RICE UNIVERSITY

Microcontroller Programming for the Modern World

by

Thomas W. Barr

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Scott Rixner, Chair
Associate Professor of Computer Science and Electrical and Computer Engineering

Alan L. Cox
Associate Professor of Computer Science and Electrical and Computer Engineering

Marcia O’Malley
Associate Professor of Mechanical Engineering and Computer Science

Houston, Texas

April, 2014
Microcontroller Programming for the Modern World

by

Thomas W. Barr

Microcontroller development is much too hard, not only for beginners, but also for experts.

While the programming languages community has developed rich high-level languages and run-time systems that make programming traditional large systems easy and fun, the microcontroller developer languishes in a world of direct register access, incomplete C compilers, and manual memory management. For the past four years, the Rice Computer Architecture Group has been addressing this by developing Owl, an open-source microcontroller development system for the modern world. Owl includes support for the proven and easy-to-use language Python. It also supports Medusa, a new language designed specifically for embedded, concurrent programming. Finally, it introduces Hoot, a distributed computing environment that allows a programmer to treat a heterogeneous collection of controllers and networks as a single large application.

This thesis presents the design of Owl as well as a detailed quantitative evaluation of it. These results show that not only is it possible to run sophisticated system software on a microcontroller, but that doing so makes building applications much easier. The results and innovations presented here are adaptable to the embedded run-times of the future and have the potential to make microcontroller development easier for everyone.
Owl has been a collaborative project, incorporating design and implementation from contributors inside and outside of Rice. The core of Owl was provided as PyMite, an open-source virtual machine written by Dean Hall. From there, it has been re-architected and re-written by professors Scott Rixner and James McLurkin, myself, graduate student Andrew Lynch, Lingo Dai, and undergraduate students Rebecca Smith and Kathleen Foster. A large number of undergraduate research students have used Owl over the years, providing invaluable feedback and example code. I credit each and every one of them with making this project successful, interesting, and fun.

I also credit Valhalla, the Rice University graduate-student pub, and all its wonderful characters, with my success. Graduate school has the potential to be a draining and isolating experience. It has provided everything from a one-year crash course in business management to my closest friends. I can’t imagine success without them.

Finally, I want to credit the friends and family that have surrounded me, supported me, and loved me every day. Even though some days were spent building toys in the lab, many have been much less fun. I never would have been here without you.
Contents

Abstract ii
List of Illustrations viii
List of Tables xi

1 Introduction 1
   1.1 Contributions .................................................. 4
   1.2 Organization .................................................... 6

2 Background and Related Work 8
   2.1 Library-based systems ........................................... 8
   2.2 Embedded virtual machines ..................................... 10
      2.2.1 Python-on-a-chip .......................................... 11
      2.2.2 eLua ....................................................... 12
      2.2.3 Android ................................................... 13
   2.3 Actors ............................................................ 13

3 The Owl Embedded Python System 17
   3.1 Introduction ..................................................... 17
   3.2 The Owl Toolchain .............................................. 20
      3.2.1 Code entry ................................................. 21
      3.2.2 Compiler .................................................. 22
      3.2.3 Memory images ............................................. 22
      3.2.4 Python Modules .......................................... 25
   3.3 The Owl Run-time System ...................................... 25
      3.3.1 Python Interpreter ...................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2 Native C functions</td>
<td>27</td>
</tr>
<tr>
<td>3.4 Profiling and Analysis</td>
<td>32</td>
</tr>
<tr>
<td>3.4.1 Static Binary Analyzer</td>
<td>34</td>
</tr>
<tr>
<td>3.4.2 VM Profilers</td>
<td>35</td>
</tr>
<tr>
<td>3.5 Applications</td>
<td>36</td>
</tr>
<tr>
<td>3.5.1 Microbenchmarks</td>
<td>38</td>
</tr>
<tr>
<td>3.6 Results</td>
<td>40</td>
</tr>
<tr>
<td>3.6.1 Static binary analysis</td>
<td>40</td>
</tr>
<tr>
<td>3.6.2 Performance</td>
<td>41</td>
</tr>
<tr>
<td>3.6.3 Memory use</td>
<td>45</td>
</tr>
<tr>
<td>3.7 Conclusions</td>
<td>47</td>
</tr>
<tr>
<td>4 The Medusa Concurrent Programming System</td>
<td>48</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Medusa language</td>
<td>51</td>
</tr>
<tr>
<td>4.2.1 The Medusa language</td>
<td>52</td>
</tr>
<tr>
<td>4.3 Implementation</td>
<td>57</td>
</tr>
<tr>
<td>4.3.1 Pattern Matching</td>
<td>57</td>
</tr>
<tr>
<td>4.3.2 Mailboxes</td>
<td>59</td>
</tr>
<tr>
<td>4.3.3 Example</td>
<td>62</td>
</tr>
<tr>
<td>4.3.4 Deadlock and finite message bounds</td>
<td>65</td>
</tr>
<tr>
<td>4.3.5 Messaging and Python</td>
<td>66</td>
</tr>
<tr>
<td>4.4 Medusa Infrastructure</td>
<td>66</td>
</tr>
<tr>
<td>4.4.1 Medusa Threads</td>
<td>67</td>
</tr>
<tr>
<td>4.4.2 Task registry</td>
<td>68</td>
</tr>
<tr>
<td>4.4.3 Medusa Compiler</td>
<td>68</td>
</tr>
<tr>
<td>4.5 Evaluation</td>
<td>69</td>
</tr>
<tr>
<td>4.5.1 Messaging performance</td>
<td>70</td>
</tr>
</tbody>
</table>
# 5 Bridges for Embedded I/O

5.1 Polling ................................................. 73
5.2 Interrupts .............................................. 75
5.3 Bridges ................................................ 77
   5.3.1 Implementation .................................. 79
   5.3.2 Chronograph ..................................... 83
   5.3.3 Debouncing in Medusa ........................... 83
5.4 Evaluation ............................................. 84
   5.4.1 I/O Latency ....................................... 84
   5.4.2 Streaming data .................................. 87
   5.4.3 Event-driven Systems ......................... 90
5.5 Conclusions ........................................... 96

# 6 The Hoot Distributed Programming System

6.1 Design and Implementation .......................... 99
   6.1.1 Destination objects ............................ 99
   6.1.2 Message switch ................................. 104
   6.1.3 Transport modules .............................. 104
6.2 Applications .......................................... 105
   6.2.1 Autonomous car ................................. 106
   6.2.2 Interactive programming ..................... 109
6.3 Results ................................................ 111
6.4 Discussion ............................................ 112

# 7 MiniMedusa: A Fly-weight Messaging System

7.1 Design and Implementation .......................... 115
   7.1.1 Input format ..................................... 116
   7.1.2 Template generation ............................ 117
   7.1.3 Finite state machine generation .............. 119
**Illustrations**

1.1 Therac-25 operator interface[16]. ........................................ 3

2.1 Measured memory requirements for Candygram. ....................... 16

3.1 The Owl toolchain. ............................................................ 21
3.2 Body of autowrapped native function. .................................... 29
3.3 Artificial horizon hardware. .................................................. 37
3.4 Static binary analysis, from top to bottom, of the Owl virtual machine
   using wrapped functions, using the foreign function interface, of the eLua
   virtual machine and of an example program using SafeRTOS. .......... 39

4.1 Python event loop for autonomous RC car. .............................. 50
4.2 The new syntax of the Medusa language in EBNF. ..................... 53
4.3 A short example of Medusa syntax. The main thread sends a message to a
   child thread which then echos it back. .................................... 56
4.4 Basic Receive Statement. ...................................................... 60
4.5 Compiled bytecodes for the program in Figure 4.4. The program receives
   one message, checks it against one pattern, then compares it against a
   second if the first pattern fails to match. .................................. 61
4.6 A simple event loop written in Medusa that uses pattern matching to
   determine which event was received. ...................................... 63
4.7 By using more than one recv block, a thread may handle one type of
   high-priority messages regardless of low-priority messages that may be
   received before it. ............................................................. 65
4.8 The two message benchmarks: broadcast/collect and chain. The number of workers (3 in the figure) is configurable. 70
4.9 Average time to send and receive a message for different benchmarks. 71

5.1 Pushbuttons have a tendency to “bounce”, or oscillate between outputs when their state changes. (http://www.protostack.com/blog/2010/03/debouncing-a-switch/) 76
5.2 Blocking debounce in Python. This program is conceptually simple, but the time spent in line 14 while the program is sleeping blocks other threads from making progress. 78
5.3 The GPIO interrupt service routine. When the microcontroller detects a change of the input on a particular GPIO port, this routine is called. It reads the new value of the input into a port, then sends that value to the virtual machine. 81
5.4 Non-blocking debounce in Medusa. 85
5.5 Distribution of I/O latency for polling and bridges. 86
5.6 Data loss rates for different transmission rates. 89
5.7 Diagram of Event-driven Autonomous Car. 91
5.8 Diagram of Event-driven Traffic light controller. 95

6.1 Data flow in Hoot/Medusa. 100
6.2 Data flow in Hoot/CPython. 101
6.3 An embedded system with several different networks. 103
6.4 A Hoot program, written for Python. It communicates with a microcontroller running the program in Figure 6.5. 106
6.5 A Hoot program, written for Medusa. It communicates with a desktop computer running the program in Figure 6.4. 107
6.6 Data flow in Hoot/Medusa. 108
6.7 Modifications to the autonomous car project to run in a distributed manner. 110

7.1 A generated search DFA. ................................. 121
7.2 The generated control node for n2 in Figure 7.1. .................. 122
7.3 Message formats used by the MiniMedusa, JSON and Protocol Version of
the benchmark program. ...................................... 126
7.4 The MiniMedusa example program shown as a standard Medusa program. 126
7.5 From top to bottom: code size for a dummy program, the un-optimized
and optimized MiniMedusa programs and the JSON and Protocol Buffer
programs. .......................................................... 128

B.1 Autonomous car hardware. ................................. 141
B.2 Autonomous car schematic. ................................. 142
B.3 Autonomous car printed circuit board layout. ..................... 142
### Tables

3.1 Fraction of each workload’s running time spent in each bytecode and the average execution time of each bytecode. (Bytecodes that occur few or no times are not shown.) ......................................................... 42

3.2 Profiler results showing how dictionaries are used by the interpreter. .... 43

3.3 A snapshot of the heap, broken down by object type, for the artificialhorizon workload. ......................................................... 46

6.1 Sizes in bytes for messages compared in several serialization formats, including Medusa packed tuples. ................................. 113

7.1 Memory use and running time for the tested serialization formats on AVR8. Memory usage is listed in bytes. ................................. 127
Introduction

For every microprocessor in a traditional computer system, there are dozens of microcontrollers in cars, appliances, and consumer electronics. These systems have significant software requirements, as users demand elaborate user interfaces, networking capabilities, and responsive controls. However, the programming environments and run-time systems for microcontrollers are extremely primitive compared to conventional computer systems.

