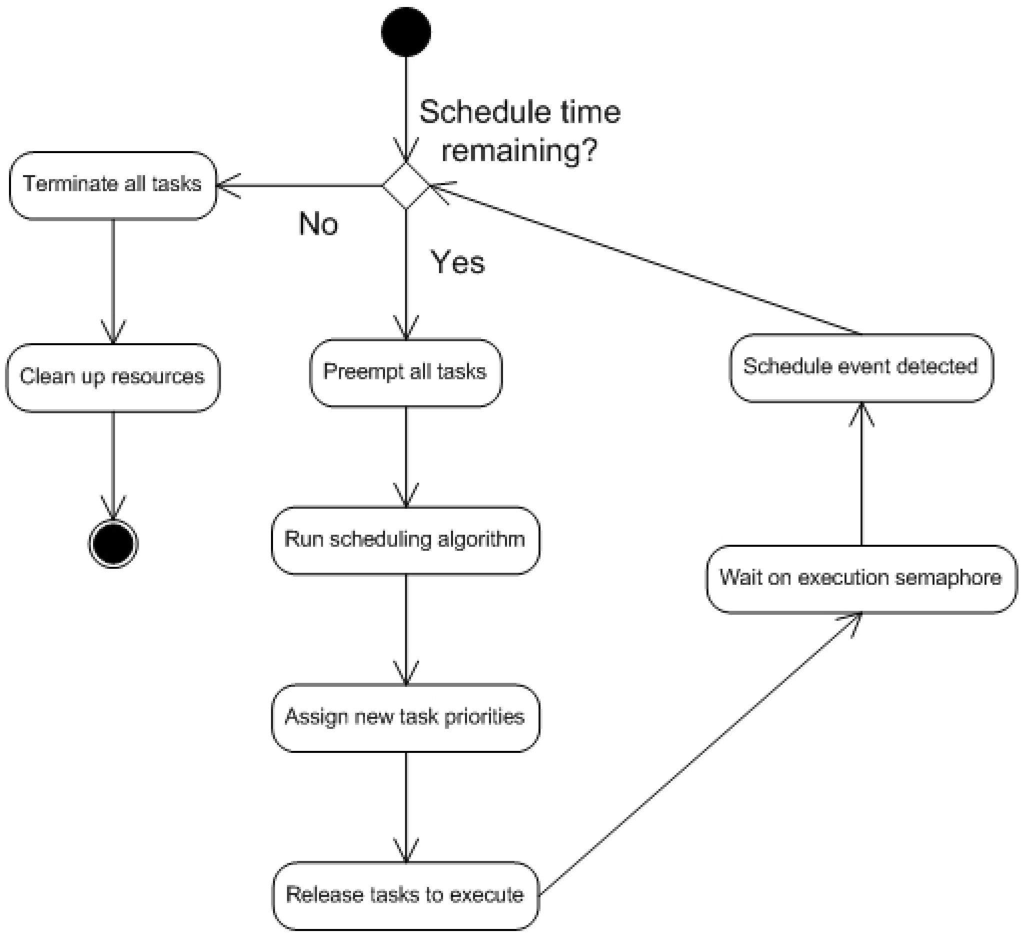
This same procedure is repeated for all three algorithms even though the priority will only change using EDF and SCT. We chose this implementation and design approach because we wanted our schedule test data (including the amount of experimental overhead) to be consistent across all tests to allow for a more uniform statistical analysis.

Task Structure and Synchronization

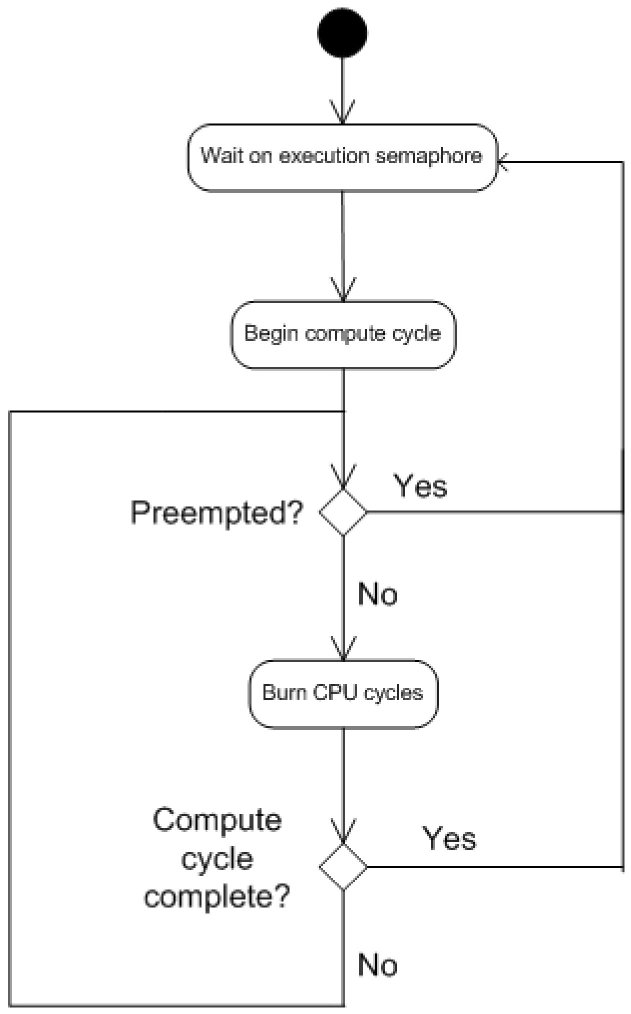
Based on our existing design and identified scheduling points, our tasking structure is composed of a single active proxy scheduler thread and an arbitrary number of task threads under control of the proxy scheduler whose execution is controlled through the use of binary semaphores. More specifically, each task blocks on a unique semaphore until the proxy scheduler posts to such semaphore, at which point the task proceeds to execute its computation cycle to completion when it is scheduled by the QNX scheduler (or until it is preempted by a period expiration event indicated by the proxy scheduler). Upon finishing the computation cycle, each task pends on its own semaphore again until it is released by the proxy scheduler for another computation cycle.

With this design the proxy scheduler relies on the fixed priority QNX scheduler to perform task scheduling after task priorities have been set at each scheduling point. This allowed us to remove all potential overhead that would be associated with manual control of the tasks by the proxy scheduler.

The entire tasking structure and synchronization is outlined below in the activity diagrams for both the proxy scheduler and each task. Note that the schedule events are timer-driven function calls called from the context of a separate, high priority thread. This is done so it will always preempt any task that is running at that time (whether it's the proxy scheduler or another task).



