PERFORMANCE COMPARISON BETWEEN LINUX APPLICATIONS
WITH AND WITHOUT THE ADDITION OF THE QUANTUM
PLATFORM

A Thesis
Presented
to the Faculty of
California State University, Chico

In Partial Fulfillment
of the Requirement for the Degree
Master of Science
in
Interdisciplinary Studies
Embedded Computer Engineering

by
Gregory W. Slavin
Spring 2012
PERFORMANCE COMPARISON BETWEEN LINUX APPLICATIONS
WITH AND WITHOUT THE ADDITION OF THE QUANTUM
PLATFORM

A Thesis
by
Gregory W. Slavin
Spring 2012

APPROVED BY THE DEAN OF GRADUATE STUDIES
AND VICE PROVOST FOR RESEARCH:

Eun K. Park, Ph.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

Sharon A. Barrios, Ph.D.
Graduate Coordinator

Dale Word, M.S., Chair

Tyson Henry, Ph.D.
ACKNOWLEDGMENTS

I want to thank Professor Dale Word for taking the time to support me in my Interdisciplinary Studies degree and for educating me about the intricacies of embedded systems development. His teaching and mentoring really helped guide me in my own teaching efforts and I owe him a great deal of thanks and gratitude.

I would also like to thank Dr. Tyson Henry for supporting my Interdisciplinary Studies degree proposal and guiding me through the early stages of my degree. I want to thank Dr. Kurtis Kredo II for teaching me about the inner workings of Network Engineering and for providing a great example of how engineering courses should be taught. I want to thank Dr. Ben Juliano for assisting me in securing post-graduation employment and for supporting me in my teaching efforts.

Finally, I want to thank my wife Frieda and son Waylon for supporting me in my journey through graduate school. They were patient and understanding while I spent countless hours doing homework and research. Their love and support helped me reach my goal, and I love them both very much.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
</tr>
<tr>
<td>Purpose</td>
</tr>
<tr>
<td>Scope</td>
</tr>
<tr>
<td>Hypothesis</td>
</tr>
<tr>
<td>Limitations</td>
</tr>
<tr>
<td>II. LITERATURE REVIEW</td>
</tr>
<tr>
<td>Reactive Systems</td>
</tr>
<tr>
<td>Event-Driven Programming</td>
</tr>
<tr>
<td>Finite State Machines</td>
</tr>
<tr>
<td>Hierarchical State Machines</td>
</tr>
<tr>
<td>Real-Time Object-Oriented Modeling</td>
</tr>
<tr>
<td>Quantum Platform</td>
</tr>
<tr>
<td>III. METHODOLOGY</td>
</tr>
<tr>
<td>Porting the Quantum Platform</td>
</tr>
<tr>
<td>Dining Philosopher Problem</td>
</tr>
<tr>
<td>Load Generator</td>
</tr>
<tr>
<td>IV. RESULTS</td>
</tr>
<tr>
<td>V. CONCLUSIONS</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Performance Metrics</td>
<td>5</td>
</tr>
<tr>
<td>2. Load Generator Calibration Run Results</td>
<td>50</td>
</tr>
<tr>
<td>3. Results Summary</td>
<td>69</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Dining Philosopher Problem (DPP) Linux-only Flowchart</td>
<td>40</td>
</tr>
<tr>
<td>2. DPP Linux-only Background Loop</td>
<td>41</td>
</tr>
<tr>
<td>3. DPP Linux-only Statechart</td>
<td>41</td>
</tr>
<tr>
<td>4. DPP Linux-only Philosopher Tasks</td>
<td>42</td>
</tr>
<tr>
<td>5. Metrics in Rusage Structure</td>
<td>43</td>
</tr>
<tr>
<td>6. DPP with QP Class Diagram</td>
<td>44</td>
</tr>
<tr>
<td>7. DPP with QP Table_serving Method</td>
<td>46</td>
</tr>
<tr>
<td>8. DPP with QP Thread Scheduling Configuration Code</td>
<td>47</td>
</tr>
<tr>
<td>9. Load Generator Class Diagram</td>
<td>52</td>
</tr>
<tr>
<td>10. Load Generator Target Web Application</td>
<td>54</td>
</tr>
<tr>
<td>11. Max Resident Set Size (KB)</td>
<td>56</td>
</tr>
<tr>
<td>12. Soft Page Faults (I/O not required)</td>
<td>57</td>
</tr>
<tr>
<td>13. User CPU Time (uSeconds)</td>
<td>59</td>
</tr>
<tr>
<td>14. System CPU Time (uSeconds)</td>
<td>60</td>
</tr>
<tr>
<td>15. Block Input Operations</td>
<td>61</td>
</tr>
<tr>
<td>16. Block Output Operations</td>
<td>62</td>
</tr>
<tr>
<td>17. Voluntary Context Switches</td>
<td>63</td>
</tr>
<tr>
<td>18. Involuntary Context Switches</td>
<td>64</td>
</tr>
</tbody>
</table>
FIGURE PAGE

19. Mean Elapsed HTTP Response Times (nSeconds) ........................................ 66
20. Number of Times Philosophers Allowed To Eat........................................ 66
ABSTRACT

PERFORMANCE COMPARISON BETWEEN LINUX APPLICATIONS WITH AND WITHOUT THE ADDITION OF THE QUANTUM PLATFORM

by

Gregory W. Slavin

Master of Science in Interdisciplinary Studies
Embedded Computer Engineering
California State University, Chico
Spring 2012

Given the complexity of software engineering in general, and embedded and real-time software design in particular, many methods have been proposed to help manage complexity. Among these design strategies, object-oriented design and state machines are arguably the most pertinent for embedded systems designers. Both present limitations and complexities of their own that can lead to misuse and design failures. The Quantum Platform is a framework developed by Miro Samek that combines the ideas of object-oriented and event-driven programming with Hierarchical State Machines, in an effort to provide engineers with reusable components to aid in embedded software creation. Unfortunately, empirical performance data related to the Quantum Platform is not
publicly available. This thesis compares software developed using the Quantum Platform to more traditional procedural based software, with the goal of providing such data. Performance data generated by the two applications is captured and analyzed in an effort to quantify the differences in cost and ease of use between the disparate design methodologies. The expectation is that the ease of use, design elegance, and increased processor efficiency associated with the Quantum Platform will come at the cost of an increased memory footprint and run-time memory usage.
CHAPTER I

INTRODUCTION

Software Engineering is a complex discipline that often involves an equal mixture of science and magic. Architecting a solution that meets user specifications while still retaining concise readability and maintainability is a lofty goal, which in practice, often proves unattainable. Many software development projects flounder and few ever successfully deliver the products or services originally envisioned.

Embedded and real-time software systems, present even bigger challenges to designers. As Selic, Gullekson, and Ward (1994) note, “Real-time and distributed systems design has proven to be one of the most difficult and intricate engineering problems ever faced . . . primarily because of the complexity of the real world in which these systems operate” (p. xxiv). They go on to state that “this environment can be bewilderingly diverse, dynamic, and unpredictable - components fail at random, communications are corrupted, interruptions occur when they are most inappropriate, and so on” (Selic et al., 1994, p. xxiv).

Many design techniques and methodologies have been proposed to help deal with embedded and real-time systems design. The ultimate goal of all of these proposed approaches is to manage complexity and thus ensure success for projects. One such framework is the Quantum Platform developed by Miro Samek. His approach combines techniques such as event-driven programming and hierarchical state machines into a
unified approach which he refers to as the active computing model (Samek, 2009, p. xxix).

Purpose

The purpose of this project is to examine and compare two real-time system design paradigms, traditional procedural design and Quantum Platform-based design, with the goal of understanding the performance implications of each. This thesis will consist of two key components. The first component will be a review of literature describing the techniques upon which the Quantum Platform is based. The second component will consist of a performance test and data comparison of these two design methods when used in conjunction with multiple versions of a sample application built using the two methodologies.

Scope

The focus of this project, as described above, is to examine two distinct Real-time system design approaches with the goal of understanding their strengths and weaknesses and using this knowledge to evaluate their suitability for use in embedded and real-time systems. The first approach is to build a real-time application that runs on the Linux platform. Linux represents traditional Real-Time Operating System (RTOS) design philosophies, and consists of a scheduler and task creation and destruction facilities. The second approach utilizes a state machine-based platform, the Quantum Platform (QP), that includes various components that aid in the creation of embedded systems, but is non-traditional in that it is event-driven and utilizes state machines to control application flow as opposed to being controlled solely by an operating system.
scheduler. The QP-based application will primarily utilize two of the QP subcomponents, the Quantum Framework (QF) Real-Time Framework and the Quantum Event Processor (QEP).

This project will consist of two major sections. The first piece is a literature review of the theories upon which the Quantum Platform is based. This will include a look at Finite State Machines (FSM), Hierarchical State Machines (HSM), Real-Time Object-Oriented Modeling (ROOM), the Quantum Platform (QP), and event-driven programming.

The second and more significant part of the project will involve conducting performance tests on a sample embedded application, which presents sufficient resource contention to necessitate the services and sophistication offered by a scheduler. The sample application will be a solution for the classic Dining Philosophers problem (DPP). The author will develop one version of this application utilizing the POSIX threading components included with Linux. The other version of the DPP application is packaged with the Quantum Platform and will also be ported to the target Linux machine. Detailed run-time statistics, provided by the Linux kernel, will be captured to analyze performance differences between the two versions of the application. Additionally, a simple web application will be running under the Apache web server on the test machine, concurrently with the DPP application. Sample web traffic will be generated by a custom-built, multi-threaded Linux-based load generating application, running on a separate load generator machine. The purpose of this web application is to provide artificial load levels on the test machine to intentionally exercise and strain the scheduler while running these two versions of the DPP application. Through the combination of theoretical and
empirical analysis, the effects of the two designs on system performance and efficiency will be evaluated.

Hypothesis

Incorporating the QP into an embedded application design with Linux should have three observable differences from an application using a traditional Linux-only design. First, the QP version should consist of more object-oriented code than the Linux-only version. Second, the QP version should require increased overall memory usage, including both static footprints and runtime utilization levels, due to the extra required data structures and hierarchical state machine infrastructure code. Finally, the QP version should require lower CPU utilization rates than the Linux-only version, as the QP version uses run to completion state objects and thus spends less time using CPU cycles in the busy loop (i.e., idle task) in main().

The pieces of this hypothesis will be tested as follows:

1. Thorough code comparisons of the Linux-only and Linux/QP versions of the DPP application will be performed. Also, code structure including design techniques and patterns will be analyzed and overall code size and RAM footprints will be measured and compared.