There are many low-cost, low-power, embedded microcontrollers that have very limited resources and performance, including the low end of Microchip Technology’s PIC microcontrollers and Atmel’s AVR microcontrollers. These microcontrollers are ideally suited for systems where extremely low cost or power constraints are primary concerns (e.g., sensors, actuators, toys, implantable medical devices, remote controls, tools, and appliances), and potentially warrant low-level programming in order to properly manage scarce resources. On the opposite end of the spectrum, “microcontrollers” like the ARM Cortex-A series are effectively conventional microprocessors and are used to power devices like the iPhone, iPad and other tablets/netbooks. These systems run operating systems and use higher level abstractions by necessity.

In between these two extremes lies a rich collection of mid-range microcontrollers that enable complex behavior from deeply embedded, often low-cost, systems. In recent years, these mid-range systems have become extremely popular. In fact, shipments of ARM mi-
Microcontrollers increased by 100% in 2010, most of which has been attributed to Cortex-M series devices [1]. In 2013, the Cortex-M class microcontroller became the best-selling microcontroller on the market [2] and is forecast to continue growing exponentially. ARM predicts that in 2015, the market for Cortex-M class microcontrollers will be around 18 billion units, over 4 times the market for Cortex-A class microcontrollers [1] and over 40 times the current market for PCs [3].

Currently, these controllers are almost always programmed in C, either by hand or generated automatically from models. This code, which runs at a very low level with no reliance on operating systems, is extremely difficult to debug, analyze, and maintain. At best, a simple real-time operating system (RTOS) is used to facilitate thread scheduling, synchronization, and communication [4, 5, 6, 7]. Typically, such RTOS’s provide primitive, low-level mechanisms that require significant expertise to use and do very little to simplify programming. As the capabilities of embedded systems increase, this situation is becoming untenable. Programming must be simplified to meet the demand for increasingly complex microcontroller applications.

As embedded systems become more complicated, they inevitably have to do more than one thing at a time. Even a simple, single controller device like a microwave oven must read from sensors, timers, and monitor the state of the door. The programmer must ensure that these tasks don’t interfere with each other and that a failure of one component does not lead to a failure of the system. For example, if the timer fails, the magnetron must still turn off when the user opens the door to avoid radiation burns to the user. When more controllers are added, working in parallel, the system becomes even more complex and intertwined.

The modern car is a prime example of such a system. It has dozens of microcontrollers and up to a hundred million lines of code [8]. Even for expert programmers, synchroniz-
ing the large number of tasks that code performs is nearly impossible. As such, the cost of software is a sizable fraction, anywhere between 15-25% of the total cost of each new car [9]. Even despite all this money and effort, bugs remain. Recent work has shown that automotive embedded software is littered with bugs that expose serious security vulnerabilities [10, 11, 12, 13, 14, 15].

Since this code controls a machine that has the potential to kill, failures can be deadly. In the 1980s, the Therac-25, a radiation therapy device, suffered several serious failures. The system was controlled by a PDP-11/23, a computer similar to the microcontrollers studied in this thesis. The developer used hand-coded software in several interrupt handlers to control a linear accelerator and an operator interface (Figure 1.1). If the operator edited fields too quickly, a rare synchronization bug occurred, activating the X-ray beam. This bug was not discovered in testing. As a result, at least six patients received massive overdoses of radiation. Three patients died [16].

Unfortunately, while this particular incident has been exhaustively studied and dis-
cussed as a cautionary tale, the actual process by which embedded systems are programmed has not significantly changed. Programmers and industry regulators have been more careful with safety-critical systems, but accidents continue. A race condition in a GE Energy power control system [17] caused widespread blackouts in 2005. More recently, the suspect systems in the Toyota unintended acceleration case used shared-state and hand-coded interrupt handlers [18].

1.1 Contributions

This thesis presents and evaluates a new software stack, Owl, that does fundamentally change the way embedded software is written. It is designed to make expert programmers more productive in building real-life applications and to prevent bugs in those applications. Owl programmers have their choice of programming language in the system. They can use Python, a modern language used in production systems everywhere from NASA to Google. Owl is both simple enough for inexperienced programmers to get started quickly and powerful enough for expert programmers to build full-scale applications. For example, it makes it easy for a programmer to interactively experiment with the peripheral set to figure out how to interface with devices. Experienced programmers frequently cite this as one of the slowest and most tedious parts of development. We have found that the interactive development capability of Owl makes this process much faster. Owl is provided under a liberal open-source license on our website at http://embeddedpython.org/.

It also presents a new programming language derived from Python and Erlang that we call Medusa. Medusa is supported alongside Python in Owl. While programmers can mix them in the same program, using Medusa brings significant safety and reliability benefits. It is based on the Actor model of programming. This model excludes, by design, many of the classes of bugs that are shown to be problematic in practice [19]. In other words, using
Medusa, it is *impossible* to introduce many of the kinds of bugs that current research uses complicated tools to detect (Chapter 2).

Medusa includes a feature called *message bridging* that, for the first time, extends the boundary of the Actor model from software threads to hardware peripherals. It transparently converts external events and interrupts to messages, received by threads that subscribe to a particular type of event. This thesis presents examples and experimental results to show that message bridging has performance similar to non-blocking, interrupt-based program while avoiding their pitfalls of reliability and complexity.

Finally, this thesis presents *Hoot* and *MiniMedusa*. These systems allow a programmer to treat a heterogeneous collection of microcontrollers as a single distributed system. Devices running Hoot, either traditional computer systems running the standard Python implementation or microcontrollers running Owl, can send messages from device to device using the same syntax used in Medusa to send a message from one thread to another. This makes building distributed systems, or adapting a concurrent system into a distributed system extremely easy. Additionally, the MiniMedusa library allows very small microcontrollers to communicate using the same message passing environment.

Throughout, the system is evaluated with microbenchmarks as well as realistic embedded applications. This serves several purposes. First, it demonstrates the scope and scale of high-level programming environments that can be practically built and used on microcontrollers. Second, it presents detailed quantitative analysis that illuminates how these systems perform in practice, insights that we expect will be translatable to other such systems. Third, it presents several innovations that dramatically improve such systems.

The environment described in this thesis is a complete platform for building embedded systems. Such systems can include everything from multi-core, 64-bit processors running Linux to the smallest microcontrollers with only a few bytes of memory. This system will
have impact far beyond the contributions presented here. It is already being used as the base for two student’s theses and it has the potential to support research far into the future.

1.2 Organization

- Chapter 2 discusses background and related work. It discusses historical techniques for programming embedded systems as well as more recent research. This research is largely separated into techniques based around compiled code interfacing with libraries and code that runs on top of a managed run-time. This thesis focuses largely on the latter class, but it contributes to both.

- Chapter 3 presents the Owl Embedded Python System. It presents detailed benchmark analysis on both microbenchmarks as well as realistic embedded applications. This shows the overheads of running a managed run-time system on a microcontroller. It addresses several of these overheads, presenting a new foreign function interface and representation of constant objects.

- Chapter 4 presents the design and implementation of the Medusa language. It formally defines the language and gives several examples of how programs are written and compiled. It details the infrastructure present in Owl to support it and measures that infrastructure with microbenchmarks.

- Chapter 5 presents message bridging, a new technique that automatically converts hardware events to software messages. It details the implementation in software and provides realistic applications to show how complete Medusa applications use message passing, including message passing from peripherals.

- Chapter 6 presents Hoot, Medusa’s support for message passing between micro-
controllers. It details several example applications and presents benchmark results. Hoot’s message encoding is compared to other serialization formats.

- Chapter 7 presents *MiniMedusa*, a library that allows extremely small systems to send and receive Hoot messages. It is evaluated on an 8-bit Atmel AVR microcontroller and compared to messaging libraries using JSON and Google Protocol Buffers.

- Finally, I conclude in Chapter 8.
Background and Related Work

Embedded programming is widely known to be difficult with tools and systems that lag far behind their corresponding large-system counterparts [20, 21]. To help make programming easier, a great deal of research has been done in the area of embedded languages and tools. This research largely falls into two categories: The first category focuses on systems with very limited memory capacity, 8 and 16-bit devices with only a few kilobytes of memory. These systems compile code directly to native instructions, linking in functionality via libraries. This results in small code sizes, though it is potentially dangerous because the run-time cannot prevent improper execution like array bounds violations. The other category of system interprets user code, usually stored as bytecodes. This is more resource intensive, though it is potentially safer since the run-time can check each operation before it executes. This thesis contributes to both categories.

2.1 Library-based systems

Traditionally, microcontrollers are programmed in C. To address the challenges inherent in C programming, a typical microcontroller tools vendor provides a suite of software to help analyze and debug low-level C programs running directly on the microcontroller or on top of a thin RTOS. These tools have evolved to perform static analysis to detect specific, common bugs: array bounds checking [22, 23] and string processing [24] amongst many
others. One of the most common themes of these systems is the detection of data races [25, 26, 27, 28, 29, 30, 31, 32].