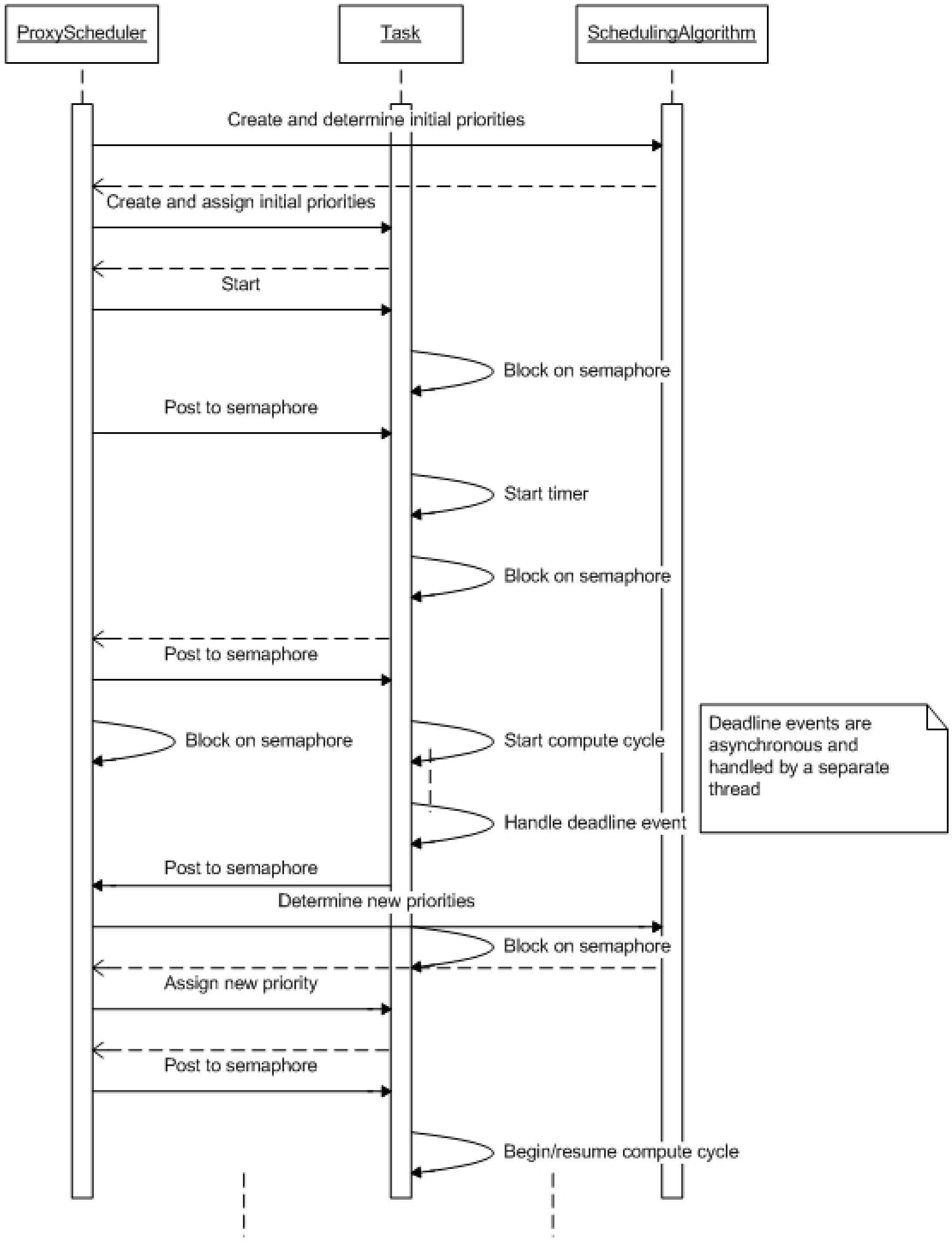
Proxy Scheduler Activity Diagram



Task Activity Diagram

It is important to emphasize that in the task activity diagram above the synchronization logic involved with the computation cycle is performed in the background of the system, whereas the timer-related behavior is (software) interrupt-driven and thus operates in the foreground.

The sequential behavior between the tasks and proxy scheduler for the startup of our test fixture and one complete schedule event is shown in the sequence diagram below.



2.2 Task Instrumentation and Data Collection

In order to collect the data used in the analysis portion of this experiment the proxy scheduler and tasks are instrumented using QNX user events (which are inserted into the kernel event stream using the TraceEvent() function) and simple data collection logic. The schedule of tasks that are set to execute is dynamically generated and displayed using user events by the proxy scheduler. The remaining metrics are compounded during the entire execution cycle of the proxy scheduler and then displayed once the schedule test is complete. The following C structure outlines the data that is compounded at each scheduling point by the proxy scheduler (which, from an implementation perspective, is actually delegated to each task object).

```
typedef struct
{
        uint64_t deadlineEvents;
        uint64_t deadlinesMissed;
        uint64_t deadlinesMet;
        uint64_t totalComputationTimeMissed;
        uint64_t totalComputationTime;
        uint64_t totalComputationCycles;
} TaskData;
```

Putting all of this application instrumentation data together, we collected the following set of information (shown in the order in which they appear in the output CSV strings) for each schedule test.

Task Scheduling Data (during test)

- Task identifier being scheduled
- Task identifier that missed a deadline

Task Data (post test)

- Task identifier
- Total deadline events
- Total deadlines missed
- Total computation time remaining after a missed deadline event
- Total task computation time
- Total computation cycles
- Computation time jitter
- Start and end computation time jitter

Overall Schedule Test Data

- Expected and real scheduling test time
- Total scheduling points

2.3 Test Fixture Experimental Procedure

Our test fixture was designed to read in task characteristics from standard input so as to provide an script-friendly interface for test automation. Upon the start of the program the user is requested to enter the following information:

1. Desired scheduling algorithm (0 - RMA, 1 - EDF, 2 - SCT)
2. Schedule test run time (seconds)
3. Number of tasks
4. Task compute and period time pairs

Using this information, the program then creates the proxy scheduler and task set with this specific set of characteristics. Once this is complete, the proxy scheduler begins the schedule test and runs until the specified test time expires (which is determined by a background timer event). Once the test is complete the kernel event trace file that was captured while profiling is saved for offline data processing, which includes stripping out the information and data specified in section 2.2.

3. Task Sets

In order to generate an appropriate set of data to analyze we first devised a set of tasks that would exercise the limits of each algorithm sufficiently. These task sets are listed below.

Table 1: Task set characteristics

Task Set ID	**Number of Tasks**	**(Compute, Period) Time Pairs**	**Resource Utilization**	**Reasoning**
1	4	(10,500), (10,500), (10,500), (10,500)	0.08	Very low resource utilization among all tasks, used for verification of scheduling algorithms
2	2	(30,40), (20,80)	1.0	Full resource utilization, schedulable by EDF but not SCT
3	2	(20,40), (20,80)	0.75	Schedulable by EDF and SCT
4	3	(45,135), (50,150), (80,360)	0.89	Task set to exercise schedulability theoreom 3 for the RMA algorithm
5	3	(10,40), (2,50), (70,200)	0.64	High variance in task compute cycles
6	3	(10,950), (10,960), (960,990)	0.99	Higher variance in task compute cycles and near impossible task set
7	3	(15,30), (25,30), (25,30)	2.16	Unschedulable task set
8	3	(45,200), (90,400),	0.675	67.5% resource utilization

		(135,600),		
9	3	(24,100), (48,200), (96,400)	0.72	72% resource utilization
10	3	(50,200), (100,400), (25,100)	0.75	75% resource utilization
11	3	(110,400), (165,600), (55,200)	0.825	82.5% resource utilization
12	3	(120,400), (180,600), (60,200)	0.90	90% resource utilization
13	3	(68,200), (68,200), (68,200)	1.02	102% resource utilization (unschedulable task set)
14	3	(58,200), (116,400), (116,400)	0.87	87% resource utilization
15	8	(10,100), (10, 100), (20,200), (20,200), (30, 300), (30, 300), (40, 400), (40, 400)	0.80	80% resource utilization with large task set size

These task sets were devised using the following principles.