2. Performance benchmarking tests, using a Linux-based http traffic generating application, will be executed and the statistics listed in Table 1 will be captured and analyzed. The resulting data will be used to evaluate the hypothesis.
Table 1

*Performance Metrics*

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Response Time</td>
<td>Elapsed Response – Request Time</td>
<td>Load Generator</td>
</tr>
<tr>
<td>ru_utime.tv_usec</td>
<td>User CPU Time</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td>ru_stime.tv_usec</td>
<td>System CPU Time</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td>ru_maxrss</td>
<td>Maximum Resident Set Size (Kilobytes)</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td>ru_minflt</td>
<td>Soft Page Faults (I/O Not Required)</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td>ru_inblock</td>
<td>Block Input Operations</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td>ru_oublock</td>
<td>Block Output Operations</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td>ru_nvcsw</td>
<td>Voluntary Context Switches</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td>ru_nivcsw</td>
<td>Involuntary Context Switches</td>
<td>Linux rusage structure</td>
</tr>
<tr>
<td># Times Philosophers Eat</td>
<td>Application Productivity as Indicated by Number of Times Philosophers Allowed to Eat</td>
<td>DPP and DPP with QP applications</td>
</tr>
</tbody>
</table>


**Limitations**

This project is faced with several limitations. First, the Quantum Platform comes with very limited built-in instrumentation. However, by integrating the Quantum Platform with Linux, the author will be able to take advantage of the extensive run-time statistics provided by the Linux kernel to capture detailed metrics with which a thorough
empirical performance analysis can be performed. Additionally, the load generating
application will include performance logging metrics.

Another limitation is that the Quantum Platform will have to be ported to the
target Linux machine. The book, *UML Statecharts in C/C++* by Samek (2009), provides
porting instructions, however so this should not be a problem.

A final limitation is that the sample web server and Dining Philosophers
Problem application are not necessarily realistic representations of real-world embedded
applications. However, such applications do exhibit resource and scheduling
requirements similar to those found in many real-time applications. Furthermore, the
Dining Philosophers Problem was specifically created to provide an application scenario
with built-in resource contention, and thus will help test the resource management
provided by the two competing design methodologies. Exercising the web server with the
external Load Generator component will allow for a comparison of how these two
methodologies perform under various load scenarios. Thus, including these two
application components will allow the author to compare and contrast the two
methodologies, and thus should provide a suitable testing platform for the hypothesis.
CHAPTER II

LITERATURE REVIEW

Several key topics addressed in this Thesis have received attention in various prior research efforts and publications. These topics include event-driven programming, Finite State Machines (FSM), Hierarchical State Machines (HSM), Real-Time Object-Oriented Modeling (ROOM), and the Quantum Platform (QP). Much of this prior work has direct relevance to this Thesis and thus an examination and review of this prior work is helpful in understanding the questions addressed herein.

Reactive Systems

A key distinction found in the literature on event-driven programming is the distinction between reactive and transformational systems. As noted by Selic et al. (1994), “Transformational systems start off with some initial data that is transformed by a series of computations into the desired output data . . . [and] once the output is produced, the system terminates execution” (p. 23). Selic et al. continue by stating:

In contrast, reactive systems generally do not terminate. Instead, they are involved in a continuous interaction with the environment. The environment generates input events at discrete intervals through one or more interfaces and the system reacts by changing its state and possibly generating output events. (p. 23)

Essentially, transformative systems are typical data processing applications while reactive systems are systems, such as typical embedded or real-time systems, that run continuously and respond to data and activities.
In the article, “Concrete Programming with Reactive Objects,” Gauvin and Smedley (2002) state, “reactive systems are characterized by an event driven collection of communicating components which respond to internal and external stimuli” (p. 74). In their paper, they “present an approach for enabling the visual development of reactive systems by combining object-orientation and message-passing” (p. 74). They note that one of the issues with developing reactive systems is that

the use of objects alone to develop such systems increase the difficulty of programming since the method call semantics of objects does not provide for asynchronous communication, thus producing a significant burden on the programmer to create such functionality. (p. 74)

In an attempt to resolve this issue, they propose a “new language construct, the reactant, to provide message-passing functionality to objects” along with a programming technique they call “concrete programming” to aid in developing reactive systems (p. 74).

In the article, “A Specification Idiom for Reactive Systems,” Sridhar and Hallstrom (2009), also address the subject of reactive systems and their associated benefits and challenges. They state, “Interrupt and event-driven applications constitute an important system class, with connections to desktop computing, embedded systems, and sensor networks . . . [and] we refer to this set of applications collectively as reactive systems” (p. 267). In this paper, they discuss how “split-phase operations – operations that involve a request, followed by a deferred out-of-context callback” can be used to programmatically model reactive systems (p. 267). They provide specific examples of this method of development with examples using an operating system named TinyOS and a programming language called nesC.
TinyOS, originally designed to support sensor networking applications, also provides features which are helpful for reactive, event-driven systems. Sridhar and Hallstrom (2009) proceed to describe how these features of TinyOS and nesC can be used to design reactive systems. Specifically they state:

Operations that involve input/output, and hence require the processor to wait, are implemented as split-phase operations. The basic idea here is to avoid putting the processor in a position where it has to block waiting for some I/O operation to complete. Instead, the operation is split into two phases. In the first phase, the component that is invoking the I/O operation (for example, sending a wireless message through the radio) calls a command to initiate the operation (e.g., send()). At this point, the component that receives this command immediately returns control to the caller, after registering the request. This prevents the caller component from blocking on the processor. At a later point, when the operation has completed, an event is signaled (e.g., sendDone()) to the calling component notifying it of the completion of the split-phase operation. (p. 268)

Thus, their approach helps address one of the key problems facing embedded systems developers in general, and state machine and event-driven developers specifically, which is how to model systems to handle code that requires blocking calls.

Event-Driven Programming

Event-based or event-driven programming is a topic closely related to reactive systems. Faison (2006) notes that “a software system is said to be event-based if its parts interact primarily using notifications” (p. xxi). He goes on to state that “notifications are basically signals sent from on part to another, in response to an event” (p. xxi). The question then becomes why is this approach superior to other embedded software development models? Faison answers, “A system designed this way is easier to build, test, and maintain than a traditional one . . . [and] the larger the system, the greater the benefits of an event-based approach” (p. xxi).
In the article “Event Driven Programming,” Carryer provides a better explanation of why event-driven programming is often a superior model for embedded systems. He states that in traditional programming paradigms, the “program was always in control of when the input occurred . . . [and] your code looked for input at specific places in your program, and waited for it to appear” (2005a). He goes on to state that, “While your program was looking for a particular kind of input, be it a mouse click or a key press, it wasn’t doing any other work” (2005a). This is obviously problematic in multitasking systems, where the device cannot afford to have the processor tied up in ineffective polling loops.

Carreyer offers event-driven programming as a possible solution to this dilemma. He describes event-driven programming as a sort of divide and conquer approach whereby the system is allowed to more effectively service multiple inputs. Specifically he states:

Under the event-driven programming model, the program structure is divided into two rough groups, Events and Services. An event represents the occurrence of something interesting. A service is what you do in response to the event. While events most often originate from the outside your program, such a mouse clicks, events can also be generated by other parts of the program. The program executes by constantly checking for possible events and, when an event is detected, executing the associated service. In order for this approach to work, the events must checked continuously and often. This implies that the services must execute quickly, so that the program can get back to checking for events. In order to meet this requirement, the service can not go into a state where it is waiting for some long or indeterminate time. (2005a, “Event Driven Programming,” para. 1-2)

In other words, he suggests, “in order for the event driven programming model to work, you must only write ‘Non-Blocking’ code” (2005a, “Event Driven Programming,” para. 2). Thus, his ideas then tie directly in to the solutions offered by Sridhar and Hallstrom in their article.
Carreyer next discusses the concepts of event checkers in his article. Since event-driven code responds to events, there must be a mechanism for the program to determine that an event has occurred for which it must respond. As Carreyer (2005a) states, “event checkers, then, are small pieces of code that test for the occurrence of an event” (“Event Checkers,” para. 2). He then describes how the key components of event checkers are state variables, which keep track of the current state and of the previous state. This is necessary so that the code can properly progress through the states of the program in the correct order. In other words, he is describing a mechanism to use event-driven programming to provide an organized approach to state machine programming.

The next event-driven programming concept Carreyer addresses is services. He states that “Services are simply the actions that you want your program to perform when an event has been detected” (2005a, “Services,” para. 1). He goes on to describe the key points of services as follows:

Services should be very compact functions that initiate the required action and quickly return. This allows the program to get back to checking for other events. The core assumption in event-driven programming is that you can check for events quickly enough so that none are missed. This can only happen if both the event checkers and the service routines execute quickly. Neither event checkers or service routines should enter into an indefinite loop. If you find yourself wanting to code a while loop for something other than stepping through an array, you probably need another event. The way to handle a situation like that is to have a service routine that starts the activity going, and an event checker to detect the end condition. (2005a, “Services,” para. 2-3)

Of particular importance, here is his point that the services should be as compact and efficient as possible. This design principle is true for event-driven programming in general, as well as for interrupt service routines and hierarchical state machine event handlers (Samek, 2009, p. 264).
Carryer goes on to discuss examples of how to implement event-driven programming principles. He concludes by stating that, “The event driven programming model represents a way of thinking about software that lends itself very nicely to situations where there are many possible inputs and the arrival time of those inputs will be unpredictable” (2005a, “Summary,” para. 1).

Finally, he states that while the event driven programming model alone is capable of dealing with many problems it may fall short when not tackling many of the more complex problems that you may encounter . . . however, when combined with the concept of state machines, event driven programming is capable of tackling almost any problem, independent of complexity. (Carryer, 2005a, “Summary,” para. 2)

Thus, while not the ultimate solution to every problem, the event-driven design paradigm can serve as a valuable tool in an embedded software designer’s toolbox, and can be paired especially well with state machines.

A key concept associated with event-driven programming is the concept of Inversion of Control. Samek (2009) explains that, “most modern event-driven systems are structured according to the Hollywood principle, which means ‘don’t call us, we’ll call you’ ” (p. xxviii). As opposed to traditional sequential programs, this means that “an event-driven program is not in control while waiting for an event; in fact, it’s not even active” (Samek, 2009, p. xxviii). He goes on to explain that only once the event arrives, the program is called to process the event and then it quickly relinquishes the control again...[and] this arrangement allows an event-driven system to wait for many events in parallel, so the system remains responsive to all events it needs to handle. (p. xxviii)
Therefore, using the event-driven paradigm can actually enable a software engineer to produce more work by using the processor in a more efficient manner than is possible with traditional programming.

Another key concept associated with event-driven programming is coupling. Simply stated, “coupling indicates the presence of interdependencies between classes or components” (Faison, 2006, p. 1). Ideally, “high-quality software should have a low degree of coupling between its components, because coupling introduces complexity, and complexity makes a system more difficult to understand, test, and maintain” (Faison, 2006, p. 1). Faison explains that the key problem which event-driven programming attempts to solve is coupling (p. xxi). It does this by architecting software as “a collection of independent parts that interact using event notifications” (Faison, 2006, p. xxi). By designing software using this event-based paradigm, a software engineer is able to create discrete components that communicate using well-defined interfaces and message exchanges.