Academic research has contributed to the development of real-time operating systems as well. The LiteOS [33] system is an extremely light-weight system that attempts to mimic a UNIX-like, traditional computer environment. Its authors have built an interactive console environment for it, a feature also present in Owl that has been shown to be very useful [34]. It also features a dynamic code loader, a feature that this thesis shows to be detrimental.

Piconet and TinyOS are both library-based systems that provide message passing functionality [35, 36]. These systems were designed for sensor networks where the topology frequently changes. Therefore, careful consideration was given to mobility and routing. On the contrary, Hoot pays no attention to these issues and defers to existing solutions, potentially solutions like Piconet and TinyOS.

TinyOS is a particularly interesting comparison since it also was developed alongside a programming language, nesC [28]. It has no native support for threads and long running operations are written in a split-phase manner. A block of code quickly starts such an operation and terminates. When the operation finishes, another block of code is run as a callback. This is a potentially difficult-to-use programming model. In fact, more recent research has provided a threading model that runs on top of TinyOS’s split-phase model [37]. Comparatively, Medusa is designed around threads that block for external events frequently. We show that this model leads to programs that are simpler and easier to reason about.

Arduino is a library-based system that provides an accessible platform for programming a simple development board in C++. Its main advantage is that it packages the compiler, programmer, and a rich set of libraries into an easily installable package. It has become incredibly popular in recent years and is used for everything from introductory program-
ming classes [38] to educational robots [39] to art installations [40]. Compared to Python, however, C++ is a very difficult language to work with. The programmer must manually manage memory, allocating and freeing space for the program to store data. There are also no runtime checks to verify that data is where the programmer thinks it is.

The managed run-time system in Owl avoids these problems. Memory is allocated and freed automatically, and the system only permits valid access to data types.

### 2.2 Embedded virtual machines

Despite their comparative rarity, embedded systems have long used a virtual machine, a software system that reads and executes a user program. This intermediate program can manage resources for the programmer and check for errors as a program is running. The Apollo Guidance Computer used such a system. The software that flew to the moon was made of a mixture of interpreted and compiled code [41].

Early commercial attempts to build embedded run-time systems, such as the BASIC Stamp [42], required multiple chips and have not been used much beyond educational applications. Academic projects have largely focused on extremely small 8-bit devices [43, 44]. These systems are built to run programs that are only dozens of lines long and are simply not designed for more modern and capable 32-bit microcontrollers.

The Java Card system ran an embedded JVM subset to allow smartcards to perform some limited computation [45]. These led to studies on code compression [46] and security [47], which could be transferred to our work. While there were some small proof-of-concept applications developed for it such as a basic web-server [48], the limited computational and I/O capabilities of smartcards rendered building large applications in Java Card impractical [49]. A more recent lightweight Java implementation, Squawk, runs on a slightly larger system, the Sun SPOT sensor node [50].
Recently, two open-source run-time systems for high-level languages on microcontrollers have been developed: python-on-a-chip (p14p) and eLua. p14p is a Python run-time system that has been ported to several microcontrollers, including AVR, PIC and ARM.

### 2.2.1 Python-on-a-chip

The p14p system [51] is a portable Python VM that can run with very few resources and supports a significant fraction of the Python language. The fundamental innovation of p14p is the read-eval-print-loop that utilizes the host Python compiler to translate source code into Python bytecodes at run-time. A p14p memory image is built from the compiled code object and then sent to the microcontroller. On the microcontroller, an image loader reads the image, creates the necessary Python objects, and then executes the bytecodes. In this manner, an interactive Python prompt operating on the host computer can be used to interact with the embedded run-time system over USB or other serial connection. This leads to an extremely powerful system in which microcontrollers can be programmed interactively without the typical compile/link/flash/run cycle. This process has been re-architected and improved in Owl, as described in Section 3.2.3.

The interactive Python prompt also gives unprecedented visibility into what is happening on the embedded system. Typically, a user is presented with a primitive command system that only enables limited interaction and debugging on the microcontroller. Debuggers, such as gdb, are needed for additional capability. In contrast, an interactive prompt allows users to run arbitrary code, print out arbitrary objects, and very easily understand the state of the system. This leads to much more rapid software development and debugging.

A key feature that p14p lacks is the ability to interact with peripherals in the embedded system. A significant fraction of embedded systems programming deals with I/O. The whole point of using a microcontroller is to interact with the environment, which requires
access to on-chip I/O, such as SPI, I2C, and UARTs, in order to talk with SD cards, GPS receivers, sensors, actuators, etc.

In p14p, native C functions can be wrapped in a Python function. This allows arbitrary C functions to be called from Python, enabling access to the microcontroller’s peripherals. However, the C functions are specialized to p14p, use cryptic macros to access parameters and return values, and must be rewritten for every platform to which p14p is ported. The maintainers of p14p leave it to the user to figure out how to best provide access to I/O devices for their platform and application.

The Owl system is based upon a snapshot of p14p from April 2010. Using experience gained from having a large user base at Rice, we then significantly expanded, re-architected, and improved the robustness of p14p.

2.2.2 eLua

Similarly, the eLua project is a portable Lua virtual machine that can run with very few resources [52]. The overall objectives and method of operation are very similar to p14p, so they will not be repeated here. However, there are two key differences between eLua and p14p.

First, eLua does include a Lua compiler on the microcontroller. This is possible because Lua is a much simpler language than Python, although there are still significant overheads involved when compiling code on the microcontroller.

Second, eLua presents an abstract device architecture for I/O access that each port is supposed to follow. This brings consistency, and presumably improved portability, to any eLua code that is written. The developers have provided ports of the abstract device architecture for many microcontrollers. However, these ports are built in much the same way as they are in p14p, and thus must be rewritten for every platform to which eLua is
ported. These projects otherwise use a very similar architectural approach to building an embedded run-time system.

2.2.3 Android

Android uses an interpreter, Dalvik, that runs on embedded systems, and executes code compiled from Java source. However, Dalvik targets a very different sort of embedded system than is discussed in this thesis [53]. Dalvik relies on the underlying Linux kernel to provide I/O, memory allocation, process isolation and a file system. It is designed for systems with at least 64 MB of memory, three orders of magnitude more than is available on ARM Cortex-M microcontrollers.

2.3 Actors

The library and virtual-machine based programming environments discussed so far in this chapter all are based on the procedural model of programming, by far the dominant model for programming languages. However, it is not the only way to specify how a computer should operate and it is not necessarily the best. An alternative, the actor model [54, 55], is a mathematical framework that directly represents event-driven systems. Actors represent distinct modules of computation, each responsible for a logical task. They receive a message, send messages to other actors, then decide how to respond to future messages. They do not, however, modify shared state. This structure makes program control-flow much easier to understand. The actor model cleanly separates and isolates system function [19].

The actor model was first described in 1973 by Hewitt, Bishop and Steiger [54] as a framework for artificial intelligence. Early work on actors established a strong mathematical and theoretical basis for the proving and verifying aspects of actor-based systems. These include formal definitions of what an actor system is [56], laws for actor systems [57], com-
mitment [58] and formal analysis of divergence and deadlock [55].

Many programming languages have been constructed to directly support the actor model, including Act [59], Rosette [60], Erlang [61], Scala [62], Go* and Timber [63]. Extensions are available for Java[64]. This work has collectively built useful infrastructure for practical actor-based systems, formal verification systems (enabled by the strong theoretical background of functional and actor programming) [65], efficient systems for generating and passing object references [66] and mechanisms for safely sharing some mutable state [67]. Many of these languages, most notably Go and Erlang, contain threading implementations that are lighter than operating system threads [61, 68] but are still more resource intensive than Medusa. Go and Scala require more than an order of magnitude more memory just to start their run-time libraries than our systems have in their entirety.

These systems have been used to build large scale systems, including the core of Facebook’s messaging architecture [69]. Google uses Go extensively internally. Erlang has a proven track record in large, microprocessor-based embedded systems, such as several telecom switches from Ericsson [61, 70].

Scala and Erlang both provide a messaging layer that is similar to Medusa and Hoot; they can transport complex objects and have sophisticated pattern matching abilities. The Go language provides a much lower-level messaging layer. It is incapable of transferring complex objects or doing any form of pattern matching.

The Erlang system itself has been used on embedded systems such as phone switches and base stations. However, these devices are very different from those targeted by Medusa. The Erlang website† states, “People successfully run the Ericsson implementation of Erlang on systems with as little as 16MByte of RAM.” This is over 200 times the minimum re-

*http://golang.org/
†http://www.erlang.org/faq/implementations.html
quirements of Medusa.

Other attempts have been made to extend the actor model to Python. Candygram is a pure-Python library that emulates the Erlang messaging model [71]. Unlike Medusa, Candygram does not introduce new syntax to the Python language. While this makes it more portable, representing message patterns is much more cumbersome. Additionally, Candygram implements processes as operating system threads. These are not portable to embedded systems and require much more system resources than our lightweight threads.

This was quantified by testing Candygram on a desktop computer running Python 2.7.5 ([GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin) A simple benchmark was adapted from Section 4.5 that spawns several threads and sends messages from one thread to the next. Each thread required 3.5 KB of memory, several hundred times what is required in our system (Figure 2.1).
Figure 2.1: Measured memory requirements for Candygram.
CHAPTER 3

The Owl Embedded Python System

3.1 Introduction

This thesis discusses the design and implementation of a complex run-time system that enhances programmer productivity. While such systems have been built in the past (eLua, p14p, etc.), little scholarly analysis has been done to show how they actually work in practice. How large are they? Where are the performance bottlenecks? Is it even possible to build and run realistic applications within the constraints of a mid-range microcontroller?