1. Coverage of resource utilization ratios
2. Variance in compute cycle times
3. Coverage of all RMA schedulability theorem test cases

The first two properties are discussed in the reasoning column in the above table, whereas the results of the RMA schedulability tests are shown in the table below.

Table 2: RMA schedulability test results for each task set

Task Set ID	**Test One**	**Test Two**	**Test Three**
1	Pass	Pass	Pass
2	Pass	Fail	Pass
3	Pass	Pass	Pass
4	Pass	Fail	Pass
5	Pass	Pass	Pass
6	Pass	Fail	Fail

7	Fail	Fail	Fail
8	Pass	Pass	Pass
9	Pass	Pass	Pass
10	Pass	Pass	Pass
11	Pass	Fail	Pass
12	Pass	Fail	Pass
13	Fail	Fail	Fail
14	Pass	Fail	Pass
15	Pass	Fail	Pass

4. Results

4.1 Experimental Overhead Anaylsis

In order to determine if a schedule test either passed or failed it is important to determine the amount of overhead that was contributed to each test by both the test fixture and QNX. This overhead was measured in a variety of critical sections during the test and includes:

1. Start and stop periods for task compute cycles
2. Compute cycle (e.g. nanospin()) jitter (expected time versus the actual clock time)
3. Proxy Schedule execution time
4. Application instrumentation overhead

Given that the schedulability tests assume there is no overhead in context switching (or any part of the task execution) it is important to understand the impact this has on each test. The initial statistical measures for all of these values that have been compiled across all tests for every algorithm are shown in the table below.

Table 3: Initial experimental overhead results

Overhead Metric	**Value**
Average start and stop jitter for task compute cycles	0.000821ms
Average compute cycle jitter (among all tests)	33.51%
Proxy Schedule execution time	0.000034 ms
Application instrumentation overhead	No reliable measurement method found.

From this we can see that the code that is executed during a schedule event handler and around each task's raw compute cycle (e.g. the nanospin() loop) has a negligible effect on the schedule results for each test. However, the compute cycle jitter seems as though it will have a very significant effect on the schedulability of each task set. Since this jitter corresponds to the variance between each task's expected compute time and the actual compute time, it is clear that each task will consume more resources than originally claimed.

In an attempt to decrease this amount of jitter we investigated the use of nanospin() and its relation to the calling thread's priority. Based on the QNX documentation, nanospin() places the calling thread in a busy loop without blocking it (i.e. it is still left on the running thread queue for scheduling). This led us to believe that the calling thread's priority has an impact on the accuracy of each call to nanospin(). Therefore, we experimented with different task priorities and measured the timing jitter using raw CPU cycle counts for each call to nanospin(). The results of this experiment are shown in the table below.

Table 4: Results from thread priority experiment

Task Priority	**Average Compute Cycle Jitter (%)**
10 (low)	0.3325 (33.25%)
50 (medium)	0.3468 (34.68%)
100 (high)	0.3636 (36.34%)

As one can see, the task priority did not seem to have a significant impact on the amount of jitter from nanospin(). Therefore, in order to generate more reliable timing measurements we introduced a modified time parameter that is approximately 20% less than 0.1ms (that is, 0.08ms), the original proposed nanospin() time quantum. Using this new value we were able to significantly reduce the computation cycle jitter for all tasks, as shown in the table below.

Table 5: Final experimental overhead results

Overhead Metric	**Value**
Average start and stop jitter for task compute cycles	0.000723 ms
Average compute cycle jitter (among all tests)	9.79%
Proxy Schedule execution time	0.000131 ms
Application instrumentation overhead	No reliable measurement method found.

While such a modification would not be an appropriate solution in an industry setting, for the purposes of this experiment it allowed us to perform a more accurate analysis of each task set. Given that the computation jitter was consistently above 30% for each task in every experiment, we felt that this adjustment was appropriate. Furthermore, based on our readings and research, this seemed to be the only plausible solution to the computation cycle jitter problem we were experiencing. Therefore, we felt it was reasonable to use this new time quantum as the basis for each task set experiment.

As a further analysis of the experimental overhead we examined the amount of CPU usage by the operating system and by user processes running in the operating system at the time of each test. This data was collected by profiling the test fixture during each schedule test and consisted of the alleged task set utilization percentage and the actual CPU usage by the test fixture. The result of this data analysis is shown below.

Table 6: System overhead with RMA algorithm

Task Set ID	Algorithm	Test Fixture CPU Usage (%)	Task Set Utilization Percentage (%)	CPU Usage Difference (%)
1	RMA	8.8	8	0.8
2	RMA	78	100	-22.0
3	RMA	66.7	75	-8.3
4	RMA	70	89	-19.0
5	RMA	64.7	64	0.7
6	RMA	75.9	99	-23.1
7	RMA	83.7	216	-132.3
8	RMA	68.5	67.5	1.0
9	RMA	58.8	72	-13.2
10	RMA	60.7	75	-14.3
11	RMA	64	82.5	-18.5
12	RMA	82.7	90	-7.3
13	RMA	68.6	102	-33.4
14	RMA	86.7	87	-0.3
15	RMA	96.2	80	16.2

Table 7: System overhead with EDF algorithm

Task Set ID	Algorithm	Test Fixture CPU Usage (%)	Task Set Utilization Percentage (%)	CPU Usage Difference (%)
1	EDF	11	8	3.0
2	EDF	61	100	-39.0
3	EDF	57.4	75	-17.6
4	EDF	72.4	89	-16.6
5	EDF	58.6	64	-5.4
6	EDF	77.5	99	-21.5
7	EDF	97.4	216	-118.6
8	EDF	72	67.5	4.5
9	EDF	76.2	72	4.2
10	EDF	78.4	75	3.4
11	EDF	82.2	82.5	-0.3
12	EDF	79.4	90	-10.6
13	EDF	84.3	102	-17.7
14	EDF	89.8	87	2.8
15	EDF	96.5	80	16.5

Table 8: System overhead with SCT algorithm

Task Set ID	Algorithm	Test Fixture CPU Usage (%)	Task Set Utilization Percentage (%)	CPU Usage Difference (%)
1	SCT	9.7	8	1.7
2	SCT	81.1	100	-10.9
3	SCT	73.3	75	-1.7

4	SCT	94.2	89	5.2
5	SCT	72.2	64	8.2
6	SCT	76.3	99	-22.7
7	SCT	96	216	-120
8	SCT	65.9	67.5	-1.6
9	SCT	78.3	72	6.3
10	SCT	81	75	6.0
11	SCT	88	82.5	5.5
12	SCT	86.1	90	-3.9
13	SCT	95.5	102	-6.5
14	SCT	90	87	3.0
15	SCT	94.8	80	14.8

Theoretically, with no other background tasks or processes running on the system, we should expect these two percentages to be roughly similar up to a certain threshold. From this data it appears as though the CPU usage seems to very roughly match the theoretical results up to approximately 80% task utilization, at which point the test fixture CPU usage reaches a seemingly asymptotic limit when the task utilization increases any further. This is an indication that there are other background tasks and processes in the system that consume the other 20% of the CPU cycles (potentially less depending on the current state of the system) for separate jobs. Therefore, this overhead is accounted for in our task analysis.