Finite State Machines

A key software engineering pattern is the Finite State Machine (FSM). FSMs have been studied and used for years to try to bring order and understanding to algorithmic complexity. As Lafreniere (2000) states, “Implementing code using a state machine is an extremely handy design technique for solving complex engineering problems” (“Why Use a State Machine,” para. 1).

The key concept with FSMs is that they are “algorithm(s) that can be in one of a small number of states” (Gomez, 2004, p. 101). A software system is decomposed into
a set of unique or finite states. As noted by Arlow and Neustadt (2005), a state is “a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event” (p. 439). A software object can only be in a single state any point in time. Defining a system in terms of states helps to dramatically cut down on ambiguity and complexity, as given the current state, only a single part of the software that is specifically designed to handle that state, should execute.

Events and State Transitions are two additional closely related components of FSMs. As Arlow and Neustadt (2005) state, an event is “the specification of a noteworthy occurrence that has location in time and space” (p. 439). They continue by explaining that a transition is “the movement from one state to another in response to an event” (p. 439). Thus, as a software system executes, events occur, which in turn trigger transitions to different states.

Carryer (2005b) describes several ways in which finite state machines can be implemented in software. He first explains the, “most straightforward approach is probably a series of nested IF-THEN-ELSE statements” (2005b, “A State Machine in Software,” para. 1). Each IF tests for a possible state. When a match is found, the code contained within that clause of the IF is executed. While this is a solid approach, he notes that it works best for problems where “there are only a few states and only one or two paths out of each state” (Carryer, 2005b, “A State Machine in Software,” para. 3). He notes that, “as the state machine becomes more complex it will be easier to read and often generate more efficient code if the problem is expressed using a nested CASE structure, rather than the nested IF-THEN-ELSE clauses” (2005b, “A State Machine in Software,”
para. 3). Thus, as with general software design, a switch or CASE statement is often better and cleaner to implement and read than a group of nested IF statements.

An alternative approach to FSM design was offered by Gamma, Helm, Johnson, and Vlissides (1995) in their book *Design Patterns*. Essentially, they provided a new design alternative to the classic switch statement implementation. Gamma et al. explain that “the State pattern puts each branch of the conditional in a separate class . . . [and] this lets you treat the object’s state as an object in its own right that can vary independently from other objects” (p. 306). They called this approach the State design pattern.

Using the State design pattern has several consequences. First, this design “localizes state-specific behavior and partitions behavior for different states . . . [and] puts all behavior associated with a particular state into one object” (Gamma et al., 1995, pp. 307). What this means is that “the logic that determines the state transitions doesn’t reside in monolithic if or switch statements but instead is partitioned between the State subclasses” (Gamma et al., 1995, pp. 307). In other words, polymorphism determines which state is called in a given instance instead of a switch statement located in the controller.

Another consequence of using the State pattern is that transitions between states become explicit. Stated more precisely,

When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables . . . [whereas] introducing separate objects for different states make the transitions more explicit. (Gamma et al., 1995, pp. 307)
In other words, by defining a state as a separate class, the state has a separate identity from the other states, instead of merely being driven by a value in the current state variable being evaluated in a switch statement.

In his article, Lafreniere (2000) describes several methods to construct state machine-based programs. Like Carryer (2005b), he states that “a switch statement provides one of the easiest to implement and most common version of a state machine. [Where] each case within the switch statement becomes a state” (Lafreniere, 2000). He then describes some problems associated with this method of coding state machines. First, he notes that this method limits the developer’s ability to enforce the rules surrounding state transitions. He states that basically, “Any transition is allowed at any time, which is not particularly desirable” (Lafreniere, 2000). Another problem he describes involves the difficulty in sending additional data or messages to specific states. The problem is that “since the entire state machine is located within a single function, sending additional data to any given state proves difficult” (Lafreniere, 2000). Finally, this design is not well-suited to multithreaded applications and the “designer must ensure the state machine is called from a single thread of control” (Lafreniere, 2000).

Lafreniere presents a state machine design framework which resolves these issues using facilities provided by C++, in which states are no longer cases within a switch statement, but are instead individual StateMachine classes with their own data members and methods. In order to handle the issue of enforcing state transition rules, he proposes a state map. This map defines the accepted order for states to occur and only allows for transitions to occur if they follow the order specified in the map. Specifically, he notes, “when an event is generated, a lookup is performed to determine the state
transition course of action. [For which] there are three possible outcomes to an event: new state, event ignored, or cannot happen” (2000, “State Transitions,” para. 1). In other words, if the transition is valid, execution proceeds to the next state. If on the other hand, it is invalid either the transition is ignored, a fault occurs, or some predefined exception handler can execute.

Lafreniere addresses the problem of sending custom data or messages to a state by defining an EventData abstract base class. An event can define a subclass that inherits from this EventData class and contains specific data values it requires and pass this object along to the StateMachine class when that event is generated. Lafreniere describes this by stating, “when an event is generated, it can optionally attach event data to be used by the state during execution . . . [and] once the state has completed execution, the event data is considered used up and must be deleted” (2000, “Event Data,” para. 1). The EventData class provides a generic mechanism by which customized data can be sent to specific events if needed, and can be left empty or undefined if not.

The final problem Lafreniere addresses with his solution is to provide a thread safe solution. He achieves this by enforcing the rule that “once the state machine is executing, it cannot be interrupted” (2000, “State Transitions,” para. 2). In order to guarantee this, and “to prevent preemption by another thread when the state machine is in the process of execution, the StateMachine class uses semaphores” to provide exclusive access to event processing (Lafreniere, 2000, “Multithread Safety,” para. 1). Thus, thread safety is provided albeit with the added complexity of semaphore usage and management.

Simpler finite state machine design patterns are also available. For example, Fischer (1990) describes the use of a simple table-driven solution using the C
programming language. His solution involves creating a C structure that will be used to programmatically implement a state machine. He states, the four items necessary in the structure to define the state table are: the state, a valid event for that state, the next state to transition to for that event, and a list of functions to perform for the state/event combination” (“State Table,” para. 1). Once you have this structure built, you then create a driver program that will “read the events and drive the operation of the [state machine application] as specified in the state table” (Fischer, 1990, “Table Driver,” para. 1). The downside to using Fischer’s approach is that you do not resolve the issues raised by Lafreniere. However, as Fischer notes, his approach does provide certain benefits, including simplicity and the ability to easily add new features. He states that “for complex or multiple instance devices, organizing the source code as a mirror of the device’s actual operation provides a versatile, straightforward, and efficient means of writing software” (“Summary,” para. 1).

In many ways, both the approach defined by Lafreniere and the table-driven approach are similar to the State pattern introduced by Gamma et al. As Gamma et al. note, however, “the key difference between table-driven state machines and the State pattern can be summed up like this: The State pattern models state-specific behavior, whereas the table-driven approach focuses on defining state transitions” (1995, pp. 308). In other words, classes defined as part of the State pattern defines the key activities and attributes of the system in terms of the specific classes. Since the table-driven approach is relying on monolithic constructs to model the system, such as a switch statement which analyzes the current state, it instead focuses on using the transitions as the key players in providing system functionality.
Hierarchical State Machines

Finite State Machines (FSM) offered a great improvement over traditional, procedural-based programs for many problem domains. However, these systems are prone to suffer from a phenomenon referred to as “state explosion” (Samek, 2009, p. 68). Carryer (2005b) explains, “as the problems become more complex, especially if they involve multiple reactions to multiple stimuli, the number of required states can increase dramatically . . . [and] this problem, [is] known as state explosion” (“Extended Example,” para. 2). Samek notes that this phenomenon was largely due to the inability of FSMs to model repetitions between states. In other words, “a conventional FSM . . . has no means of capturing such a commonality and requires repeating the same actions and transitions in many states” (Samek, 2009, p. 68). This repetition causes code and design bloat and in the end adds system complexity.

Harel (1987), in his article “A Visual Formalism for Complex Systems,” provided the seminal treatise on hierarchical state machines. He began his article by stating that finite state machines were often ineffective in modeling complex systems, “because of the unmanageable, exponentially growing multitude of states, all of which have to be arranged in a ‘flat’ unstratified fashion, resulting in an unstructured, unrealistic, and chaotic state diagram” (pp. 232-233). As a potential solution for this problem, he introduced “diagrams, which we call statecharts, [that] extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication” (p. 231).

The statecharts described by Harel are a visual representation of hierarchical state machines. Harel noted that,
technically speaking, the kernel of the approach is the extension of conventional state diagrams by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components. (1987, p. 233)

At a higher level, he gave the equation: 
\[ \text{statecharts} = \text{state-diagrams} + \text{depth} + \text{orthogonality} + \text{broadcast-communication} \] (Harel, 1987, p. 233).

The key idea behind these statecharts is inheritance and aggregation. In other words, instead of repeating common themes throughout many states and leading to bloat and state explosion, commonality was factored out into superstates, which are roughly analogous to super-classes in Object-Oriented terms. Harel referred to this concept as orthogonality of the states. Specifically, he describes this phenomenon as “AND decomposition, [which is the] capturing [of] the property that, being in a state, the system must be in all of its AND components” (Harel, 1987, p. 242). According to Harel, this repetition “of course, is root of the exponential blow-up in the number of states, which occurs when classical finite-state automata or state diagrams are used, and orthogonality is our way of avoiding it” (1987, p. 243).

Ahluwalia (2004) also discussed the issue of commonality in his paper. As he describes, “in real life, many states handle most messages in a similar fashion and differ only in handling of a few key messages . . . [and] even when the actual handling differs, there is still some commonality” (p. 111). In order to deal with this,

hierarchical state machine design captures the commonality by organizing the states as a hierarchy . . . [and] the states at the higher level in hierarchy perform the common message handling, while the lower level states inherit the commonality from higher level ones and perform the state specific functions. (Ahluwalia, 2004, p. 111)

The key point is that the state handlers form a hierarchy with
the highest level state defining all the dummy “do nothing” handlers, thus the inheriting lower level states need not worry about messages that they do not handle . . . [and] all the handlers have been declared virtual so that they can be overridden by the deriving states. (Ahluwalia, 2004, p. 115)

Thus, by using this approach the amount of repetitive code is reduced, complexity is reduced, and greater efficiency is achieved.

Harel goes on to describe several other concepts that are integral to hierarchical state machines. First, he describes delays and timeouts. These are specifically targeted towards real-time systems, and provide a mechanism “to limit the system’s lingering in a state” (Harel, 1987, p. 254). In other words, timeouts provide a mechanism to trigger a time-expired event to wake up a system or break it out of a waiting or blocking status.

Another set of concepts Harel describes are Actions and Activities. He states that a key piece of hierarchical state machines “is the ability of statecharts to generate events and to change the value of conditions” (Harel, 1987, p. 256). In other words, statecharts can act as an event-driven system, both generating and responding to events. Instead of passively responding to state changes, hierarchical state machines provide a more holistic solution that actively generates state transitions, and then responds to them accordingly. Further than that, the states can contain other substates and form a hierarchy of states. As Harel describes, “this, in effect, results in a hierarchy of activities, with a statechart controlling the reactive behavior of each, in terms of its own input and output events and its subactivities” (1987, p. 258).