This chapter answers these questions by presenting the Owl Embedded Python System, an open-source embedded Python run-time system developed in part at Rice for this thesis. Owl is a complete Python development toolchain and run-time system for microcontrollers that cannot run a real operating system but are still capable of running sophisticated software systems. It is designed for ARM Cortex-M microcontrollers and includes an interactive development environment, a set of profilers, and an interpreter. Owl is derived from portions of several open-source projects, including CPython and Baobab. Most notably, the core run-time system for Owl is a modified version of Dean Hall’s Python-on-a-Chip, as discussed in Chapter 2.

The Owl system was originally designed for education. It was built as a part of Rice University’s r-one educational robot [72], designed to introduce first-semester students to
engineering concepts. Unlike other educational robots that are used by students in advanced classes primarily focused on learning programming skills [73, 74, 75], the r-one is accessible to beginning programmers. In practice, it has been highly successful. A class of twenty-five first-semester students, some of whom had never used a text editor before, became adept microcontrollers in just a few classes. In practice, Owl is extremely stable. Students experienced *no* interpreter crashes.

With this encouragement, Owl was developed into a much larger and more complete system for developing production and research software. Large applications were developed by both this author and other members of the Rice Computer Architecture Group, including a GPS tracker, a web server, a read/write FAT32 file system, and an artificial horizon display. Most interestingly, Owl was used to develop a complicated soft real-time system, an autonomous model car.

Using these applications, the system was benchmarked and analyzed in detail on a Texas Instruments Stellaris Cortex-M3 microcontroller, the LM3S9B92. This chip has 96 KB of SRAM, 256 KB of flash memory and was operated at 50 MHz. Using the results of this analysis, as well as experience from having a large user base at Rice, we significantly expanded, re-architected, and improved the robustness of the original p14p system.

Some highlights include:

1. A runtime profiler suite: Unlike previous work, the Owl system is highly instrumented with profilers. Users can see exactly how their code is running, allowing them to build larger and more efficient programs. Additionally, it allows systematic exploration of the embedded virtual machine design space. This data has led to many improvements, some of which are described below.

2. Loaderless architecture: Owl is rearchitected from p14p to not use any form of dy-
dynamic loader at run-time. The virtual machine was modified to be able to use objects (including compound objects) directly out of flash, copying them only if they are to be modified. This saves a massive amount of RAM.

3. End-user toolchain: We built a redistributable toolchain that allows end-users to program the microcontroller without needing to recompile the entire virtual machine. This includes drivers for Linux, OS X and Windows and support for an IDE. Additionally, it includes a runtime bootloader and user-level tools that can incrementally program the device without needing to reprogram the virtual machine and libraries.

4. Static binary analysis: We constructed a novel tool that visualizes the size of the of the virtual machine in flash, broken down by which portions of the source tree use the most space. This led to us being able to identify portions of the code that took up unnecessary space, such as GNU newlib’s snprintf implementation.

5. Two new external function interfaces: These are described in detail in Section 3.3.2.

6. Stack protection: Owl has optional runtime checks to ensure that stack frames are not overflowed. Additionally, it checks that uninitialized portions of the stack are not dereferenced.

7. Automatic type to object conversion (autoboxing): We expanded basic types in p14p to be automatically converted to an object when their attributes and methods are accessed. This means that basic types still have small memory overhead (since they don’t generally need attribute dictionaries), but still can be used like an object, as in traditional Python.

8. Module caching: Modules are cached so that only one instance is ever present in memory. This saves considerable memory when multiple user modules include a
common set of library modules.

As it currently exists, Owl is a powerful platform for developing realistic embedded applications entirely in Python. It has been successfully used by dozens of beginning programmers, none with previous embedded experience, showing that programming microcontrollers with a managed run-time system is not only possible but extremely productive.

Furthermore, the results gathered from analyzing Owl have provided insight into the nature of virtual execution environments on microcontrollers. Embedded applications tend to be far more control dependent than data dependent, which has significant impact on design trade-offs for the system. For example, garbage collection performance is less critical than name binding performance. Additionally, using an interpreter does not have a large program size overhead since much of the interpreter consists of large libraries that are also used by more traditional microcontroller applications.

As embedded systems continue to become more universal and become more complex, better programming environments are needed. The Owl system demonstrates that not only is it possible and practical to use a managed run-time system on such systems, but also that it makes programming complex embedded applications dramatically easier.

The next two sections (3.2 and 3.3) describe the toolchain and run-time of our system. Section 3.5, describes the applications we built using our system. In Section 3.6, I quantitatively analyze them. Finally, I conclude in Section 3.7.

3.2 The Owl Toolchain

Owl takes the form of a toolchain, running on a standard desktop computer, and a run-time, running on a microcontroller. The programmer interacts with the toolchain to input programs, monitor results, debug programs and monitor resource usage. The toolchain is
written entirely in Python, and is portable between UNIX-like systems such as Linux and Mac OS X as well as Windows.

This section describes the Owl toolchain, as shown in Figure 3.1. The toolchain transforms code on the host into a form that is directly runnable on the microcontroller. Code starts on the host and is entered either into a file or an interactive prompt. It is compiled into a code object by the standard Python compiler, which is then transformed into a memory image. This image is copied into the microcontroller’s flash memory. The images in flash can then be executed by the virtual machine, which will be described in Section 3.3.

3.2.1 Code entry

Programmers can enter code into the microcontroller using several methods:

1. Type Python code at an interactive prompt connected to the virtual machine on the microcontroller. This code gets dynamically compiled on the host, stored in SRAM on the microcontroller, and is then executed immediately.

2. Load code from a Python source file on the host via the interactive prompt. The file gets dynamically compiled on the host, stored in SRAM on the microcontroller, and
is then executed immediately.

3. Store code from one or more Python source files in flash on the microcontroller after being compiled on the host. This code can then be executed at any future time.

While p14p has the first capability, the latter two are unique to Owl. Therefore, one can program Owl systems using only Python without writing any C, needing a C compiler, or needing any specialized knowledge or hardware to program flash. Furthermore, code executed from all three sources can interact. At the command prompt, for instance, one can import a module that was previously stored in flash.

3.2.2 Compiler

All code in the Owl system is compiled by a Python compiler on the host system. The compiler emits a Python code object containing an array of bytecodes and a list of all the Python constants (integers, strings, etc.) that will be used in the execution of that particular block of code. These code objects can be used in any compliant Python implementation on the host or microcontroller.

By default, the toolchain uses the standard, unmodified Python 2.7 compiler. Its bytecodes are not modified in any way and can be executed by either Owl or the desktop Python virtual machine.

3.2.3 Memory images

Loaders and dynamic linkers are integral parts of traditional computer systems. They enable the compiler to generate code that is relocatable and can be combined with other libraries when the program is first run, as well as throughout its execution. On such systems, the overhead incurred by copying the binary is warranted due to the flexibility it generates. Furthermore, it provides an opportunity to save memory resources, as there are likely many
programs utilizing the same shared libraries.

One can instead compile non-relocatable, statically linked programs. Dynamic loading and linking are also an integral part of run-time systems for most interpreted languages. For example, the desktop Python implementation (CPython) uses the marshal format to store compiled source files: each object used by the source is loaded from the file, wrapped in an object header, and placed on the Python heap. Java .class files are loaded similarly [76]. The p14p system also uses a similar architecture.

These design decisions are predicated on the assumption that programs cannot be directly executed off the disk and that memory is effectively infinite. On an embedded system, the situation is different. First, flash is fast enough (often with 1–2 cycle access times), that programs can be stored in, and executed directly from, flash. Second, memory (SRAM) is scarce. Therefore, it makes sense to do everything possible to keep programs in flash, copying as little as possible into SRAM.

Microcontroller C compilers do exactly that by producing statically linked binaries, programmed into flash where they can be directly executed. When the microcontroller starts, only the sections of the binary that hold read/write data need to be copied into memory. As for the code, the instruction pointer can be set to point to the start of the program in flash and immediately begin execution.

The Owl system architecture also eliminates the need for dynamic code loading. Instead, the Owl toolchain compiles Python source code into a relocatable memory image, which contains all of the objects needed to run the user program. The run-time system then executes programs in these memory images directly from flash without copying anything to SRAM.

One of the key challenges in eliminating dynamic loading is handling compound objects which contain other objects. Compound objects created at run-time simply keep ref-
erences to the objects they contain, which are located elsewhere in the heap. However, the compound objects within memory images cannot be handled in this way. They must be relocatable and therefore cannot contain references. In a traditional system with a dynamic loader, such as p14p or Java, the compiler toolchain would generate special relocatable compound objects that are stored in a memory image. At run-time, the dynamic loader would first copy the relocatable compound object’s constituent sub-objects from the memory image to the heap. Then, the loader would generate the compound object itself on the heap, populating it with references to the sub-objects.

To avoid these copies, Owl introduces packed tuples and code objects, which store objects internally rather than using references. Each internal object is a complete Python object, with all associated header and type information. The packed types therefore enable the internal objects to be referenced directly, completely eliminating the need to copy them into SRAM. The compiler toolchain never places compound objects that are not packed into a memory image, guaranteeing that a memory image is completely usable without any modification. The run-time system recognizes these packed objects and handles them appropriately. These novel compound objects are therefore both relocatable and directly usable without the need for dynamic loading. They also can be stored anywhere, including flash, SRAM, or an SD card, and can even be sent across a network.

One complexity in the run-time system is when packed objects are stored in SRAM. In that case, if a reference to an internal object is created, that object must be copied and the reference must refer to the copy. Otherwise, if the packed object is garbage collected, the internal object will be lost, rendering the reference invalid. It is always safe to copy objects in this way, as all packed objects are immutable (a necessary condition to keep them in flash).
3.2.4 Python Modules

Unlike code compiled with a traditional C compiler, Python code is not linked into a single executable. In fact, it is not possible to do so in a dynamic language, as code written and executed at different times can interact. For instance, the modules for the standard library are compiled with the run-time system. The user's program is compiled and stored in flash via the run-time system's bootloader. Finally, code can be compiled and executed via the interactive prompt. All of this Python code can, and does, interact via Owl's module system.