It is also important to note that the experimental overhead is due to the latency from the proxy scheduler and the QNX operating system components of the test fixture. Our test fixture is balanced with the appropriate instrumentation mechanisms to capture the overhead from the proxy scheduler, but the overhead from the QNX scheduler and operating system primitives is more difficult to quantify without causing significant performance issues during each test. Therefore, for the sake of our analysis, we assume there is an arbitrary amount of overhead from all QNX-related services that we cannot measure that impacts the timing results of each test.

One element of experimental overhead that we did not fully measure is the amount of time it took the QNX scheduler to perform context switches between our proxy scheduler and the tasks that were scheduled to run. The reason we left this out of our analysis and data collection is that, from our CPU usage results, it appeared that there was too much other background activity going on that would impact our values for this measurement. Furthermore, the additional overhead and synchronization that would be needed between the separate proxy scheduler and task threads in order to measure this context switch time seemed as though it would burden our test fixture and produce negative results.

4.2 Raw Schedule Data Analysis

Each task set was run against all three scheduling algorithms outlined in section 1. The quantitative results from these tests, as well as the qualitative results that followed from an analysis of this data, are shown below for each task set.

The schedule data for each task set is compiled over all tasks in the set. This was done because the tasks are not independent in terms of schedulability, so looking at the entire task set as a whole yields the most appropriate schedule data for analysis. Also, percentages are displayed in their decimal format (i.e. 50% is recorded as 0.5).

Task Set 1 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	10	0.37345	0.001485	Pass
EDF	0	10	0.35881	0.001488	Pass
SCT	0	10	0.343089	0.001562	Pass

Task Set 1 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	10	0.079634	0.001149	Pass
EDF	0	10	0.095548	0.0011963	Pass
SCT	0	10	0.082878	0.0011963	Pass

Remarks: Since the resource utilization of the task set is so low, it is clear that each test passes without any missed deadlines. It is important to note that, despite this low utilization ratio, the computation time jitter is still approximately 33%, meaning that the nanospin() jitter is independent of the task parameters.

Task Set 2 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.9932	25.475	0.4809195	0.0009515	Fail
EDF	1.0	25.33	0.4069545	0.000851	Fail
SCT	0.9932	25.1	0.4175575	0.0009815	Fail

Task Set 2 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.3287	25.18	0.2268915	0.0016645	Fail
EDF	0.3287	25.458	0.154524	0.0015085	Fail
SCT	0.3310	25	0.215437	0.027846	Fail

Remarks: The utilization ratio for this task set is 1.0, meaning that the only way to avoid missed deadlines is to remove all experimental overhead, which is not possible. Furthermore, given the high percentage of missed deadlines (which spans across all tasks in the task set), it is clear that the majority of the tasks missed their deadlines, meaning that in practical settings, this task set would not be schedulable.

Also, even though EDF is an optimal scheduling algorithm, it is shown to have a higher percentage of missed deadlines for this task set. This is due to the experimental overhead associated with each test, and does not reflect the effectiveness of the algorithm since the utilization ratio is significantly larger than this overhead.

Task Set 3 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.3310	20.06	0.386833	0.001391	Fail
EDF	0.3310	20.075	0.4007085	0.001007	Fail
SCT	0.3310	20.28	0.4150105	0.001653	Fail

Task Set 3 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	20	0.0939695	0.000948	Pass
EDF	0	20	0.084678	0.000879	Pass
SCT	0	20	0.1081665	0.0007565	Pass

Remarks: Since each algorithm passed the schedulability test with the adjusted computation time quantum and failed the test with the original time quantum, it is clear that the task computation time jitter from our original time quantum had a significant impact on the schedulability of this task set.

Task Set 4 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.15384	66.1923	0.4738	0.000642	Fail
EDF	0.15384	59.2606	0.3962	0.000678	Fail
SCT	0.15384	60.927	0.3883	0.00056	Fail

Task Set 4 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.0625	63.35384	0.08254	0.00048	Fail
EDF	0.15625	60.26153	0.1223	0.000606	Fail
SCT	0.06153	60.8897	0.106263	0.034607	Fail

Remarks: Both experiment types failed to meet all task deadlines, although the experiment with the old time quantum suffered more failures than the experiment with the new time quantum. Given that the resource utilization is 1.0 and there is experimental overhead in both cases, this is an acceptable result.

Task Set 5 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	27.3833	0.3946	0.001640	Pass
EDF	0.00507	27.35	0.315966	0.001722	Fail
SCT	0	27.5166	0.37146	0.00291	Pass

Task Set 5 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	27.7	0.108443	0.002093	Pass
EDF	0	27.6	0.1102	0.0027	Pass
SCT	0	27.4666	0.13304	0.00187	Pass

Remarks: One interesting aspect of these results is that, for the old time quantum experiment, the RMA and SCT tests passed and the EDF test failed. Since EDF is considered to be an optimal scheduling algorithm, this seems to contradict the expected theoretical result. However, it is most likely just a side effect of the large amount of compute cycle jitter.

Task Set 6 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.3333	457.5	0.3472	0.001335	Fail
EDF	0.3333	459.333	0.35455	0.001342	Fail
SCT	0.3333	449.1666	0.35486	0.001276	Fail

Task Set 6 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition	Schedulability Test Result

				Jitter (ms)	
RMA	0.3333	603.333	0.0708	0.002069	Fail
EDF	0.3333	125.857	0.6767	0.001126	Fail
SCT	0.3333	326.666	0.08302	0.001815	Fail

Remarks: Given the high utilization ratio of this task set (0.99) it is clear that each task set should fail within our test fixture, so this result is acceptable.