In his conclusion, Harel notes that people working with complex systems have for a long time appreciated the simplicity and appropriateness of the state/event approach but have lacked a
formalism for it that possesses certain elementary properties (such as depth and modularity) that are provided by most programming languages and by many conventional approaches to the physical and functional aspects of system description. (1987, p. 272)

Furthermore, he notes “the lack of these, as well as the exponential blow-up syndrome and the inherent sequentiality of conventional state machines, seem to have hindered serious use of states and events in the design of really large systems” (Harel, 1987, p. 272). In defining his statechart methodology he hoped to create a means to model systems that addressed these issues and provided more elegance and flexibility.

Since the introduction of statecharts and hierarchical state machines by Harel, many others have built upon these ideas. For example, these ideas now form the basis for UML state machines. As noted by Arlow and Neustadt (2005),

UML state machines are based on the work of Harel and tend to be used for modeling the life cycle history of a single reactive object as a finite state machine – a machine that can exist in a finite number of states . . . [and] makes transitions between these states in response to events in a well-defined way. (p. 439)

In accordance with Harel’s ideas, the UML includes time events, and other communication mechanisms such as call, signal, and change events (Arlow & Neustadt, 2005, p. 449). The UML also incorporates Composite states, which are “states [which] contain one or more nested submachines,” thereby, providing an implementation of the orthogonality concept described by Harel (Arlow & Neustadt, 2005, p. 458).

The idea of hierarchical state machines lends itself nicely to the creation of reusable application frameworks. Two such frameworks are the Quantum Platform and Real-Time Object-Oriented Modeling, both of which are discussed in more detail below. Another framework built upon hierarchical state machines is described by Dmitry Babitsky (2005) in his article “Hierarchical State Machine Design in C++.” In Babitsky’s
words, “the Quantum Framework source code available with Miro’s [2] book implements the hierarchy using a tree-like navigation from child to parent state and from parent to child that is rather complex.” Another fault he found with Samek’s code was the use of a switch statement inside of each event handler function and numerous macros. Babitsky had several goals including to “get rid of switch() statements over events in every state function . . . [and to] avoid macros because I believe they unnecessarily obfuscate the code.” The solution he proposes to do this is “based in part on the State design pattern in which State is a class and Event a pointer to a function in that class” (Babitsky, 2005).

In his implementation for hierarchical state machines, Babitsky utilizes the C++ programming language. He notes that because “C++ is naturally expressed with inheritance . . . by making state classes that can derive from each other, the need to create artificial hierarchy with tree-like code is eliminate[d].” Furthermore, he notes that because “the [C++] language automatically calls the most-derived virtual function processing the event. The need to handle events in every state function with switch() code is eliminated, making code easier to read and maintain.” Thus, by using the inherent built-in features of inheritance and polymorphism he is able to eliminate some of the complexities associated with the Quantum Platform.

Babitsky concludes his paper with some ideas about handling on_enter() and on_exit() functions and other runtime object management issues. He also provides a sample implementation based upon the coding ideas described in his paper. All in all, he made some interesting points and provided some alternatives to the approaches used by Miro Samek in the Quantum Platform.
Bordeleau, Corriveau, and Selic (2000) provide ideas in their paper about software design patterns, in addition to the State pattern, to achieve easier implementation of hierarchical state machines. According to Bordeleau et al.,

One of the most crucial and complicated phases of real-time systems development lies in the transition from system behavior (generally specified using scenario models) to the behavior of interacting components (typically captured by means of communicating hierarchical finite state machines). (p. 78)

In other words they identify a disconnect between the models used to model system behavior and the creation of components, represented in hierarchical state machines that implement this system behavior. In order to address this disconnect, they “describe one of the several behavior integration patterns [they] have identified to help designers define communicating hierarchical state machines from scenario models” (Bordeleau et al., 2000, p. 79). Specifically, they focus on the State Machine Integration pattern.

The State Machine Integration pattern “specifically addresses the design of a hierarchical state machine from a set of simpler state machines” (Bordeleau et al., 2000, p. 80). The authors describe how software systems are dynamic entities that will need to grow over time. One of the difficulties with traditional finite state machines is that it can often be difficult to insert a new state handler into the monolithic state handler code. Hierarchical state machines make this expansion somewhat easier by allowing software engineers to create new classes to represent new states and add them into existing hierarchies. The pattern described by Bordeleau et al., makes this process even simpler by defining a “component’s behavior as a set of integrated simpler state machines, each of which is associated with a set of scenarios” (p. 81). There is an overarching control state machine that is the controller for these simpler state machines. Thus, the pattern helps to
break system behavior into a series of “inter-scenario relationships” (Bordeleau et al., 2000, p. 81). As new functional needs emerge, new small components can be added and relationships to them can be defined and thus the controller can manage them. In essence, this pattern helps turn the hierarchical state machine into a plug and play platform where new state handlers can be easily added and plugged in.

Real-Time Object-Oriented Modeling

*Real-Time Object-Oriented Modeling*, by Selic et al. (1994) describes an approach to combining the ideas of hierarchical state machines, event-driven programming, and reactive systems design, specifically as related to real-time systems. In introducing their work, they note that “two fundamental drawbacks to using a general-purpose methodology for real-time system development are: inadequacy of modeling language . . . [and] implementation difficulties” (Selic et al., 1994, p. xxv). They then state, that
to avoid these pitfalls and to make optimal use of the powerful new features of object-oriented technology, we devised a new real-time modeling language, and a methodology based on this language . . . [and] we call the modeling language the Real-Time Object-Oriented Modeling language, or ROOM for short. (1994, p. xxv)

To a large degree, ROOM builds upon the hierarchical state machine work done by Harel. ROOM uses hierarchical state machines to model the behavior of the states contained in a software system. According to Selic et al., “ROOM was organized around the following three key elements: The operational approach, A phase-independent set of modeling abstractions, [and] The object paradigm” (1994, p. 35). Of these elements ROOM’s use of the object paradigm is most relevant to this thesis.
Selic et al., deviate from most traditional discussions of object-oriented programming in that they do not solely view objects as a “collection of data and its associated procedures” (1994, p. 48). Instead, they “define an object as a software machine, or as an active agent implemented in software, which is a component of a computer system” (Selic et al., 1994, p. 50). In ROOM, “programming with objects thus involves constructing different specialized machines in this way, and then interconnecting them in order to achieve the required higher-level functionality” (Selic et al., 1994, p. 50). In other words, they describe a software system as a set of connected “active agents” or machines that communicate amongst themselves to achieve the desired functionality. This is an extension of the description of hierarchical state machines provided by Harel.

A key difference between state objects in a ROOM system and objects in a traditional system relates to their activity patterns. As Selic et al. state, with traditional objects “an instance of an abstract data type typically ‘wakes up’ when one of its procedures is invoked, carries out the procedure, then ‘goes to sleep’ until the next invocation” (1994, p. 50). They continue by stating, “a software machine, on the other hand, may be active over extended periods of time, even when it is not being invoked, because it has its own thread of control” (1994, pp. 50-51). This idea of independent threads of control for objects is very important and ultimately became a cornerstone of the Quantum Platform discussed in the next section.

In this literature review, the author is solely focusing on the aspects of ROOM directly related to hierarchical state machines and the Quantum Platform. ROOM is much larger than that, however, and the book by Selic et al., offers much valuable information
on real-time systems design. Many of the ideas of ROOM have been incorporated into
other frameworks such as the Quantum Platform and the UML. ROOM is worthy of more
detailed analysis, and would make a good area for future study.

Quantum Platform

The Quantum Platform (QP) developed by Miro Samek (2009), builds upon
many ideas associated with state machines, object-oriented programming, and event-
driven programming. Specifically, it

brings together [the] two most effective techniques of decomposing event-driven
systems: hierarchical state machines and an event-driven framework . . . [and] the
combination of these two elements is known as the active object computing model.
(Samek, 2009, p. xxix)

Essentially, it is a framework for use in building applications designed as a series of these
active objects.

The QP consists of numerous layers, including the QEP Hierarchical Event
(2009), “the QEP event processor . . . executes state machines according to the UML
semantics” (p. 13). These state machines exist as active objects within the framework.

Samek goes on to state,

Active objects in QF are encapsulated state machines (each with an event queue, a
separate task context, and a unique priority) that communicate with one another
asynchronously by sending and receiving events, whereas QF handles all the details
of thread-safe event exchange and queuing. (p. 13)

In other words, the QF is a framework encompassing communication and scheduling
mechanisms within which the active objects can execute and exchange messages.
The QEP, or Quantum Event Processor, is the cornerstone of the QP. Samek (2009) states, “QEP is a generic, efficient, and highly portable hierarchical event processor that you can use in any event-driven environment, such as GUI systems, computer games, or real-time embedded (RTE) systems” (p. 150). It supports many of the features of the UML state machine specifications. Some of the most important features are, “full support for hierarchical state nesting,” “entry/exit action execution on arbitrary state transition topology,” and “full support of nested initial transitions” (Samek, 2009, p. 150).

At the heart of QEP are two classes, QHsm and QEvent. The QHsm class is used for the “derivation of state machines” and the QEvent class is used for the “derivation of events with parameters, or used as is for events without parameters” (Samek, 2009, p. 152). The key design factor surrounding the QP is this central object hierarchy. Each state machine must inherit from QHsm. In doing so these subclasses, inherit the init() and dispatch() functions which the framework uses to execute the state machines, and also data members which keep track of the current state (Samek, 2009, p. 252). Likewise, by using the standard QEvent class, or a subclass derived from it, the framework is able to more easily coordinate communications between the state machines.

The actual state handling code exists within the QHsm derived classes. As Samek (2009) notes, “in QEP, states are represented as state-handler functions that handle all events in the state they implement” (p. 158). In order to do this, the QHsm class receives an event in its event queue. These events represent signals in the UML specification. As Samek notes, “a signal in UML is the specification of an asynchronous stimulus that triggers reactions and as such is an essential part of an event . . . [and
basically] the signal conveys the type of occurrence” (p. 154). Each QEvent object has a data member sig that contains the value for the specific signal that is being sent. Inside the state-handler function, a switch statement evaluates the value contained in sig, and takes an appropriate action. In the event that “a hierarchical state handler function does not handle the current event, it returns the macro Q_SUPER() to the event processor” (Samek, 2009, p. 158). This is the mechanism which provides the implementation for the hierarchical state machine event management described by Harel. Events are handled as close to the bottom of the QHsm derivation chain as possible, but are passed up the chain towards the super class if necessary until a handler is found for that event.