A module represents a single Python source file. Each image has a corresponding name and is loaded when the Python `import` statement is executed. A module is compiled into a collection of code objects (one per function and one for the module scope itself) and then converted into a memory image, as described throughout this section.

Module memory images are stored sequentially in flash. When a module is imported by the currently executing code, the run-time system finds the appropriate module, creates the module name space, and executes any code associated with the module. There is no difference between modules that were compiled with the run-time system and those that were stored in flash by the user.

3.3 The Owl Run-time System

The Owl run-time system executes the memory images prepared by the toolchain. It interprets bytecodes, manages memory and calls external functions through both wrapped functions and the embedded foreign function interface.
3.3.1 Python Interpreter

The main component of the run-time system is the interpreter, which executes Python bytecodes from the memory image. These bytecodes operate with one or more Python objects on a stack. For example, they may add values (BINARY_ADD), load a variable name (LOAD_NAME), or switch execution to a new code object (CALL_FUNCTION). The Owl interpreter is derived directly from p14p, uses bytecodes identical to CPython, and matches the overall structure of the CPython interpreter. The heap is managed by a mark-and-sweep garbage collector which automatically runs during idle periods and under memory pressure.

One of the key advantages of using an interpreter is that the virtual machine is the only code that can directly access memory. If the virtual machine and all native functions are stable, it is impossible for a user to crash the system. Additionally, error detection code can optionally be included throughout the system to ensure that bugs inside the interpreter are detected at run-time and reported as exceptions. Normally, such events would be silent, difficult to trace, errors. Partly because of these features, the Owl system itself does not crash, even though it has been heavily used.

The Owl interpreter also includes several additions to the original p14p interpreter. First, Owl includes stack protection, via optional run-time checks to ensure that stack frames are not overflowed and that uninitialized portions of the stack are not dereferenced. Second, Owl includes transparent conversion from basic types to objects through autoboxing. Basic types are automatically converted to an object when their attributes and methods are accessed. This means that basic types still have small memory overhead, since they don’t generally need attribute dictionaries, but can be used like an object, as in traditional Python. Finally, Owl caches modules so that only one instance is ever present in memory. This saves considerable memory when multiple user modules include a common set
of library modules.

3.3.2 Native C functions

While the use of Python on embedded systems provides enormous benefits in terms of productivity and reliability, it is necessary to write portions of many programs in C in order to provide access to existing C libraries and to allow critical sections of code to run quickly. This is especially critical on a microcontroller where programs need to access memory-mapped peripherals via vendor-provided I/O libraries. For example, Texas Instruments provides a C interface to the entire peripheral set on their Cortex-M class microcontrollers, called StellarisWare, that simplifies the use of on-chip peripherals.

This section presents and compares two different techniques in Owl for allowing user code to call C functions: wrapped and foreign functions. While their implementations differ significantly, both systems make a native C library appear exactly like any other Python library.

While interpreters on the desktop have allowed programs to access external C libraries for some time, they have typically relied on features such as dynamic linking and run-time readable symbol tables that are too large for a microcontroller. In contrast, this section shows that a light-weight foreign function interface can be implemented in very little space without these features, and serve as an efficient bridge between Python and C code.

Providing the ability to execute native C functions introduces a way for the user to crash the system. However, all C functions must be compiled directly into the run-time system. Therefore, when discussing robustness and stability, they must be considered part of the run-time system itself. For peripheral and other library routines, such as StellarisWare, that are reused among applications, these functions are likely to be heavily tested and as stable as the rest of the run-time system. For application code that is ported to C for performance,
special care must be taken to preserve the stability of the system.

**Wrapped functions**

A Python program calls a function by loading a callable Python object and executing the `CALL_FUNCTION` bytecode. The callable object can be a *Python* code object or a *native* code object which serves as an interface to a native C function. In p14p, the native functions themselves are responsible for pulling arguments off of the Python stack, checking argument types, executing the actual code of the function, and generating a Python object as a return value. Argument and return values are read/written via a set of C macros that provide access to the Python stack. With this interface, functions can be written in C and then accessed or called like any other Python object. In fact, p14p provides the ability to embed such C code in the document string of a Python function.

Figure 3.2 shows the C code required to wrap a call to the native function with the prototype:

```c
void SysCtlPeripheralEnable ( uint32_t peripheral );
```

In this function, one Python integer is first type checked and then converted into the variable `peripheral`. This variable is used as the argument for the call to `SysCtlPeripheralEnable`. Since this function does not return anything, the Python object `None` is pushed back on to the Python stack and the function returns.

Note that `SysCtlPeripheralEnable` could have been inlined, but was instead called indirectly for clarity. In this case, the underlying function is a StellarisWare function that manipulates hardware registers to enable an on-chip peripheral. This cannot be written in Python and is a prime example of the value of native functions.

Given that the argument and return value marshalling is tedious and mechanical, it is a prime target for automation. The Owl toolchain includes an *autowrapper*: an automated
Variable declarations
PmReturn_t retval = PM_RET_OK;
pPmObj_t p0;
uint32_t peripheral;

If wrong number of arguments, raise TypeError
if (NATIVE_GET_NUM_ARGS() != 1) {
    PM_RAISE(retval, PM_RET_EX_TYPE);
    return retval;
}

Get Python argument
p0 = NATIVE_GET_LOCAL(0);

If wrong argument type, raise TypeError
if (OBJ_GET_TYPE(p0) != OBJ_TYPE_INT) {
    PM_RAISE(retval, PM_RET_EX_TYPE);
    return retval;
}

Convert Python argument to C argument
peripheral = ((pPmInt_t)p0)->val;

Actual call to native function
SysCtlPeripheralEnable(peripheral);

Return Python object
NATIVE_SET_TOS(PM_NONE);
return retval;

Figure 3.2: Body of autowrapped native function.
tool that generates a wrapper function for each function in a library. The wrapper is a small stub function that converts arguments from Python objects into C variables, calls the function, and, if necessary, converts the return value into a Python object and places it on the Python stack. In fact, the code shown in Figure 3.2 was generated by the autowrapper. Autowrapping functions is similar to the technique used by SWIG, which is commonly used to provide access to C code from high-level languages [77].

While this approach is conceptually simple, these conversions and type checks must be repeated for each function that is wrapped. This results in a massive amount of object code that is essentially repeated in the final binary.

**Embedded foreign function interface**

Given that the wrapper code can be generated mechanically, it is also a prime candidate for elimination. Foreign function interfaces, such as libffi*, have been developed for precisely this reason: to enable access to native functions from an interpreted language. Typically, a C compiler generates the code necessary to call a function. When one function calls another, it includes code that places arguments and a return address into registers and/or the stack according to that platform’s calling convention. Then, the address of the called function is loaded into the program counter. libffi does this process dynamically at run-time. A user can call libffi with a list of arguments and a pointer to a function; it then loads this data into registers and the stack and calls the given function. Java, Python, and PLT Scheme all use libffi to allow programmers to call external functions.

*eFFI*, a light-weight foreign function interface developed for the Owl system, builds upon these concepts in a fashion suitable for embedded run-time systems. It is a rewrite of libffi for the Cortex-M3 with some critical modifications. Specifically, embedded

*http://sourceware.org/libffi/*
systems do not typically include the ability to dynamically link native code, so all native libraries must be statically linked. eFFI links target libraries when the run-time system is compiled, therefore requiring much less support from either the user or the host system.

First, the header file of the library that is accessed by user programs is read into a variant of the autowrapping tool discussed in Section 3.3.2. This tool reads the names and signatures of each function to be exposed. For each function, a Python callable object is generated containing argument types, return type, and a reference to the function itself. Since this object is generated automatically at compile time, the programmer never needs to specify the number or types of arguments, eliminating one possible source of error when using foreign functions.

The signatures and addresses of all the foreign functions to be exposed to the virtual machine are stored in a compact data structure. These are each made available to the user as foreign function objects, stored in flash.

When a foreign function object is called at run-time, each argument is converted into a C variable and placed onto the C stack or loaded into registers. The address of the function is then written into the program counter, jumping into the function. When the function returns, the result is copied off the stack or out of registers, converted into the proper Python type (as specified in the foreign function object) and pushed back onto the Python stack.

This is a more complex process, since the actual C function call must be made by explicit code within the VM instead of being generated by the compiler. However, it is much more space efficient since this code is shared between all the foreign functions.

The key to the lightweight implementation of eFFI is that unlike the FFI implementations in desktop interpreters, foreign functions are not referenced by name in eFFI. Before the run-time system is compiled, arrays of function pointers are generated which are then
linked into the program. The (static) linker is responsible for including the library functions in the interpreter’s address space and placing a reference to them inside these arrays. The Python callable objects generated for each library function include indices into these arrays of function pointers rather than direct references to the functions themselves, eliminating the need for any run-time linking to determine the function address.

When a Python program calls one of these foreign functions, the interpreter first references the arrays of function pointers to find the address of the function to call. Since the function was already loaded into the interpreter’s address space by the compiler, there is no run-time library load process like there is in the desktop libffi implementation. From here, arguments are converted automatically from Python objects to C variables and the address of the function is loaded into the program counter, as in the desktop version.

This approach is particularly well-suited for the Stellaris Peripheral Driver library, which is included in ROM on many Stellaris microcontrollers. In addition to the instructions for each function, the ROM contains an array of function pointers in the same format (not coincidentally) used by eFFI. Therefore, callable objects can be generated that refer to entries within these tables directly.

### 3.4 Profiling and Analysis

There are many trade-offs in the design of an embedded run-time system. It is critical to measure the characteristics of both programs and the run-time system itself in order to better understand these trade-offs. Owl is the first embedded run-time system to provide a rich suite of performance and memory analyzers that offer insight into these design issues. With very few exceptions, space is more critical than performance for the studied embedded applications, so Owl is designed to favor memory efficiency over performance.