Task Set 7 - Old Time Quantum

Algorithm	**Percentage of Deadlines Missed (%)**	**Average Compute Time (ms)**	**Compute Time Jitter (%)**	**Compute Cycle Transition Jitter (ms)**	**Schedula bility Test Result**
RMA	1.0	25.277	0.424086	0.0012	Fail
EDF	0.5945	20.1788	1.884	0.03521	Fail
SCT	0.547	20.2649	0.3797	0.0220	Fail

Task Set 7 - New Time Quantum

Algorithm	**Percentage of Deadlines Missed (%)**	**Average Compute Time (ms)**	**Compute Time Jitter (%)**	**Compute Cycle Transition Jitter (ms)**	**Schedula bility Test Result**
RMA	0.49618	26.8787	0.265234	0.00473	Fail
EDF	0.66867	20.1800	1.056294	0.00768	Fail
SCT	0.58566	20.2334	1.239096	0.007946	Fail

Remarks: This task set has an extremely overburdened resource utilization percentage (216%), so every task should and has indeed failed the test. It is neither theoretically or practically schedulable.

Task Set 8 - Old Time Quantum

Algorithm	**Percentage of Deadlines Missed (%)**	**Average Compute Time (ms)**	**Compute Time Jitter (%)**	**Compute Cycle Transition Jitter (ms)**	**Schedula bility Test Result**

RMA	0	96.9444	0.29519	0.00024	Pass
EDF	0.02941	90	0.37759	0.000505	Fail
SCT	0.10447	90	0.35823	0.000275	Fail

Task Set 8 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	90	0.07339	0.000369	Pass
EDF	0	90	0.10413	0.000261	Pass
SCT	0	90	0.104132	0.000261	Pass

Remarks: This task set has a relatively low resource utilization percentage so it is expected that each test should pass, as is the case with the new time quantum experiments. However, due to the high compute cycle jitter in the old time quantum experiments, the EDF and SCT schedule tests failed.

Task Set 9 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.0597	56.2333	0.36644	0.00039	Fail
EDF	0	58.777	0.35777	0.000349	Pass
SCT	0.1044	59.4074	0.37294	0.024024	Fail

Task Set 9 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	56.666	0.0948	0.000531	Pass
EDF	0	56	0.0760	0.000413	Pass
SCT	0	56	0.0760	0.000413	Pass

Remarks: This task set has a relatively low resource utilization percentage so it is expected that each test should pass, as is the case with the new time quantum experiments. However, due to the high compute cycle jitter in the old time quantum experiments, the RMA and SCT schedule tests failed. This is an interesting result because in the previous task set, which had a lower utilization percentage, RMA passed and EDF failed. The seems to indicate that the failure of each likely stems from a timing issue (e.g. compute cycle jitter) that yields relatively unpredictable results.

Task Set 10 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.1343	59.814	0.343	0.03549	Fail
EDF	0.5820	60.166	0.376	0.00129	Fail
SCT	0.1343	57.888	0.360	0.000689	Fail

Task Set 10 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0	58.666	0.07142	0.000547	Pass
EDF	0	59.833	0.085808	0.00043	Pass
SCT	0	59.833	0.085808	0.000431	Pass

Remarks: This task set has an average resource utilization percentage so it is expected that each test with the new time quantum experiment should pass and each test with old time quantum experiment should fail, as observed.

Task Set 11 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.17647	112.5	0.3848	0.00028	Fail

EDF	0.55882	110.777	0.3477	0.00028	Fail
SCT	0.7058	118.4666	0.35278	0.000286	Fail

Remarks: This task set has an average resource utilization percentage so it is expected that each test with the new time quantum experiment should pass and each test with old time quantum experiment should fail, as observed.

Task Set 11 - New Time Quantum

Algorithm	**Percentage of Deadlines Missed (%)**	**Average Compute Time (ms)**	**Compute Time Jitter (%)**	**Compute Cycle Transition Jitter (ms)**	**Schedula bility Test Result**
RMA	0	124.111	0.0791	0.00029	Pass
EDF	0	110	0.0952	0.00026	Pass
SCT	0	110	0.0952	0.000260	Pass

Task Set 12 - Old Time Quantum

Algorithm	**Percentage of Deadlines Missed (%)**	**Average Compute Time (ms)**	**Compute Time Jitter (%)**	**Compute Cycle Transition Jitter (ms)**	**Schedula bility Test Result**
RMA	0.1764	120.333	0.35741	0.00027	Fail
EDF	0.7352	120.9166	0.381	0.000321	Fail
SCT	0.1764	143.5	0.39446	0.000449	Fail

Task Set 12 - New Time Quantum

Algorithm	**Percentage of Deadlines Missed (%)**	**Average Compute Time (ms)**	**Compute Time Jitter (%)**	**Compute Cycle Transition Jitter (ms)**	**Schedula bility Test Result**
RMA	0.125	125.111	0.1081	0.000303	Fail
EDF	0.558	121.388	0.0904	0.00031	Fail
SCT	0.1470	92.7222	0.1064	0.00031	Fail

Remarks: This task set has a high utilization percentage that should still be schedulable according to the theoretically expected results for each algorithm. However, as observed, each test fails the test due to the real-time constraints on our test fixture.

Task Set 13 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.3333	74	0.499	0.00089	Fail
EDF	0.3333	76.5	0.47226	0.000865	Fail
SCT	0.3333	70	0.3915	0.00078	Fail

Task Set 13 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.3333	68.444	0.1280	0.001266	Fail
EDF	0.3333	68.888	0.1208	0.000572	Fail
SCT	0.3333	69.055	0.1159	0.000572	Fail

Remarks: This task set is theoretically unschedulable so all of the task sets should fail the schedulability test, as observed.

Task Set 14 - Old Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedulability Test Result
RMA	0.2432	99.8666	0.3553	0.1169	Fail
EDF	0	105.333	0.3962	0.0002	Pass
SCT	0.2432	104.916	0.39974	0.00028	Fail

Task Set 14 - New Time Quantum

Algorithm	Percentage	Average	Compute	Compute	Schedula

	of Deadlines Missed (%)	Compute Time (ms)	Time Jitter (%)	Cycle Transition Jitter (ms)	bility Test Result
RMA	0	100.740	0.0788	0.00022	Pass
EDF	0.0540	99.555	0.1042	0.00026	Fail
SCT	0.0540	100.296	0.1045	0.00026	Fail

Remarks: This task set has a relatively high utilization percentage that should be schedulable theoretically, but in fact fails the schedulability test for the EDF and SCT dynamic algorithms. Interestingly enough, the RMA algorithm passed the test, which seems to correspond with the results from task set 5.

Task Set 15 - New Time Quantum

Algorithm	Percentage of Deadlines Missed (%)	Average Compute Time (ms)	Compute Time Jitter (%)	Compute Cycle Transition Jitter (ms)	Schedula bility Test Result
RMA	0.2857	21.671	0.0813	0.0060	Fail
EDF	0.285	21.735	0.0829	0.00075	Fail
SCT	0.285	21.694	0.0807	0.00072	Fail

Remarks: This task set has a large number of tasks with a relatively high utilization percentage. It should be theoretically schedulable using all three algorithms, but due to the overhead of dealing with many threads (e.g. context switches) and already existing experimental overhead each schedule test failed.

5. Analysis of Schedulability

Given the quality of our results using the newly adjusted time quantum they were used as the basis for a more in-depth analysis for each scheduling algorithm.