At first glance, an intricate framework such as QP appears to be overkill. There are many layers of objects that handle various aspects of hierarchical state machine object management and communication. Samek (2009) explains,

it really takes more than ‘just’ an API, such as a traditional RTOS, to execute concurrent state machines . . . [because] state machines require an infrastructure (framework) that provides, at a minimum, run-to-completion (RTC) execution context for each state machine, queuing of events, and event-based timing services. (pp. xxviii-xxix)

This RTC idea is central to the functioning of QP. As Samek (2009) notes, “QF guarantees the universally assumed RTC semantics of state machine execution, by queuing events and dispatching them sequentially (one at a time) to the internal state machines of active objects” (p. xxix). In other words, the “processing of one event must necessarily complete before processing the next event” (Samek, 2009, p. 13). Because a state machine exists as a lone logical entity, it makes sense that one state must complete executing before another state can be made ready to run.
The RTC concept has numerous implications, many of them related to system performance and scheduling. According to Samek (2009), “the key advantage of RTC processing is simplicity” (p. 68). Applications are composed of state machines which exist independently. Since they are self-contained, the processing steps within the state are easier to visualize as program flow does not leave the state machine. When communications need to take place, events are posted to the queues of other active objects, but the current state machine keeps executing until it has finished its work for the current event. As Samek describes,

new incoming events cannot interrupt the processing of the current event and must be stored (typically in the event queue) until the state machine becomes idle again . . . [and] these semantics completely avoid any internal concurrency issues within a single state machine. (p. 68)

Samek (2009) further notes, the “biggest disadvantage is that the responsiveness of a state machine is determined by its longest RTC step . . . [and] achieving short RTC steps can often significantly complicate real-time designs” (p. 68). It is important to note,

that RTC does not mean that a state machine has to monopolize the CPU until the RTC step is complete . . . [but rather] the preemption restriction only applies to the task context of the state machine that is already busy processing events. (Samek, 2009, p. 68)

Samek goes on to explain that

in a multitasking environment, other tasks (not related to the task context of the busy state machine) can be running, possibly preempting the currently executing state machine . . . [and] as long as other state machines do not share variables or other resources with each other, there are not concurrency hazards. (p. 68)

Much of Samek’s book explains the design decisions behind the components of the QP. He also provides details on how to architect applications to run optimally using his platform. Additionally, he provides several sample ports of the framework to
various operating systems including uC/OS-II and Linux. The sections of his work dealing with porting are covered in the Methodology section of this thesis. In summary, however, the key idea behind porting ease for the framework is the platform abstraction layer (PAL), “which encapsulates all the platform-specific code and cleanly separates it from the platform-neutral code” (Samek, 2009, p. 389). As with most aspects of the framework, the PAL was designed with a focus on ease of use for application designers. In order to maximize implementation ease, “all software components of the QP event-driven platform, such as the QEP event processor and the QF real-time framework, contain a PAL” (Samek, 2009, p. 390).

The common thread among all of the techniques analyzed in this literature review is the goal of achieving efficient programmatic response to external stimuli. Event-driven programming and state machines are two mechanisms that developers can use to achieve this. However, implementing these techniques often resulted in “code riddled with a disproportionate amount of convoluted conditional logic that programmers call ‘spaghetti’ code” (Samek, 2009, p. 96). Hierarchical state machines were intended to bring more order, efficiency, and clarity to state machines and event-driven programming, by eliminating code repetition and bloat. The QP provides a means to simplify HSM development by providing a framework that takes care of “most of the heavy lifting for you” (Samek, 2009, p. 201). It achieves this at the cost of added complexity, in terms of a somewhat involved object model, and strict rules that developers must follow in structuring their applications. In essence, developing QP-based applications requires that “instead of thinking in terms of individual C or C++ statements, you should think in terms of state machine elements, such as states, transitions, entry/exit
actions, initial transitions, and guards” (Samek, 2009, p. 201). Samek (2009) explains that “you will start thinking at a higher level of abstraction about the best ways to partition behavior into states, about the events available at any given time, and about the best state hierarchy for your state machine” (p. 201). Thus, the QP offers many benefits, but in order to achieve these benefits software engineers must adapt their code and designs to it, and not vice versa. In short, the QP represents a major shift in software design and is not something that can just be tacked on to a traditional design.
CHAPTER III

METHODOLOGY

This project utilized a methodical approach to test and analyze the performance differences between the Linux-only and Linux with QP solutions. First, the author studied the Quantum Platform documentation, including framework design details as well as application design and integration best practices. The knowledge gained from this study was used to port the QP over to the target Linux machine.

Next, the author conducted performance tests on two versions of the Dining Philosopher application. In order to enhance the testing of the performance and resource utilization of the two Dining Philosopher Problem applications, a simple web server application was run concurrently on the same machine. This traffic represents HTTP requests as would be generated by a web browser running on a client computer or mobile device, and directed at an appliance machine running embedded Linux. Such a device might be a configurable router, streaming media server, or a similar machine capable of servicing multiple concurrent clients.

An older PC running Linux was used as the target platform for the performance testing. This PC is equipped with a Pentium 4 processor (operating at 1.5 GHz) and 512 MB SDRAM. This computer was chosen as the target computer for testing due to its similarity to modern high-end embedded systems. This allows for the
extrapolation of test results from this thesis to be applied towards modern real-time systems.

This project captured performance data using the performance metrics provided by Linux as well as metrics generated by the Load Generator and Dining Philosopher applications. This data was be collected and written out to text files and then subsequently imported into Microsoft Excel for numerical analysis.

All development work for this project was done on Linux using the KDevelop Integrated Development Environment. The Dining Philosopher and Quantum Platform code was written in the C programming language and compiled with gcc. The HTTP traffic generation application was written in the C++ programming language and compiled with g++. Finally, the sample web server application was written using PHP running under the Apache web server.

Porting the Quantum Platform

One of key challenges in using any framework is porting that framework to your target platform. For this thesis, the original plan was to port the Quantum Platform to the Micrium uC/Eval-STM32F107 evaluation board running the uC/OS-III operating system. In his book, Samek (2009) explains that

porting QF [the Quantum Framework] is relatively easy because QF has been designed from the ground up to be portable . . . [and] in particular, QF contains a clearly defined platform abstraction layer (PAL), which encapsulates all the platform-specific code and cleanly separates it from the platform-neutral code. (p. 389)

Samek describes the porting process as a trivial exercise, and states that “depending on the chosen RTOS/OS, the CPU architecture, and the compiler, porting QF might require
writing or modifying between 5 and 100 lines of code within the PAL” (p. 389). Based upon this assertion, the author believed that the process of porting the QP and QF to uC/OS-III would be a straightforward exercise.

Samek (2009) provides a sample port of the QP to uC/OS-II with the QP framework download and describes the port in some detail in Chapter 8 of his book. He states

I have carefully designed the provided QF port to uC/OS-II to be generic and applicable to most CPUs and compilers to which uC/OS-II has been ported…[and] in the case of porting QF to an external RTOS (uC/OS-II in this case), the RTOS forms an indirection layer that insulates QF from the CPU and the nonportable compiler extensions. (p. 421)

Immediately following this explanation, Samek states “what this means is that you still need to port the RTOS to the specific CPU and compiler, but you don’t need to modify the QF port to the RTOS because the RTOS API does not change” (p. 421). In other words, you can use the uC/OS-II port he provides on any target board running uC/OS-II. The only changes you need to make are in the porting of uC/OS-II to that target board.

The problem the author encountered was in porting the QP and QF to a new RTOS, in this case uC/OS-III. The author underestimated the changes needed to adapt the port from uC/OS-II to uC/OS-III. The primary difficulty lay in the fact that the API calls related to tasks and message queues are significantly different between these two versions of the Micrium RTOS. Furthermore, the port offered by Samek relies rather heavily on function calls embedded in C preprocessor directives and macros. This made the process of adapting and converting the port between the two versions of uC/OS time consuming and error-prone. Many errors passed through the compiler unseen and were caught instead by the less intuitive linker or not caught and all until runtime. Samek uses the
uC/OS objects such as message queues to handle the event message passing. Also, he leverages uC/OS to handle the memory block allocations. There are enough differences between these facilities between uC/OS-II and uC/OS-III to make the port fail to run correctly at runtime. In many cases, a general fault would occur and the code would wind up in the generic Cortex-M3 fault handler, App_Fault_ISR. Thus, in the end the efforts to port the framework to uC/OS-III were unsuccessful.

Fortunately porting the code to the Fedora 14 distribution of Linux proved much simpler. In order to port the framework to Fedora, first the framework code must be downloaded from the QPC SourceForge website. Then a simple modification to the .bash_profile to add QPC to the environment variables is required. Finally, the integrator must make, or compile, the framework code using the provided make file and gcc. Next, the application which is utilizing the framework must be built using the object library archive files output from the previous step. At this point, the application will be ready to run.

Overall, the complexity of porting the framework was largely dependent upon the target platform and operating system. The author’s main recommendation would be to choose an operating system to which the QP and QF have already been ported. If this is done, then the porting exercise will be relatively trivial, as was the case with the Linux port. Attempting to port the framework to a new operating system, for which no documentation exists, can be a very frustrating, time consuming and perhaps ultimately an unsuccessful endeavor.
Dining Philosopher Problem

In order to fully compare and contrast the performance of the traditional and QP-based design paradigms a sample application was required. This application needed to involve adequate complexity, resource contention, and multitasking to require scheduling activity and task switching. In this case, an ideal application scenario is provided by the Dining Philosopher Problem (DPP).

Edsger Dijkstra (1971) first publicly described the Dining Philosopher Problem in his paper “Hierarchical Ordering of Sequential Processes.” He set up the problem by creating a scenario in which there are “Five Dining Philosophers” (1971, p. 218). Dijkstra then explains, “The life of a philosopher consists of an alternation of thinking and eating” (p. 218). He relates how each of the five philosophers has a place at the table and wishes to eat some spaghetti. The philosophers are arranged at a circular table where each plate has a fork to the right and left of it. In order to eat, the philosopher must simultaneously use the right and left fork. As Dijkstra states, “there are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously” (p. 218). In other words, a philosopher cannot eat at the same time as either his right or left neighbor, as doing so would make it impossible to gain exclusive control over both forks that he requires to eat.

In his paper, Dijkstra goes on to propose a solution using both a binary semaphore to control each fork. He notes, however, that although [this solution] guarantees that no two neighbours are eating simultaneously – [it] must be rejected because it contains the danger of the deadly embrace…. [because] when all five philosophers get hungry simultaneously, each will grab his left-hand fork and from that moment onwards the group is stuck. (1971, p. 219)
He suggests using an array of state variables to resolve this concern.

Tanenbaum (2001) provides more details on a solution which uses the state array. His solution “uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks) . . . [and] a philosopher may move only into eating state if neither neighbor is eating” (2001, pp. 127-128). In other words, prior to taking possession of either fork, the philosopher first must test to make sure that both his left and right forks are available. If they are not, then he must remain in the hungry state. If both forks are available, he should acquire the left and then the right fork, each with an exclusive lock. The philosopher may then eat, release the forks, and return to the thinking state.