Building a general-purpose C profiler for microcontrollers is exceptionally difficult be-
cause the execution environment can vary so much between systems; there is no common file system and no heap. A profiler must be customized for each particular system. For example, the commercial toolchain used for this project (which costs thousands of dollars from a major vendor) leaves much of the provided gprof implementation incomplete. The user needs to write assembly code to be called during every function call, somehow recording data from registers into a file on the host for post-processing.

In contrast, building a profiler as a part of a managed run-time system is much easier. Since the interpreter indirectly executes the program one step at a time, it can easily be modified to record information about that execution. This information can be recorded in scratchpad regions of memory, or even inside of other objects on the heap. Since the virtual machine has complete control over the memory space, this is completely transparent to the user’s running program.

The Owl run-time system includes a line number profiler and a call trace profiler that can be turned on and off by the programmer. These are statistical profilers that operate similarly to gprof and provide information about the time spent on individual Python lines or functions. It also includes two profilers that measure the performance of the virtual machine itself. Additionally, the Owl run-time system also includes a memory analyzer, which would not even be possible to build for conventional C programs. Finally, the Owl toolchain includes a novel static binary analyzer to visualize which portions of the virtual machine take up the most space in flash.

These profilers are useful not only for writing high-performance applications but also for tuning the virtual machine itself. Furthermore, they demonstrate that building tools for run-time analysis is straightforward with an embedded interpreter. This comprehensive suite of measurement tools is unique to Owl; eLua and p14p have no built-in profilers.

Another student used this capability to build a very large tool to interactively explore a
program’s memory use. The tool stops a running program and transfers a copy of the entire Python heap back to the desktop. That heap is parsed back into separate objects which are then visualized on the screen. The user can explore different parts of their program to see where dynamic memory is being used at run-time. Such a system is only possible because all user data is managed by the run-time system. Relationships between data is explicitly encoded into Python objects and can be easily decoded.

All of these profilers operate transparently to the user and store all data directly within the Python heap. They work even when the microcontroller is disconnected from the host. Therefore, these profilers can be used in mobile, untethered systems, which is not possible with any other microcontroller profiler.

### 3.4.1 Static Binary Analyzer

The on-chip flash memory of a microcontroller constrains the complexity of the programs and data that the microcontroller can utilize. It is critical to use this scarce resource efficiently. Despite this, most embedded toolchains provide little feedback on flash utilization. The Owl static binary analyzer visualizes the size of the virtual machine in flash, broken down by which portions of the source tree use the most space. An example of its output is shown in Figure 3.4.

It uses the `nm` tool to extract the list of symbols from the compiled and linked binary. Then, it generates a graphical output showing the size of different parts of the system. Each C source file is shown as a band whose size is proportional to the space that code consumes in the final binary. The bands are sorted by category or directory in the source tree. Larger files are annotated with their file name. This tool was invaluable in identifying sections of the virtual machine that were unnecessarily large, such as newlib’s `printf` suite and wrapped functions.
3.4.2 VM Profilers

Owl includes a virtual machine profiler that periodically fires a timer and records which bytecode is currently executing. By combining this data with how many times each bytecode is executed, it is possible to measure the average running time of each bytecode, and its overall impact on running time.

Since Python is dynamically linked, the VM stores variable names as strings and maintains a dictionary mapping those strings to their values for each namespace. Every time a variable is accessed the interpreter searches this dictionary. This can take a long time since multiple variable names may be looked up during a single line of execution. Additionally, the lookup currently uses a linear scan, so it is inefficient. A dedicated profiler that measures the performance of dictionary lookups quantifies this inefficiency. These results are discussed in Section 4.5.

Python profilers

We built two different sampling profilers that allow a programmer to see which parts of their program take the most time to run. These profilers require virtually no modification to a user’s program. They periodically stop the program, record the contents of the execution stack, and continue. These records are stored in a compact form and require a constant amount of space regardless of execution time.

The line number profiler measures the time spent on each line of a Python file. When the profiler is started, a hardware timer fires periodically and inspects the stack frame to see which lines are being executed. An array of counters represents the number of profiler ticks seen in each line of source. The Python stack already contains enough information for the profiler to walk up it and determine which lines are being executed; no modifications to
the run-time program are required. A tool running on the host post-processes the data and presents the running time of each line with the original source code.

Alternatively, we have built a variant we call the call trace profiler. This profiler works more like gprof, showing the time spent in each function.

While these are not particularly complex profilers, they would be comparatively difficult to include in a C-based environment. Indeed, the commercial compiler used for this product does not include complete profiler implementations. In contrast, it is quite straightforward to optionally include profiler data inside the virtual machine. The VM then updates that data at run-time, keeping track of run-time behavior. This data can be reported to the user without modifying user code.

3.5 Applications

This thesis evaluates the Owl system on a range of microbenchmarks as well as realistic embedded applications built by a number of talented undergraduate assistants working in the Rice Computer Architecture group. In the embedded applications, the 9x92 is connected to a GPS receiver, three-axis accelerometer, three-axis MEMS gyroscope, digital compass, TFT display, microSD card reader, ultrasonic range finder, steering servo, and motor controller. The applications use these devices to implement an artificial horizon display (using the display and accelerometer), a GPS tracker (using the GPS, compass, microSD, and display).

The largest example built for this project is an autonomous miniature car, designed and built by several members of the research group and myself. The electronics from an off-the-shelf RC car (an Exceed RC Electric SunFire Off-Road Buggy) were replaced with a 9x92 microcontroller, a custom printed circuit board, and several peripherals. Photographs, schematics, and PCB layouts are provided in Appendix B for reference. The car is con-
Figure 3.3: Artificial horizon hardware.
trolled entirely from Python with only a remote kill switch for user input.

The diversity of peripherals demonstrates the ease of use of a high-level language for microcontroller development. In general, figuring out how to initialize and utilize such peripherals with a microcontroller is a long and tedious process. With Python, however, the ability to experiment within the interactive prompt often shortens the process from days to less than an hour.

In all experiments, the only native C code outside of the run-time system is the StellarsWare libraries; no other foreign functions were utilized. Specifically, no application code has been rewritten in C for performance optimization. Only the profilers discussed in Section 3.4 were used to measure and improve performance.

### 3.5.1 Microbenchmarks

A small subset of the Computer Language Shootout* shows the computational performance of the system. These benchmarks were ported to the Owl system simply by converting Python 3 code into Python 2 code and removing the command-line arguments.

*http://dada.perl.it/shootout/craps.html

ackermann is a simple, eight line implementation of Ackermann’s function that computes \( A(3, 4) \). It exercises the recursive function call and compute stack. heapsort sorts a 1000-element array of random floating point numbers. This implementation is contained in a single function call and exercises the garbage collector and list capabilities. matrix multiplies a pair of 30x30 matrices of integers using an \( O(n^3) \) algorithm. Finally, nbody is a floating-point benchmark that simulates the motion of the Sun and the four planets of the outer solar system.

*http://shootout.alioth.debian.org/
Figure 3.4: Static binary analysis, from top to bottom, of the Owl virtual machine using wrapped functions, using the foreign function interface, of the eLua virtual machine and of an example program using SafeRTOS.
3.6 Results

This section presents an analysis of the Owl system. The overhead of including the virtual machine in flash can be quite small, as low as 32 KB compared to 22 KB for a simple RTOS. We show that for the embedded workloads, garbage collection has almost no impact on run-time performance. Finally, we show that using a loader-less architecture uses four times less SRAM than a traditional system.

3.6.1 Static binary analysis

Figure 3.4 shows the output of the static binary analyzer. The four rows in the figure are the Owl run-time system using autowrapping, the Owl run-time system using eFFI, the eLua interpreter, and the SafeRTOS demonstration program. SafeRTOS is an open-source real-time operating system that is representative of the types of run-time systems in use on microcontrollers today.

Consider the Owl run-time system using autowrapping. The virtual machine section includes the interpreter and support code to create and manage Python objects. The math section includes support for software floating point and mathematical functions, while the IP section provides support for Ethernet networking. The platform section is the StellarisWare peripheral and USB driver libraries. The lib sections (C and Python) are the Python standard libraries for the Owl run-time system. Finally, the I/O sections (C and Python) are the calls to the peripheral library, wrapped by the autowrapper tool.

The Python standard libraries consume a significant fraction of the total flash memory required for the Owl run-time system. The capabilities that these libraries provide are mostly optional, and therefore can be removed to save space. However, they provide many useful and convenient functionalities beyond the basic Python bytecodes, such as string
manipulation. These sections also include optional debugging information (5 KB).

With eFFI, the binary is roughly 19 KB smaller, illustrating the advantage of using a foreign function interface. While the code required to manually create stack frames and call functions marginally increases the size of the virtual machine and stores some additional information in the Python code objects, it completely eliminates the need for C wrappers.

The Owl virtual machine itself is actually quite small, approximately 35 KB. It contains all of the code necessary for manipulating objects, interpreting bytecodes, managing threads, and calling foreign functions. This is significantly smaller than eLua’s core, which takes up 63 KB, and not much larger than the so-called “light weight” SafeRTOS, which requires 22 KB (the Source and Minimal sections). Note also that the supposed compactness of SafeRTOS is deceptive, as it is statically linked directly into the user application. Therefore, with a complex application, more libraries will need to be included and the gap to Owl, which already contains these libraries, will shrink. For instance, if the application were to use floating point math, the 20 KB math library would be linked in. Similarly, if the application were to use USB, the StellarisWare USB library would be linked in.

The size of the standard Owl distribution is on the order of 150 KB. Much of this size, however, comes from C libraries. Any C application that uses these libraries needs include them, just as Owl does. While the standard Owl distribution includes a large standard library, Owl can just as easily be compiled without unused libraries. Therefore, the space overhead of using Owl can be as low as 35 KB, the size of the interpreter itself.