5.1 RMA Analysis

There are three fundamental schedulability theorems for the RMA algorithm, as shown below.

1. $U = \sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$

2. $U \leq n(2^{\frac{1}{n}} - 1)$

3. $\forall\, i\; min(\sum_{j=1}^{i} \frac{Cj}{lPk}[\frac{lPk}{Pj}]) \leq 1$

Using these tests we compared the results of each schedulability test against the experimental data presented in section 4.2. The results of this comparison is shown below.

Table 9: Schedulability results compared against the first RMA schedulability test

Task Set ID	**Schedulability Test**	**Theoretical Test Success**	**Experimental Test Success**
1	1	Pass	Pass
2	1	Pass	Fail
3	1	Pass	Pass
4	1	Pass	Fail
5	1	Pass	Pass
6	1	Pass	Fail
7	1	Fail	Fail
8	1	Pass	Pass
9	1	Pass	Pass
10	1	Pass	Pass
11	1	Pass	Pass
12	1	Pass	Fail
13	1	Fail	Fail
14	1	Pass	Pass
15	1	Pass	Fail

Remarks: Our results indicate that the RMA algorithm matches the theoretical results for this schedulability test up to a resource utilization of approximately 89% (task set 4). Even though task set 15 has a utilization of 80%, the amount of experimental overhead seems to be significant enough to account for this failed task set.

Table 10: Schedulability results compared against the second RMA schedulability test

Task Set ID	Schedulability Test	Theoretical Test Success	Experimental Test Success
1	2	Pass	Pass
2	2	Fail	Fail
3	2	Pass	Pass
4	2	Fail	Fail
5	2	Pass	Pass
6	2	Fail	Fail
7	2	Fail	Fail
8	2	Pass	Pass
9	2	Pass	Pass
10	2	Pass	Pass
11	2	Fail	Pass
12	2	Fail	Fail
13	2	Fail	Fail
14	2	Fail	Pass
15	2	Fail	Fail

Remarks: Our results indicate that the RMA algorithm matches the theoretical results for this schedulability test entirely because there are no failed test cases where the theoretical test passed. Even though task set 15 has a utilization of 80%, the amount of experimental overhead seems to be significant enough to account for this failed task set.

Table 11: Schedulability results compared against the third RMA schedulability test

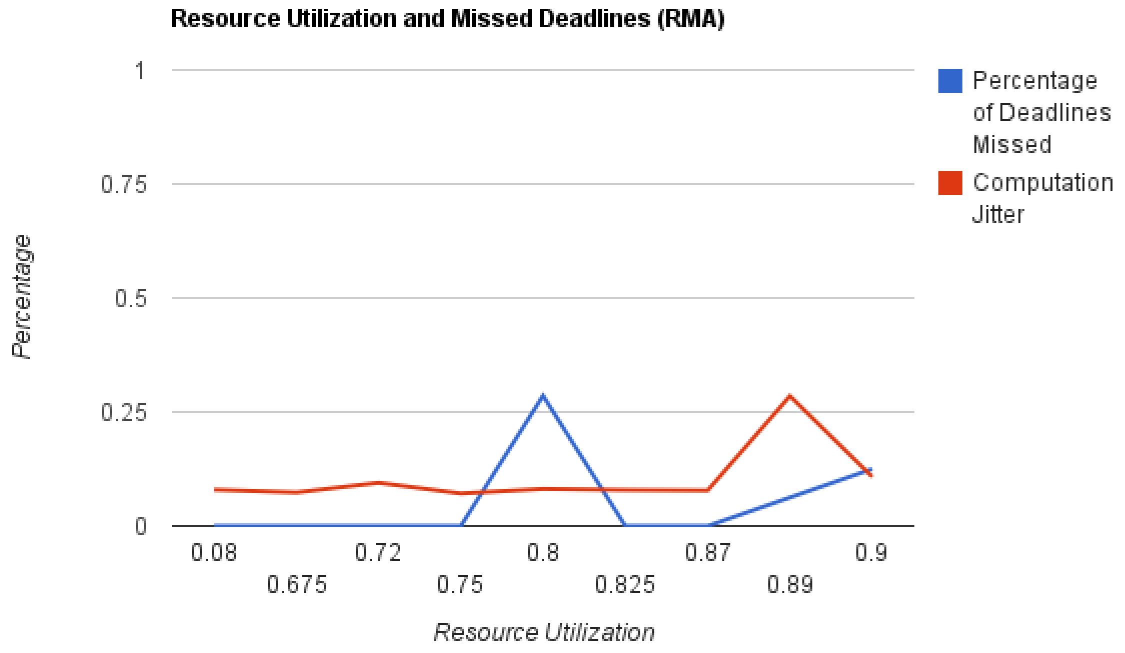
Task Set ID	Schedulability Test	Theoretical Test Result	Experimental Test Result
1	3	Pass	Pass
2	3	Pass	Fail
3	3	Pass	Pass
4	3	Pass	Fail
5	3	Pass	Pass

6	3	Fail	Fail
7	3	Fail	Fail
8	3	Pass	Pass
9	3	Pass	Pass
10	3	Pass	Pass
11	3	Pass	Pass
12	3	Pass	Fail
13	3	Fail	Fail
14	3	Pass	Pass
15	3	Pass	Fail

Remarks: Our results indicate that the RMA algorithm matches the theoretical results for this schedulability test up to a resource utilization of approximately 89% (task set 4). Even though task set 15 has a utilization of 80%, the amount of experimental overhead seems to be significant enough to account for this failed task set.

To visualize the schedulability results from the RMA algorithm graphically, we have plotted the percentage of deadlines missed against the resource utilization. It is clear that the number of missed deadlines starts to increase at approximately 85-89%, which correlates with the practical and theoretical results we have shown above. This upper bound is most likely an immediate side effect of the experimental overhead (which was approximately 10%), so for utilization percentages up this mark the practical results match the expected theoretical results almost exactly.

Also, there is an increase in missed deadlines at 80% utilization for task set 15, but these missed deadlines are likely due to the experimental overhead due to task context switching, multiple thread management, and task period timer overhead.



5.2 EDF Analysis

Theoretically, Equation 1 from the RMA schedulability tests is necessary and sufficient for the resource utilization of a given task set in order for it to be schedulable using the EDF algorithm. Using this fact and the results from section 4.2, we can determine the upper bound on the resource utilization for a task set such that it passes the schedule test. We do so by comparing the utilization ratio for each task set with its result from the schedulability test, as shown below.