The DPP scenario was specifically designed by Dijkstra to model a situation in which insufficient resources are available to satisfy all components of a software system. As Samek (2009) notes, “DPP has been specifically conceived to make the philosophers contend for the forks, which are the shared resources in this case” (p. 448). Thus, the DPP provides an ideal scenario for the test applications in this project.

**Dining Philosopher Application—Linux Only**

Design goals for the Linux only version of the Dining Philosopher application were basic. The key goals for this application were simplicity and efficiency. This application represents the control for the overall experiment. Essentially this application represents the classic Linux and POSIX multithreaded application design philosophy.
Dining Philosopher Application—Linux
Only-Design

In order to represent the basic Linux and POSIX application design philosophy, this version of the Dining Philosopher application was designed using the classic Foreground/Background and Multitasking application design paradigms. As Samek (2009) notes, “the foreground/background architecture consists of two main parts: the interrupt service routines (ISRs) that handle external interrupts in a timely fashion (foreground) and an infinite main loop that calls various functions (background)” (p. 257). Samek describes how a “multitasking [system] is like foreground/background with multiple backgrounds,” in which “tasks . . . are typically structured as endless loops…[and] control resides concurrently in all the tasks comprising the application” (p. 259). This design is reflected in the flowchart in Figure 1.

The background processing portion of the basic Dining Philosophers application consists of an infinite loop residing in the main() function. The key code inside that loop is in Figure 2. This code uses the pseudo-random number generator built in to the C programming language to generate a number from 0 to 4 which is then used to access the state array and change the selected philosopher’s state from THINKING TO HUNGRY. This allows the related foreground task for that philosopher to execute the EATING state for that philosopher. The statechart for this version of the application is shown in Figure 3.

The real processing for this application takes place within the Philosopher tasks. There is one task associated with each Philosopher, one through five (e.g., app_task_philosopher1 through app_task_philosopher5). These tasks operate in an
endless loop as Samek describes in his section on Multitasking code. The key code inside these loops is shown in Figure 4.
```c
counter = 0 + (rand() % 5);

// Check to see if current philosopher is Thinking
if (state[counter] == THINKING){
    // If so, it's his lucky day -- dinner time
    state[counter] = HUNGRY;
}
```

*Figure 2. DPP Linux-only background loop.*

*Figure 3. DPP Linux-only statechart.*

This code verifies that this philosopher is hungry and his left and right neighbors are not currently eating. If all three of these conditions are true, the philosopher takes the mutex for the left fork and then takes the mutex for the right fork. At this point the philosopher is able to eat. The philosopher then immediately releases the right mutex, then the left mutex, and then updates its state to THINKING. If any of the three conditions is false, the philosopher remains in the HUNGRY state and the philosopher task will attempt to eat the next time it receives processor time.
```c
int left = ((phil_number + 4) % 5);
int right = ((phil_number + 1) % 5);

// Check to make sure both chopsticks are available
if ((state[phil_number] == HUNGRY) && (state[left] != EATING) && (state[right] != EATING)){
    pthread_mutex_lock(&app_mutex_chopstick1);
    pthread_mutex_lock(&app_mutex_chopstick5);

    printf("Philosopher 1 is Eating!\n");

    pthread_mutex_unlock(&app_mutex_chopstick5);
    pthread_mutex_unlock(&app_mutex_chopstick1);

    printf("Philosopher 1 is now Thinking!\n");

    // Reset philosopher to Thinking
    state[phil_number] = THINKING;
}
else if (state[phil_number] == HUNGRY){
    printf("Philosopher 1 is Hungry!\n");
}
```

**Figure 4.** DPP Linux-only philosopher tasks.

In order to achieve real-time scheduling for this application, and in an effort to match the scheduling parameters utilized by the QP version of the Dining Philosopher application, the basic version of the application used the same POSIX Pthread settings used by the QP application. Essentially, the Pthread threads are created with the SCHED_FIFO scheduling parameter. These settings will be described more fully in the upcoming section on the design of the QP Dining Philosophers application.

A final noteworthy aspect of this application is the performance logging it implements. Specifically, each philosopher task as well as the main() background task calls the getrusage() system function using the setting RUSAGE_SELF. According to
Kerrisk (2010), “The getrusage() system call retrieves statistics about various system resources used by the calling process . . .” (p. 753). This system call populates a rusage structure with the values shown in Figure 5. The code shown in this figure, as excerpted

```
fprintf(pfile, "%s	%ld:%ld	%ld:%ld	%ld	%ld	%ld	%ld	%ld	%ld	%ld	%ld	%ld	%ld	%ld	%ld
",
caller,
(long)usg->ru_utime.tv_sec, (long)usg->ru_utime.tv_usec, // user cpu time
(long)usg->ru_stime.tv_sec, (long)usg->ru_stime.tv_usec, // system cpu time
usg->ru_maxrss, // maximum size of resident set (kilobytes)
usg->ru_ixrss, // integral (shared) text memory size (kilobyte-seconds) - unused
usg->ru_idrss, // integral (unshared) data memory used (kilobyte-seconds) - unused
usg->ru_isrss, // integral (unshared) stack memory used (kilobyte-seconds) - unused
usg->ru_minflt, // soft page faults (i/o not required)
usg->ru_majflt, // hard page faults (i/o required)
usg->ru_nswap, // swaps out of physical memory - unused
usg->ru_inblock, // block input operations via file system
usg->ru_oublock, // block output operations via file system
usg->ru_msgsnd, // IPC messages sent - unused
usg->ru_msgrcv, // IPC messages received - unused
usg->ru_nsignals, // signals received - unused
usg->ru_nvcsw, // voluntary context switches (process relinquished CPU before time slice up)
usg->ru_nivcsw); // involuntary context switches (preempted or time slice expired)
```

**Figure 5.** Metrics in rusage structure.

from the performance_metrics() method in the file Main.c, then writes the values from the rusage structure out to a log file where the values can later be mined. Of particular interest to this thesis are the values contained in the fields, ru_utime, ru_stime, and ru_maxrss. More information about these values can be found in the Introduction and Results sections of this paper.

**Dining Philosopher—Linux and Quantum Platform**

Design goals for this application were similar to the Linux-only DPP application. The key goals for this application were maintaining simplicity and efficiency
while also incorporating features from the Quantum Platform. In order to best achieve this, the sample DPP application provided with the Quantum Platform was utilized with as few changes as possible. Samek developed this application, and includes it with the QP code, “to test the QP ports on various CPUs, operating systems, and compilers . . . [and that] the only platform-dependent file is the board support package (BSP) definition and sometimes the main() function” (2009, p. 461). The application included with the QP framework uses the design shown in Figure 6.

Figure 6. DPP with QP class diagram.

As the figure demonstrates, the two key components in this version of the application are the Table and Philo active objects. There are five Philo active objects (e.g., one for each philosopher in the Dining Philosopher problem). Each of these active objects maintains a set of state machines for the philosophers that represent all possible states. The “states” of these philosophers are initial, thinking, hungry, and eating. This version of the application uses the same basic state transition rules as depicted in the state transition diagram shown in Figure 3.

Samek (2009) notes that when a philosopher enters the thinking state he “arms a one-shot timer to terminate the thinking” (p. 448). The philosopher remains in the thinking state until this time expires. Samek then continues, “Upon receiving the TIMEOUT event, Philosopher[m] transitions to ‘hungry’ state an posts the HUNGRY(m) event to the Table active object” (p. 449). The Table object then checks to see if the forks for that philosopher are available, and if so allows the philosopher to eat. A TIMEOUT event controls the length of the eating action. Samek notes, that if the “forks for Philosopher[n] are not available . . . [the table] does not grant permission to eat . . . [and] Philosopher[n] remains in the ‘hungry’ state” (p. 449). The code which executes this logic is found in the Table_serving method in table.c, and is shown below in Figure 7.

A key design issue in both the QP and non-QP versions of the Dining Philosopher applications is the thread configuration. As mentioned previously, in the section of this thesis describing threading for the Linux-only version of the Dining Philosophers application, POSIX Pthreads supply the thread implementation. The use of Pthreads in the QP framework is important, as it is one example of the framework taking
switch (e->sig) {
    case HUNGRY_SIG: {
        BSP_busyDelay();
        n = ((TableEvt const *)e)->philoNum;
        /* phil ID must be in range and he must be not hungry */
        Q_ASSERT((n < N_PHILO) && (!me->isHungry[n]));

        BSP_displyPhilStat(n, "hungry ");
        m = LEFT(n);
        if ((me->fork[m] == FREE) && (me->fork[n] == FREE)) {
            me->fork[m] = me->fork[n] = USED;
            pe = Q_NEW(TableEvt, EAT_SIG);
            pe->philoNum = n;
            QF_PUBLISH((QEvent *)pe, me);
            BSP_displyPhilStat(n, "eating ");

            // Record Performance
            rusage_return = getrusage(process_info, &usage);
            if (rusage_return != 0){
                perror("getrusage failed for Table_serving");
                exit(1);
            }
            performance_metrics(&usage, str_output);
        }
        else {
            me->isHungry[n] = 1;
        }
        return Q_HANDLED();
    }
}

Figure 7. DPP with QP Table_serving method.

advantage of underlying OS infrastructure as much as possible. In this case, it also allows
the framework to take advantage of Pthread scheduling flexibility. Specifically, the QP
framework configures threads to use the SCHED_FIFO scheduling policy. As Samek
(2009) mentions, he assumes “that QF (Quantum Framework) is going to be used in real-
time applications” (p. 435). He suggests that he tries “to use as much as possible the real-
time features available in the standard POSIX API” (2009, p. 435). Samek then states, “In
Linux, the scheduler policy closest to real time is SCHED_FIFO policy . . .” (p. 438). In
Figure 8, you can see the pertinent portions of the thread scheduling configuration code,
extracted from the QActive_start() method in the file qf_port.c.
The key portions of the code shown in Figure 8, are the

```c
pthread_attr_t attr;
struct sched_param param;
...
pthread_attr_init(&attr);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
param.sched_priority = prio + (sched_get_priority_max(SCHED_FIFO) - QF_MAX_ACTIVE - 3);
pthread_attr_setschedparam(&attr, &param);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
```