3.6.2 Performance

This section presents the performance of the Owl run-time system using the profilers discussed in Section 3.4.2. Table 3.1 shows the profiling results of the benchmarks and applications described in Section 3.5. Each column (before the last column) shows one work-
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY_SUBSCR</td>
<td>1%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9%</td>
<td>11%</td>
<td>4%</td>
<td>7 µs</td>
</tr>
<tr>
<td>LOAD_ATTR</td>
<td>23%</td>
<td>10%</td>
<td>7%</td>
<td>18%</td>
<td>14%</td>
<td>18%</td>
<td>21%</td>
<td>-</td>
<td>5%</td>
<td>-</td>
<td>-</td>
<td>64 µs</td>
</tr>
<tr>
<td>LOAD_CONST</td>
<td>3%</td>
<td>1%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11%</td>
<td>5%</td>
<td>1%</td>
<td>8%</td>
<td>8 µs</td>
</tr>
<tr>
<td>LOAD_FAST</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>-</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>12%</td>
<td>18%</td>
<td>12%</td>
<td>14%</td>
<td>4 µs</td>
</tr>
<tr>
<td>LOAD_GLOBAL</td>
<td>24%</td>
<td>21%</td>
<td>41%</td>
<td>24%</td>
<td>68%</td>
<td>64%</td>
<td>59%</td>
<td>16%</td>
<td>5%</td>
<td>1%</td>
<td>-</td>
<td>75 µs</td>
</tr>
<tr>
<td>STORE_FAST</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2%</td>
<td>3%</td>
<td>6%</td>
<td>4 µs</td>
</tr>
<tr>
<td>STORE_SUBSCR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2%</td>
<td>-</td>
<td>3%</td>
<td>6 µs</td>
</tr>
<tr>
<td>BINARY_ADD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3%</td>
<td>2%</td>
<td>-</td>
<td>2%</td>
<td>10 µs</td>
</tr>
<tr>
<td>BINARY_MULTIPLY</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3%</td>
<td>16%</td>
<td>-</td>
<td>11 µs</td>
</tr>
<tr>
<td>BINARY_POWER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11%</td>
<td>-</td>
<td>116 µs</td>
</tr>
<tr>
<td>BINARY_SUBTRACT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6%</td>
<td>-</td>
<td>-</td>
<td>3%</td>
<td>10 µs</td>
</tr>
<tr>
<td>INPLACE_ADD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>-</td>
<td>10 µs</td>
</tr>
<tr>
<td>INPLACE_SUBTRACT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>-</td>
<td>11 µs</td>
</tr>
<tr>
<td>CALL_FUNCTION</td>
<td>33%</td>
<td>60%</td>
<td>44%</td>
<td>41%</td>
<td>3%</td>
<td>4%</td>
<td>5%</td>
<td>33%</td>
<td>3%</td>
<td>4%</td>
<td>-</td>
<td>148 µs</td>
</tr>
<tr>
<td>COMPARE_OP</td>
<td>4%</td>
<td>-</td>
<td>1%</td>
<td>-</td>
<td>2%</td>
<td>1%</td>
<td>1%</td>
<td>-</td>
<td>7%</td>
<td>-</td>
<td>-</td>
<td>13 µs</td>
</tr>
<tr>
<td>DUP_TOPX</td>
<td>-</td>
<td>-</td>
<td>1%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5%</td>
<td>1%</td>
<td>-</td>
<td>4 µs</td>
</tr>
<tr>
<td>FOR_ITER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5%</td>
<td>-</td>
<td>6 µs</td>
</tr>
<tr>
<td>JUMP_ABSOLUTE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>-</td>
<td>-</td>
<td>2%</td>
<td>3%</td>
<td>1%</td>
<td>7 µs</td>
</tr>
<tr>
<td>JUMP_IF_FALSE</td>
<td>-</td>
<td>1%</td>
<td>-</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>-</td>
<td>5%</td>
<td>-</td>
<td>-</td>
<td>5 µs</td>
</tr>
<tr>
<td>JUMP_IF_TRUE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5 µs</td>
</tr>
<tr>
<td>POP_BLOCK</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>-</td>
<td>2%</td>
<td>-</td>
<td>-</td>
<td>137 µs</td>
</tr>
<tr>
<td>POP_TOP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>3%</td>
<td>2%</td>
<td>-</td>
<td>-</td>
<td>2 µs</td>
</tr>
<tr>
<td>RETURN_VALUE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4%</td>
<td>1%</td>
<td>2%</td>
<td>-</td>
<td>74 µs</td>
</tr>
<tr>
<td>ROT_THREE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2%</td>
<td>-</td>
<td>3 µs</td>
</tr>
<tr>
<td>SETUP_LOOP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1%</td>
<td>-</td>
<td>-</td>
<td>15 µs</td>
</tr>
<tr>
<td>UNPACK_SEQUENCE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7%</td>
<td>-</td>
<td>12 µs</td>
</tr>
</tbody>
</table>

| (garbage collector)      | 1%                | -           | -         | -        | -             | -           | 11%               | -         | 17%      | 40%   | 3%    | 3-65 ms       |

Table 3.1: Fraction of each workload’s running time spent in each bytecode and the average execution time of each bytecode. (Bytecodes that occur few or no times are not shown.)
load. The autonomous car workload is shown using the GPS, range finder, gyroscope, or some combination thereof. Each entry shows the percentage of the run-time spent executing any given bytecode. If a bytecode is executed less than 1% of the running time of the program, it is shown as a dash. The average run-time of the bytecode is calculated as an average across all executions from all workloads and is shown on the right. For the purposes of this table, the garbage collector is treated as its own bytecode.

For most applications, the single largest contributor to running time is the CALL_FUNCTION bytecode. This bytecode is particularly complex, as it is responsible for creating call frames, instantiating objects, and calling external functions. When a program calls a foreign C function, the CALL_FUNCTION bytecode does not finish until that function completes, so the profiler attributes the foreign function’s execution time to CALL_FUNCTION.

For the embedded workloads, loading and storing values takes a large fraction of the execution time. Python stores variables in a set of dictionaries that map a variable’s name, a string, to its value. The LOADGLOBAL bytecode loads objects (including functions) from the global namespace, and is particularly slow due to the large size of this namespace.

Specifically, in the artificial horizon workload, nearly half of the running time is spent in the LOAD_ATTR and LOADGLOBAL bytecodes. Table 3.2 shows how the interpreter uses dictionaries in these two bytecodes. When the user references a global variable, the compiler loads a constant representing the string name of the variable, then calls

<table>
<thead>
<tr>
<th></th>
<th>Lookups</th>
<th>Hit Rate</th>
<th>Search len</th>
<th>Avg size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADGLOBAL</td>
<td>129232</td>
<td>0.88</td>
<td>25</td>
<td>41.2</td>
</tr>
<tr>
<td>LOAD_ATTR</td>
<td>174374</td>
<td>0.71</td>
<td>10.4</td>
<td>17.0</td>
</tr>
</tbody>
</table>

Table 3.2: Profiler results showing how dictionaries are used by the interpreter.
LOAD_GLOBAL. The interpreter searches the local module’s scope for that variable. If it is not found there, the interpreter searches the built-in namespace, which mostly contains built-in functions like \texttt{max()} or \texttt{int()}. In other words, the interpreter may have to search multiple dictionaries per name lookup. However, these dictionaries are reasonably small. As Table 3.2 shows, each lookup only needs to search an average of 25 entries to find a global variable and 10 entries to find an object attribute. Since the microcontroller has single-cycle memory access, this means that using a less space efficient, faster data structure may not be appropriate.

Owl’s garbage collector (GC) is a simple mark-and-sweep collector that occasionally stops execution for a variable period of time. This uncertainty makes Owl unsuitable for hard real-time applications. However, in practice, Owl’s GC has no significant impact on our soft real-time embedded workloads. In these applications, data structures are reasonably simple, and there are only a few small objects (around 1,500 for the artificial horizon benchmark). This means that GC runs very rarely, and only for a short period of time. For the worst case embedded workload, this is never more than 8ms, 11% of the application’s running time.

Further reducing the impact of GC on embedded workloads, our virtual machine runs the collector when the system is otherwise idle. For example, all GC invocations for the car workload occur during sleep times. In other words, the garbage collector never interrupts or slows useful work. Moreover, while Owl’s current garbage collector does not provide hard real-time guarantees, different garbage collectors exist that do [78].

Unsurprisingly, in the CPU benchmarks garbage collection can be a more significant factor. These CPU benchmarks store more complex data structures on the heap, which take a long time to traverse during the mark phase. Additionally, there are a large number of objects (over 7,500) in the heapsort benchmark that take a long time to go through in
the sweep phase. Overall, garbage collection can take up to 65 ms and up to 41\% of execution time. Similarly, the bytecodes that manipulate objects (\texttt{BINARY_ADD}, etc.) are only significant for the CPU benchmarks. The embedded workloads are dominated by the bytecodes that perform control flow (\texttt{JUMP*}, \texttt{COMPARE_OP}, etc.).

Calling I/O functions is relatively fast. A simple microbenchmark that repeatedly calls a basic peripheral I/O function, accumulating the result in a variable, illustrates the overhead of I/O. This loop was calibrated by accumulating a constant into the variable and using this time as a baseline. For functions accessed with a wrapper function, this I/O call takes 11.4 µs. The foreign function interface is more complex, increasing the call time to 20.8 µs. This time increase is significant, but it is outweighed by the savings in flash.

### 3.6.3 Memory use

Table 3.3 shows a snapshot of the contents of the heap for the artificial horizon workload. It contains roughly 1200 objects for a total of 31 KB of data, which is less than half the available space on the 9x92 microcontroller.