Table 12: Schedulability results compared against the EDF schedulability test

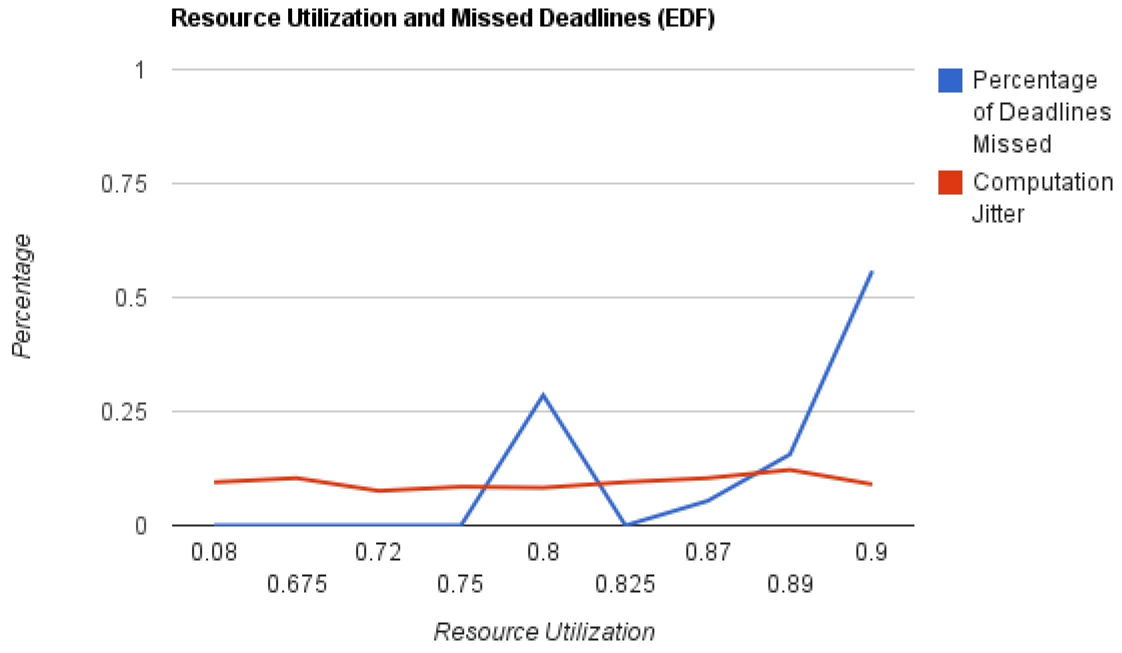
Task Set ID	Resource Utilization	Schedulability Test Result
1	0.08	Pass
2	1.0	Fail
3	0.75	Pass
4	0.89	Fail
5	0.64	Pass
6	0.99	Fail
7	2.16	Fail
8	0.675	Pass

9	0.72	Pass
10	0.75	Pass
11	0.825	Pass
12	0.90	Fail
13	1.02	Fail
14	0.87	Fail
15	0.80	Fail

From these results we can conclude that the EDF algorithm matches its schedulability test up to a resource utilization of 87% (task set 14). Even though task set 15 has a utilization of 80%, the amount of experimental overhead seems to be significant enough to account for this failed task set.

To visualize the schedulability results from the EDF algorithm graphically, we have plotted the percentage of deadlines missed against the resource utilization. It is clear that the number of missed deadlines starts to increase at approximately 82.5-87%, which correlates with the practical and theoretical results we have shown above. This upper bound is most likely an immediate side effect of the experimental overhead (which was approximately 10%), so for utilization percentages up this mark the practical results match the expected theoretical results almost exactly.

Also, as with the RMA test, there is an increase in missed deadlines at 80% utilization for task set 15, but these missed deadlines are likely due to the experimental overhead (due to task context switching, thread management, and task period timer overhead).



5.3 SCT Analysis

Although there is no concrete theoretical schedulability test for the SCT algorithm (at least one we could find during our research), we can analyze our results from section 4.2 to determine the upper bound on the resource utilization for a task set such that it passes the schedule test. In this way, we are applying the same schedulability test as the EDF algorithm to determine the theoretically expected results. This data analysis was done by comparing the utilization ratio for each task set with its result from the schedulability test, as shown below.

Table 13: Schedulability results compared against the proposed SCT schedulability test

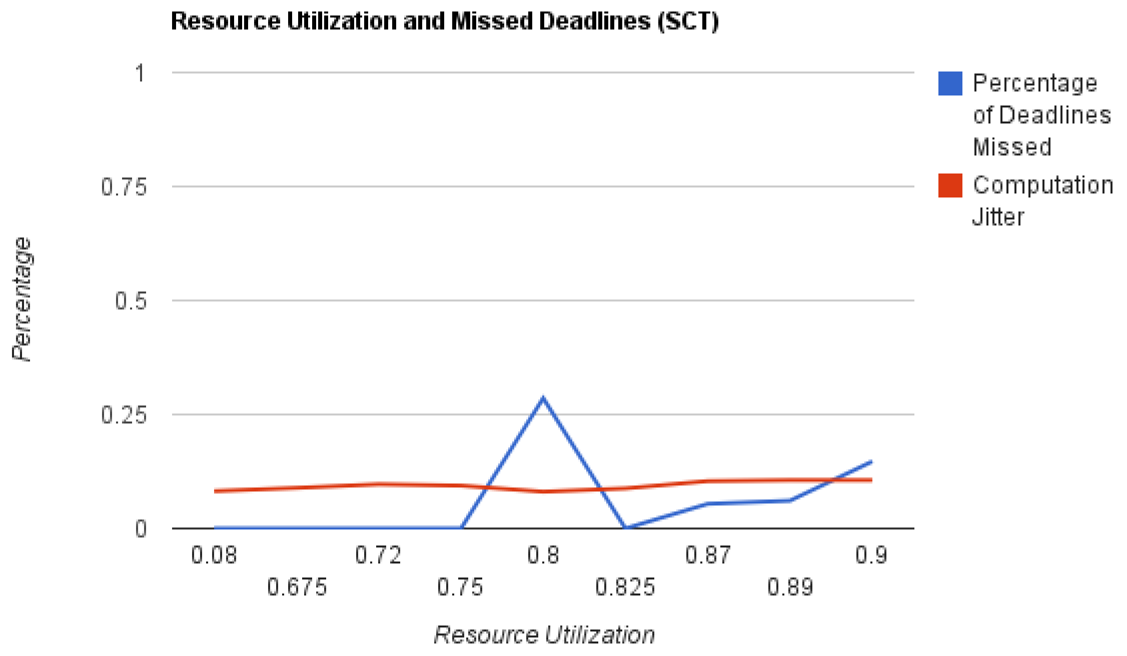
Task Set ID	**Resource Utilization**	**Schedulability Test Result**
1	0.08	Pass
2	1.0	Fail
3	0.75	Pass
4	0.89	Fail
5	0.64	Pass
6	0.99	Fail
7	2.16	Fail
8	0.675	Pass

9	0.72	Pass
10	0.75	Pass
11	0.825	Pass
12	0.90	Fail
13	1.02	Fail
14	0.87	Fail
15	0.80	Fail

From these results we can conclude that the SCT algorithm matches its schedulability test up to a resource utilization of 87% (task set 14) with almost the exact same results as the EDF algorithm. Even though task set 15 has a utilization of 80%, the amount of experimental overhead seems to be significant enough to account for this failed task set.