if (pthread_create(&me->thread, &attr, &thread_routine, me) != 0) {
  /* Creating the p-thread with the SCHED_FIFO policy failed.
     * Most probably this application has no superuser privileges,
     * so we just fall back to the default SCHED_OTHER policy
     * and priority 0.
   */
  pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
  param.sched_priority = 0;
  pthread_attr_setschedparam(&attr, &param);
  Q_ALLEGE(pthread_create(&me->thread, &attr, &thread_routine, me) == 0);
}
```

Figure 8. DPP with QP thread scheduling configuration code.

According to Kerrisk (2010), “sched_get_priority_max() returns the maximum priority” (p. 741). He describes how combining the scheduling priority returned from sched_get_priority_max() with a scheduling policy of SCHED_FIFO will assign the highest possible real-time priority to a thread, and that threads “with the same priority are equally eligible for scheduling” (Kerrisk, 2010, p. 741). In other words, through using
these settings the active objects that drive this application are scheduled in a real-time fashion and are executed before any other tasks on the system, unless they relinquish control. Samek (2009) sums this up, by stating that “thread priorities are interpreted relative to the priorities of all other processes on the machine…[and] if we set the priorities high enough, no other process (or threads running within) can gain control over the CPU” (p. 440)

Using these real-time scheduling parameters is not without issues however. In particular, Butenhof (1997) notes the following:

One of the problems of relying on real-time scheduling is that it is not modular. In real applications you will generally be working with libraries from a variety of sources, and those libraries may rely on threads for important functions like network communications and resource management. Now, it may seem reasonable to make “the most important thread” in your library run with SCHED_FIFO policy and maximum priority. The resulting thread, however, isn’t just the most important thread for your library – it is (or, at least, behaves as) the most important thread in the entire process, including the main program and any other libraries. Your high-priority thread may prevent all other libraries, and in some cases even the operating system, from performing work on which the application relies. (p. 183)

Samek does not really address this concern directly, however, in the line of code that begins, param.sched_priority, he subtracts the value QF_MAX_ACTIVE – 3 from the value returned by sched_get_priority_max() and uses this result to set the priority level for his active object threads. The result of this, he states, is that “Assuming that a QF application will be real time, this port reserves the three highest Linux priorities for the system threads (e.g., the ticker, I/O), and the rest highest-priorities for the active objects” (Samek, 2009, p. 440). In other words, to mitigate the risks described by Butenhof, the framework reserves the three highest priorities for Linux functions, in an effort to allow the essential operating system work to execute.
The only real change the author made to this application is the addition of the performance_metrics() method. The declaration and definition for this method was added to the bsp.h and bsp.c files respectively. The method was then called from the Table_serving function in the file table.c. Basically, when the Table code executes the EATING state, the performance_metrics() call is made. This method is identical to that included in the Linux-only version of the Dining Philosophers application described above. This allows for efficient metrics gathering and comparison between the two versions of the application.

Load Generator

As previously stated, the purpose of the Load Generator application is to increase resource usage and contention levels on the target machine. In creating this contention, the author did not want to overload the target machine, but rather take it to the threshold of maximum resource usage. Overloading the target machine would be counterproductive in that once the maximum capacity is exceeded, the target machine ceases to function deterministically and thus the performance metrics gathered while running the Dining Philosopher applications would lose accuracy.

A methodical approach was used to ascertain the optimal running level for the Load Generator application. The author adjusted the # of threads and # of requests made and ran the Load Generator application with these configurations, until the breaking point was found. The threads and requests were then dialed back to achieve high target machine resource usage, without actually overloading the target machine. The data for these experimental Load Generator runs is shown in Table 2.
In designing the load generator application, there were three key “must haves.” First, the application had to be multi-threaded. This allows for an accurate simulation of traffic coming from multiple users. It also allows for more control in increasing usage to stimulate processor activity. Second, it had to incorporate a wait-time mechanism. In automated testing, wait-time is a mechanism that allows testing scripts to accurately simulate real user activity. In other words, a user would not request a URL and then immediately click on a link to request a subsequent URL. Instead, there would be some amount of time that would pass while they were waiting for the page to load in their browser, followed by time necessary to read the page. The load generator thus needed to include a mechanism to account for this wait-time. Finally, the load generator needed to
include the ability to capture and output timing information. This information would include web response timings that would illustrate performance degradation as usage ramps up.

**Load Generator—Design**

The Load Generator application consists of four key components, Main, RequestThread, Request, and Socket. Main is the entry point for the application, RequestThread is the primary application controller, Request is the worker process for the application, and Socket handles communications for the application. The diagram in Figure 9 shows how these objects interrelate.

As mentioned above, Main.cpp is the entry point for the application. In this object, various parameters including the target IP address for the test as well as the HTTP request text are set. The number of independent test threads to run, as well as the total number of requests to make by each test thread, are also specified here. For example, if the test will target IP address, “192.168.0.1” and the user specifies 5 threads and 30 requests, this means that application will spawn 5 RequestThreads which will each make 30 requests to that IP address, for a total of 150 requests made against the target web server.

The RequestThread class represents the application controller. This class spawns threads running Request objects for the number of threads specified by the user in Main, up to a total of 8 threads. RequestThread uses the non-member function make_request() to interface with the individual Request objects. This class uses Pthread threads, and passes data to these threads specific to each Request through a request_data
Figure 9. Load generator class diagram.
structure containing the relevant data. The make_request() method invokes the Request.make_request() method to start Request processing.

The Request class is the worker process for the Load Generator application. This class uses the Log, Socket, HTTPRequest, and HTTPResponse classes to interact with the target web server and document the results. All the relevant input values needed for execution are received through the make_request() method called by RequestThread. The clock_gettime() system call is called before and after the call to Socket.send_data() so that end-to-end response times can be captured for each HTTP request and response transaction. The HTTP responses from the web server are captured and parsed using the parse_response() method. All of the response time information and HTTP response values are collected and written out to log files using the log_response() method.

The Socket class provides communication capabilities to this application. Basically, it is a wrapper class around the Linux socket code. The constructor sets the socket parameters and the destructor closes the socket. The connect_server() method sets the IP and port values, calls init_connection() to set up the sock_addr_in structure, calls create_connection() to create a handle to the socket, and calls connect_socket() to establish the socket connection with the web server. The send_data() method actually sends the HTTP request to the web server and receives the related response. It uses the poll() system call to create a timeout that protects threads from non-responsive web servers.

This application was designed to be as efficient and object-oriented as possible. For the purposes of this thesis, its functionality is streamlined and straightforward. It makes simple HTTP requests and receives the HTTP responses, with
the primary goal of logging the response times for the end-to-end transaction. However, this application was designed with extensibility in mind, so that it can potentially be expanded into a more fully functional load testing tool at a later time.

Load Generator—Target Website

The target website for the Load Generator application is a basic PHP page running on Apache web server on the target machine. The page was intentionally kept very basic with no graphics. The page does include a call to the rand() random number generator function, the sqrt() function with the random number as a parameter, and the strftime() function to return the current date and time. Because the purpose of this page was to serve as a target for the Load Generator, and ultimately to aid in the stress of the target machine, these calls were made to force the web server to load the page each time a request for it was made, and to eliminate the possibility of caching. Figure 10 shows a screenshot of the webpage running inside a browser.

![Figure 10. Load generator target web application.](image)
CHAPTER IV

RESULTS

The hypothesis for this thesis had several clearly defined expectations. First, the QP version of the Dining Philosophers application was expected to be more object-oriented. This expectation is admittedly objective in nature, in that it is difficult to measure the degree to which an application is “object-oriented.” However, the object-oriented nature of the QP-based version of the application is undeniable. It consists of a clearly defined object model as is shown in Figure 6. The application code consists of two classes which inherit from parent classes in the QP framework. Additionally, the application code relies up underlying objects and types defined within the framework. The Linux-only version of the DPP application on the other hand consists of a single C language file. It is certainly not object-oriented and is instead an example of procedural design. The author attempts to make no judgments as to which design paradigm is better, just that they are very different. Thus, this aspect of the hypothesis was verified.

A second portion of the hypothesis expressed the expectation that the QP version of the DPP application would require increased overall memory usage due to the extra required data structures and hierarchical state machine infrastructure code. In order to measure the memory usage of the applications, several measurements were used. First, the compiled program size of the various versions of the applications was noted. The compiled image size of the Linux only version of the application is 16.2 KB. The image
size of the DPP version with debugging information included is 77.3 KB and without the debugging information is 19.7 KB. Thus, just from a ROM/RAM footprint perspective, the Linux-only version of the application, offers a 17.8% decrease in size over the QP version without the debugging information included.

Another measurement of the memory used by a process is represented by its resident size. According to Kerrisk (2010), “At any one time, only some of the pages of a program need to be resident in physical memory page frames; these pages from the so-called resident set” (p. 119). The measurement ru_maxrss mentioned in the Introduction is a measurement of the maximum resident set size of a process. Figure 11 shows the maximum resident set sizes for the various versions of the DPP application.

![Figure 11. Max resident set size (KB).](image)
As Figure 11 shows, the Linux-only version of the application had a maximum resident set size of 628 KB when running alone and 640 KB when running concurrently with the load generator. Likewise, the QP Release version of the application had a maximum resident set size of 564 KB when running alone and also a 564 KB size when running concurrently with the load generator. Thus, in comparing these two runtime scenarios (i.e., running alone and running concurrently with the load generator), the QP version of the application showed 10.2% and 11.9% memory usage decrease advantages respectively, as compared to the Linux-only version of the application.

Another memory metric captured and analyzed was the number of minor, or soft, page faults as reported in the ru_minflt member of the rusage structure as shown in Figure 12.

![Soft Page Faults (I/O Not Required)](image)

*Figure 12. Soft page faults (I/O not required).*
According to Johnson and Troan (2005), “Minor faults are memory accesses that force the processor into kernel mode but do not result in a disk access” (p. 67). They go on to state that, “These occur when a process tries to write past the end of its stack, forcing the kernel to allocate more stack space before continuing the process, for example” (p. 67). Thus, applications that are more efficient in memory usage should have less soft page faults. According to the data in Figure 12, the Linux-only version of the application running alone experienced 1,276 soft page faults and when running concurrently with the load generator experienced 1,335. The QP version of the application experienced 758 when running alone and 828 when running concurrently with the load generator. Thus, as reflected by this metric, the QP version of the application offered decreases in soft page faults of 40.6% when running alone and 38% when running concurrently with the load generator, as compared to the Linux-only version.

The final component of the hypothesis was the expectation the QP version of the DPP application would require lower CPU utilization rates than the Linux-only version. One of the advertised benefits of the Quantum Framework portion of the QP is that it simplifies programmers’ lives and increases efficiency. Samek (2009) notes:

A QF application has no more need to fiddle directly with critical sections, semaphores, or other such mechanisms. You can program active objects effectively and safely without even knowing what a semaphore is. Yet your application as a whole can reap all the benefits of multitasking, such as optimal, deterministic responsiveness and good CPU utilization. (p. 443)

Earlier Samek also stated, “A big advantage of multitasking is better CPU utilization because when some tasks are waiting for events, other tasks can continue execution, so fewer CPU cycles are wasted on polling for events” (p. 260).
In order to test Samek’s assertion that incorporating QP and the QF into an application’s design would improve CPU utilization, several different metrics were observed. These metrics are listed in the Introduction, and will be evaluated here in the order in which they are listed there. First, the User CPU time of the applications was measured using the ru_utime measurement of the rusage structure, returned by the getrusage() system call. Those measurements are reflected in Figure 13. As Figure 13 shows, the Linux-only DPP application had total User CPU times of 64,990 microseconds when running alone and 75,998 microseconds when running concurrently with the load generator. The QP version had total User CPU times of 151,976 microseconds when running alone and 174,973 microseconds when running concurrently with the load generator. This directly contradicts Samek’s claims, and represents a 57.2%
and 56.6% decrease advantage of the Linux-only versions over the QP versions when running alone and concurrently with the load generator, respectively.

The second CPU utilization rate is illustrated by the total System CPU times utilized by each process and is shown in Figure 14. The Linux-only version of the application had total System CPU times of 247,962 microseconds when running alone and 262,960 microseconds when running concurrently with the load generator. The QP version had System CPU times of 813,876 microseconds when running alone and 950,855 microseconds when running concurrently with the load generator. This represents decrease advantages of 69.5% and 72.3% in both scenarios of the Linux-only version over the QP version.

Figure 14. System CPU time (uSeconds).
Another measurement offered by the rusage structure is ru_inblock. According to Kerrisk (2010), this reflects the “Block input operations via file system” (p. 754). In other words, it is the number of times input was performed by the file system in servicing a request. This measurement was largely a non-factor as is shown in Figure 15.

![Block Input Operations](image)

**Figure 15.** Block input operations.

There were 120 block input operations performed by the QP version of the application that included debug information during one of the test runs, and thus it is included here for completeness. However, in another run of the same version of the application 0 block input operations were performed so it appears that this occurrence of 120 was an anomaly and is thus not analyzed any further in the course of this project.

The next metric captured and analyzed was the ru_oublock member of the rusage structure. According to Kerrisk (2010), this represents the “Block output operations via file system” (p. 754). In other words, it is the number of times output was
performed by the file system in servicing a request. The values captured are shown in Figure 16.

As reflected in Figure 16, the Linux-only version of the application running alone experienced 160 of these operations while the version running concurrently with the load generator experienced 176. The QP version running alone experienced 88 and 96 when running concurrently with the load generator. In this case, the advantage is to the QP version of the application and the differences reflect 45% and 45.5% decreases over the Linux-only version when running alone and with the load generator respectively.

Voluntary context switches, as reflected in the rusage member ru_nvcsw, is the next metric that was captured. According to McKusick and Neville-Neil (2005), “A voluntary context switch occurs whenever a thread must await the availability of a
resource or the arrival of an event” (p. 90). They go on to state that “a thread typically blocks each time that it requests data from an input device such as a terminal or a disk” (p. 90). As Hunt and Binu (2012) point out, “The cost of voluntary context switch at a processor clock cycle level is an expensive operation, generally upwards of about 80,000 clock cycles” (p. 37). Figure 17 shows the values captured during the test runs.

![Voluntary Context Switches](image)

**Figure 17.** Voluntary context switches.

As shown in Figure 17, the Linux-only version of the application experienced 1,065 voluntary context switches when running alone and 1,117 when running with the load generator. The QP version of the application experienced 17,868 when running alone and 20,217 when running with the load generator. The differences between these two versions of the application is quite dramatic and represents a 94.04% and 94.47% decrease on the part of the Linux-only application over the QP version.
Another metric captured from the rusage structure is ru_nivcs, which represents the number of involuntary context switches experienced by an application. According to McKusick and Neville-Neil (2005), “An involuntary context switch takes place when a thread executes for the duration of its time slice or when the system identifies a higher-priority thread to run” (p. 89). Hunt and Binu (2012) state that, “High involuntary context switches are an indication there are more threads ready to run than there are virtual processors available to run them” (p. 40). Hunt and Binu offer several mitigation strategies for this, with the common idea being that of “reducing the number of application threads being run on the system” (p. 40). Figure 18 shows the number of involuntary context switches experienced by these applications.

![Involuntary Context Switches](image)

*Figure 18. Involuntary context switches.*
As Figure 18 shows, the Linux-only version of the application experienced 3 involuntary context switches when running alone and 5 when running concurrently with the load generator. The QP version experienced 42 when running alone and 42 when running with the load generator. This difference represents a 92.86% and 88.1% decrease for the Linux-only version over the QP version, and reflects a clear advantage for the Linux-only version.

The final two metrics captured during the performance testing of these applications were not obtained from the getrusage() system call, but instead are derived from the test applications themselves. The first of these metrics is the measured mean response time of the HTTP requests made from the load generator to the web server.

Figure 19 shows these response times. In this diagram, the baseline figure, of 38,114.845 nanoseconds, represents the value obtained when the load generator application was running and the target machine was running neither of the DPP applications. When the load generator was run concurrently with the Linux-only version of the application, the mean response time was 38,477.355 nanoseconds. When the load generator was run concurrently with the QP version of the application the mean response time was 40,237.641 nanoseconds. These figures represent an advantage for the Linux-only version and a 4.37% decrease from the QP version.

The second metric generated from the applications was a count of the total number of times each application allowed the philosophers to eat. In other words, a counter was incremented each time a philosopher went into the EATING state. The values obtained for these counts are illustrated in Figure 20.
Figure 19. Mean elapsed HTTP response times (nSeconds).

Figure 20. Number of times philosophers allowed to eat.
In this case, the higher number is better as it represents more work being done by the application, in that the goal of the application is for the philosophers to get the chance to eat successfully. As Figure 20 shows, the Linux-only version allowed 760 philosophers to eat when running alone and 800 when running concurrently with the load generator. The QP version allowed 530 to eat when running alone and 600 when running with the load generator. In both cases, the numbers were higher when running with the load generator which was unexpected and unexplained. In any case, these numbers clearly represent an advantage for the Linux-only version of the application. Specifically, the Linux-only version experienced an increase of 43.4% and 33.3% when running alone and with the load generator respectively, over the QP application.
CHAPTER V

CONCLUSIONS

The testing described in the Results section provided an abundance of data. Table 3 summarizes this data. In evaluating whether the data confirms the hypothesis, it is necessary to evaluate each hypothesis component separately.

The first component of the hypothesis states that the QP version of the code should consist of more object-oriented code than the Linux-only version. It is true that the QP version is indeed more object-oriented than the Linux-only version. However, this fact by itself is not an indicator of superior quality. Object-Orientation does offer certain design benefits, but can also come at the cost of runtime efficiency. Also, as noted earlier in this thesis this object orientation comes at the cost of implementation ease. For example, porting the framework to a new operating system can be extremely challenging and can pose a barrier to its inclusion in a project. Thus, while indeed true, this component of the hypothesis is not entirely meaningful by itself.

The second component of the hypothesis states that the QP version of the application should require increased overall memory usage. In looking at the summary data shown in Table 3, the QP version of the application actually offers smaller runtime memory usage, in the form of a 10.2% smaller resident size, and increased runtime memory efficiency, as shown by the 40.6% decrease in the number of soft page faults over the Linux-only application. The memory footprint was 17.8% smaller for the
<table>
<thead>
<tr>
<th>Metric</th>
<th>Relates To</th>
<th>Result</th>
<th>Favors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Design</td>
<td>Usability</td>
<td>Linux-only</td>
<td>Procedural</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QP</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>Code Image Size</td>
<td>Memory - Footprint</td>
<td>Linux-only</td>
<td>16.2 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QP</td>
<td>19.7 KB</td>
</tr>
<tr>
<td>Resident Size</td>
<td>Memory - Runtime Usage</td>
<td>Linux-only</td>
<td>628 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QP</td>
<td>564 KB</td>
</tr>
<tr>
<td>Soft Page Faults</td>
<td>Memory - Efficiency</td>
<td>Linux-only</td>
<td>1,276</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QP</td>
<td>758</td>
</tr>
<tr>
<td>CPU Time - User</td>
<td>CPU - Usage</td>
<td>Linux-only</td>
<td>64,990 uSeconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QP</td>
<td>151,976 uSeconds</td>
</tr>
<tr>
<td>CPU Time - System</td>
<td>CPU - Usage</td>
<td>Linux-only</td>
<td>247,962 uSeconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QP</td>
<td>813,876 uSeconds</td>
</tr>
<tr>
<td>Metric</td>
<td>Relates To</td>
<td>Result</td>
<td>Favors</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------</td>
<td>-----------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Block Output Operations</td>
<td>CPU - Efficiency</td>
<td>Linux-only 160 QP 88</td>
<td>QP Decrease 45%</td>
</tr>
<tr>
<td>Voluntary Context Switches</td>
<td>CPU - Efficiency</td>
<td>Linux-only 1,065 QP 17,868</td>
<td>Linux-only Decrease 94.04%</td>
</tr>
<tr>
<td>Involuntary Context Switches</td>
<td>CPU - Efficiency</td>
<td>Linux-only 3 QP 42</td>
<td>Linux-only Decrease 92.86%</td>
</tr>
<tr>
<td>Mean HTTP Response Times</td>
<td>System Response</td>
<td>Linux-only 38,477.4 nSeconds QP 40,237.6 nSeconds</td>
<td>Linux-only Decrease 4.37%</td>
</tr>
<tr>
<td>Number Times Philosophers Eat</td>
<td>Work Accomplished</td>
<td>Linux-only 760 QP 530</td>
<td>Linux-only Increase 43.4%</td>
</tr>
</tbody>
</table>
Linux-only application, although at 19.7 KB, the QP application is still rather small. Thus, the facet of the hypothesis related to QP’s increased memory usage was both proved and disproved, depending upon which metric is used.

The final component of the hypothesis states that the QP version of the application should require lower CPU utilization rates than the Linux-only version. Five CPU-related metrics were analyzed during this thesis project. Of these five, four favored the Linux-only version of the application. Specifically, the Linux-only application used 57.2% less User CPU time, 69.5% less System CPU time, 94.04% less Voluntary Context Switches, and 92.86% less Involuntary Context Switches. The one metric where the QP version outperformed the Linux-only version was in Block Output Operations, where the QP version required 45% less operations. However, by far it appears that this aspect of the hypothesis was disproved, and that in actuality the QP version of the application required higher CPU utilization rates, and a consequence less CPU efficiency.

The final two metrics discussed in the Results section don’t directly deal with the hypothesis, but rather deal with the “real-world” usefulness and desirability of including the QP framework in an application. The first metric dealt with HTTP response times as an indicator of system response times for applications sharing resources with a QP application. In this case, the version without QP showed a decrease in response times of 4.7%. The second metric dealt with a measure of work accomplished by an application using QP compared to an application without QP. In this case, the version without QP accomplished 43.4% more work. Thus, when taken together these two indicators show that a non-QP system can accomplish more work and offer better response times than a similar application using the QP framework.
QP does offer advantages. It provides reusable components that can increase application design efficiency and reduce development times. The price for this flexibility is complexity, both in terms of porting and implementation. It also offers a mixed bag of performance over a non-QP implementation. It seems to offer more efficient memory management, but less efficient CPU usage. In the end, the decision on whether or not to use QP would need to be made on a case-by-case basis with attention paid to the unique constraints of the system being built.
REFERENCES
REFERENCES