To visualize the schedulability results from the SCT algorithm graphically, we have plotted the percentage of deadlines missed against the resource utilization. It is clear that the number of missed deadlines starts to increase at approximately 82.5-87%, which correlates with the practical and theoretical results we have shown above and is very similar to the EDF algorithm results. This upper bound is most likely an immediate side effect of the experimental overhead (which was approximately 10%), so for utilization percentages up this mark the practical results match the expected theoretical results almost exactly.

Also, as in both the RMA and EDF results, there is an increase in missed deadlines at 80% utilization for task set 15, but these missed deadlines are likely due to the experimental overhead (due to task context switching, thread management, and task period timer overhead).



6. Analysis of Failure

In order to emulate a common real-time system in which every task compute cycle that is scheduled must be completed, we chose to let our compute cycles compound on top of one another instead of having them reset at every deadline event. This allows us to measure the impact of cascading failures by a single task on the rest of the tasks in a given set (as well as on the entire application and system as a whole).

Table 14: Task deadline data from the RMA tests

Task Set ID	Algorithm	Number of Tasks	Number of Tasks with Missed Deadlines
1	RMA	4	0
2	RMA	2	1
3	RMA	2	0
4	RMA	3	1
5	RMA	3	0
6	RMA	3	1
7	RMA	3	2

8	RMA	3	0
9	RMA	3	0
10	RMA	3	0
11	RMA	3	0
12	RMA	3	1
13	RMA	3	1
14	RMA	3	0
15	RMA	8	3

Remarks: Task set 7, with a resource utilization of 2.16, has more than one task with missed deadline events. On the other hand, all other task sets with a missed deadline only have a single task out of the entire set with a missed deadline.

Table 15: Task deadline data from the EDF tests

Task Set ID	**Algorithm**	**Number of Tasks**	**Number of Tasks with Missed Deadlines**
1	EDF	4	0
2	EDF	2	1
3	EDF	2	0
4	EDF	3	1
5	EDF	3	0
6	EDF	3	1
7	EDF	3	3
8	EDF	3	0
9	EDF	3	0
10	EDF	3	0
11	EDF	3	0
12	EDF	3	1
13	EDF	3	1
14	EDF	3	1

15	EDF	8	3

Remarks: Task set 7, with a resource utilization of 2.16, has missed deadlines for every task. On the other hand, all other task sets with a missed deadline only have a single task out of the entire set with a missed deadline.

Table 16: Task deadline data from the SCT tests

Task Set ID	Algorithm	Number of Tasks	Number of Tasks with Missed Deadlines
1	SCT	4	0
2	SCT	2	1
3	SCT	2	0
4	SCT	3	1
5	SCT	3	0
6	SCT	3	1
7	SCT	3	2
8	SCT	3	0
9	SCT	3	0
10	SCT	3	0
11	SCT	3	0
12	SCT	3	1
13	SCT	3	1
14	SCT	3	1
15	SCT	8	3

Remarks: Task set 7, with a resource utilization of 2.16, has more than one task with missed deadline events. On the other hand, all other task sets with a missed deadline only have a single task out of the entire set with a missed deadline.

Also, a single missed deadline in task set 15 appears to have impacted the schedulability of a couple other tasks in the set for every scheduling algorithm. Furthermore, our data from each algorithm shows that at least one task in this set was starved entirely due to the increased CPU usage by other tasks and the test fixture.

With the exception of task set 7 (which had a resource utilization far beyond 100%) and 15 (which had many different tasks and a high resource utilization), there was only one task that consistently missed deadlines for each task set that failed the schedule tests. For task set 7, which is a theoretically unschedulable task set, our raw data indicates that certain tasks are actually starved and never scheduled to execute on the CPU due to the overhead from QNX task management and missed deadline events. This is an expected result because there exists a subset of the tasks that requires at least 100% of the CPU cycles, so the remaining tasks in the set are never given a chance to compute. Furthermore, since the resource utilization was so large the deadline event handlers for the tasks that were starved were never invoked from their respective period timers, which is an indication that both the task and timer threads were starved from CPU access. For task set 15, our raw data indicates that multiple tasks suffer missed deadlines and at least one is starved entirely from access to the CPU. As with task set 7, this is an appropriate response using all three algorithms.

One interesting aspect from these results is that for tasks that are completely starved (i.e. no compute time or deadline events) we can see that they received such a low initial priority so that they were never unblocked from their initial start semaphores in order to begin a test. In other words, the other tasks in the system consumed so much CPU access that the proxy scheduler's post to the starved task's semaphore was never given a chance to "register". This means that even for the EDF test in which such a task would always receive a high priority at some point since its deadline will eventually become the smallest it seems as though the task is blocked during the entire duration of the test.

These results are an indication that when the resource utilization and task set size increase to a point where the entire task set is no longer schedulable (e.g. the system is overloaded) only a single task will initially suffer missed deadlines. As the resource utilization or task set size increases beyond this point then it is likely that more than one task will miss a deadline event and that a task may even become starved. Depending on the requirements for the system, task starvation may or may not be seen as a system failure. In any case, this seems to be an appropriate response to an overloaded system because tasks will likely start to miss deadlines as time goes on and eventually become starved from a lack of CPU access.

In a practical real-time environment task starvation may not be appropriate, so instead of compounding compute cycles at every deadline event for a task, the deadline event count reset the current compute cycle if appropriate. This is an engineering decision that must be made based on the responsibility of each task in the set and the response-time constraints that are placed on the system.

This data also indicates that failures to meet deadlines are isolated to single tasks at a time. In other words, a single task failure will not ripple through the rest of the system and affect the schedulability of other tasks immediately. Significant increases in the task set resource utilization or the number of running tasks will have to occur before more than one task begins to suffer missed deadlines.

7. Reflection and Conclusion

As a team, there are several items of importance that we learned throughout the course of our experiments, as shown below.

1. Timing issues are incredibly important in real-time and embedded systems, especially when dealing with job scheduling in a system with explicit deadlines and bounded response-time constraints.
    a. Using hardware resources to drive timing primitives (e.g. timers) is a more reliable solution for accurate timing measurements than software resource.
    b. Timing solutions are not always deterministic and predictable, especially within the context of a real-time operating system, because there may be other background and foreground tasks happening that assume a higher priority than such solutions that have an immediate affect on their accuracy.
2. Timing becomes even more unpredictable when an application is instrumented to capture and log statistical data.
    a. It is necessary to strike a balance between the amount of information that is captured at runtime while profiling an application. Capturing too much information or allowing every kernel event to be inserted into the event stream can cause significant performance issues, whereas limiting the amount of information that can be inserted into the event stream can yield a lack of sufficient data for analysis.
3. Mathematical theory often places constraints that are too tight on the problem so that practical application results will not match the expected theoretical results.
    a. On average, each scheduling algorithm matched the expected theoretical results up to a resource utilization of approximately 87%.
        i. RMA - 89%
        ii. EDF - 87%
        iii. SCT - 87%
    b. These results were correct given the overhead that exists from both our test fixture and the operating system itself.