In general, the embedded workloads do not need to store a great deal of dynamic data on the heap. The bulk of the space used on the heap consists of references to other constants. A \texttt{segment} object is a portion of a list, and a \texttt{function} object points to a code object and the variables in its scope. In contrast, the heapsort benchmark stores over 7500 dynamic objects, most of which are integers and lists.

In the artificial horizon workload, the objects in SRAM point to 5056 objects in flash, consuming a total of 98 KB. Most of this space is used by code objects which contain bytecodes, constants, and strings. These are all immutable, so the Owl system keeps them in flash, as discussed in Section 3.2.3. However, other systems, such as p14p and eLua, would have to copy most of this data out of flash and into SRAM, increasing SRAM usage.
<table>
<thead>
<tr>
<th>Type</th>
<th>Object count</th>
<th>Avg size (bytes)</th>
<th>Total size (bytes)</th>
<th>Fraction of total heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>2</td>
<td>8.0</td>
<td>16</td>
<td>0%</td>
</tr>
<tr>
<td>int</td>
<td>180</td>
<td>12.2</td>
<td>2188</td>
<td>7%</td>
</tr>
<tr>
<td>float</td>
<td>3</td>
<td>12.0</td>
<td>36</td>
<td>0%</td>
</tr>
<tr>
<td>string</td>
<td>29</td>
<td>19.9</td>
<td>576</td>
<td>2%</td>
</tr>
<tr>
<td>bool</td>
<td>2</td>
<td>12.0</td>
<td>24</td>
<td>0%</td>
</tr>
<tr>
<td>tuple</td>
<td>47</td>
<td>15.2</td>
<td>716</td>
<td>2%</td>
</tr>
<tr>
<td>packed tuple</td>
<td>5</td>
<td>8.0</td>
<td>40</td>
<td>0%</td>
</tr>
<tr>
<td>set</td>
<td>1</td>
<td>12.0</td>
<td>12</td>
<td>0%</td>
</tr>
<tr>
<td>seglist</td>
<td>92</td>
<td>16.5</td>
<td>1520</td>
<td>5%</td>
</tr>
<tr>
<td>segment</td>
<td>251</td>
<td>40.0</td>
<td>10048</td>
<td>32%</td>
</tr>
<tr>
<td>list</td>
<td>7</td>
<td>12.0</td>
<td>84</td>
<td>0%</td>
</tr>
<tr>
<td>dict</td>
<td>200</td>
<td>16.3</td>
<td>3252</td>
<td>10%</td>
</tr>
<tr>
<td>xrange</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>module</td>
<td>22</td>
<td>36.4</td>
<td>800</td>
<td>3%</td>
</tr>
<tr>
<td>class</td>
<td>14</td>
<td>12.0</td>
<td>168</td>
<td>1%</td>
</tr>
<tr>
<td>function</td>
<td>317</td>
<td>36.1</td>
<td>11440</td>
<td>36%</td>
</tr>
<tr>
<td>instance</td>
<td>2</td>
<td>12.0</td>
<td>24</td>
<td>0%</td>
</tr>
<tr>
<td>code obj</td>
<td>2</td>
<td>44.0</td>
<td>88</td>
<td>0%</td>
</tr>
<tr>
<td>packed obj</td>
<td>1</td>
<td>12.0</td>
<td>12</td>
<td>0%</td>
</tr>
<tr>
<td>thread</td>
<td>1</td>
<td>36.0</td>
<td>36</td>
<td>0%</td>
</tr>
<tr>
<td>method</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>frame</td>
<td>5</td>
<td>94.4</td>
<td>472</td>
<td>1%</td>
</tr>
<tr>
<td>block</td>
<td>2</td>
<td>20.0</td>
<td>40</td>
<td>0%</td>
</tr>
<tr>
<td>(all)</td>
<td>1185</td>
<td>26.7</td>
<td>31592</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 3.3: A snapshot of the heap, broken down by object type, for the artificialhorizon workload.
by over a factor of four. This is a critical advantage of design of the Owl toolchain. Program complexity is limited by flash, not by the much more scarce SRAM.

### 3.7 Conclusions

This chapter provides a concrete demonstration that not only are mid-range microcontrollers capable of running a complete run-time system for a modern programming language but that such a system is not much larger than a traditional real-time operating system.

By implementing and investigating a full-scale embedded run-time system, we can observe several key insights about such systems in general. First, a large fraction of the system as a whole consists of support libraries that would also need to be included in a native C executable. This means that Owl is not appreciably larger than a similar C application that also uses large libraries. Second, the run-time characteristics of embedded applications are very different from traditional computational workloads. For instance, garbage collector and math performance have much less of an impact on the types of programs that are likely to be run on a microcontroller than they do on data intensive workloads. Instead, the execution speed is limited by efficient variable lookup and function calls. Finally, an embedded control program often uses many more constants than dynamic objects. By keeping these constants in flash, the overall dynamic memory footprint in SRAM of a complex embedded application can be kept relatively small. A traditional dynamic loader, as used by CPython and the Java JVM should be avoided on such systems.
CHAPTER 4

The Medusa Concurrent Programming System

4.1 Introduction

Chapter 3 shows that it is possible to raise the level of abstraction when programming an embedded system. Instead of manually managing memory, the run-time system can manage memory on behalf of the programmer. Instead of debugging a complex peripheral initialization routine, the he or she can experiment with the peripheral at a prompt. Instead of programming in C, the programmer can use a language like Python that is inherently simpler and safer. These changes made it significantly easier to develop the example applications discussed earlier. However, they do not directly address the core complication of embedded systems: they are all concurrent systems.

Microcontrollers are fundamentally designed to be a part of an event-driven system. They are connected to sensors and actuators and operate in response to external stimuli. Given that microcontrollers are used to control physical systems—such as microwave ovens, cars, and industrial machinery—embedded software must be robust and reliable. The challenge of such systems stems from the fact that the real world that the microcontroller is supposed to integrate with is a massively parallel system. Events can happen at any time, in any order.

Unfortunately, massively parallel systems lead to concurrency and synchronization is-
sues that are notoriously difficult to manage, even in large scale systems with system software support [19, 79, 80]. The Python language itself provides little support for such systems, nor do either the desktop or embedded implementations.

Owl, along with other embedded real-time operating systems (RTOS) provide primitive, low-level mechanisms for thread scheduling, synchronization, and communication [4, 5, 6, 7]. Threads run largely independently and exchange data by mutating shared state. These systems do not directly address the challenges of event-driven systems. Programmers are still responsible for allocating resources to tasks, arbitrating access to peripherals, and synchronizing access to shared data. These responsibilities are difficult and even expert systems make errors doing so. As an example, consider the autonomous car introduced in Section 3.5. An outline of this code is shown in Figure 4.1. Logically separated tasks like steering and obstacle avoidance are mixed together into the same code. In line sixteen, the program reads from the range-finder. If this read takes time, the read from the range-finder on line seven will also be delayed. Therefore, the programmer had to write a significant amount of manual scheduling code: lines 6, 13, 15, 23, etc. This code is difficult to write and tune.

Medusa is a programming language and run-time system that addresses these issues head-on. Medusa is based on the actor model of concurrency [54, 55]. The actor model solves the fragmented control flow and shared state problems inherent in traditional approaches to event-driven programming [19]. The Medusa programming model and run-time system utilize and expand upon these ideas for small embedded systems.

This chapter presents a new actor-based programming language, Medusa. It has a built-in message passing system is based on Erlang, which has been rigorously evaluated, both formally and in practice. This combination makes Medusa simple enough to be used by a novice and yet expressive enough to build complex concurrent applications.
for location in self.route:
    ...
    atTarget = False
while not atTarget:
    ...
    if curtime > range_update_time:
        # Read range finder;
        # get distance to nearest obstacle
        if dist_to_obstacle < RANGE_MAX:
            # Set motor proportional and servo
            # inversely proportional to distance
            ...
            range_update_time = curtime + RANGE_PERIOD
        ...
    if curtime > loc_update_time:
        # Read GPS; get current location, 
        # heading, and distance to destination
        if dist_to_goal < ERROR_MARGIN:
            # Set motor to min speed
            atTarget = True
        else:
            # Set motor proportional to dist_to_goal
            loc_update_time = curtime + LOC_PERIOD
    ...
    if curtime > heading_update_time:
        # Calculate degrees to turn (deg_to_turn)
        heading_update_time = curtime + HEADING_PERIOD
    ...
    if curtime > gyro_update_time:
        # Read gyro; update steering and integral
        if dist_to_obstacle == RANGE_MAX:
            # Set steering to gyro’s recommendation
            gyro_update_time = curtime + GYRO_PERIOD
        ...
        # Stop the car when it reaches the final location

Figure 4.1: Python event loop for autonomous RC car.
Secondly, we present an implementation of the Medusa system as a set of extensions to Owl. These extensions are very small, less than 3KB of compiled code. This allows Medusa to run on systems that have less than 1% of the minimum memory requirements of Scala or Erlang. An empty thread only consumes 130 bytes, and there is no fixed limit to the number of threads in the system. It is also fast: The time needed to prepare a message, send it from one thread to another and finally process it on the other end is less than the time required for five function calls. The scheduler can both spawn a new thread and perform a context switch in less than the time required for a single function call. Even on a microcontroller with 96 KB of RAM, Medusa can support hundreds of threads. Finally, they are backwards compatible. Existing Python code runs unmodified under Medusa. We describe this implementation in enough detail to be generalizable to any bytecode virtual machine. We evaluate the complete system with microbenchmarks and realistic embedded applications, demonstrating that modern language features can be used on systems smaller than any that have been previously demonstrated—or even proposed. By using Owl and Python as the basis of Medusa, it is possible to directly compare embedded systems built in both environments. This quantifies the costs and benefits to directly supporting concurrency in

Section 4.2 describes the Medusa language itself. Sections 4.3 describe the implementation of the messaging and toolchain systems of Medusa in detail. Section 4.5 presents a quantitative analysis along with real-life applications built with Medusa.

4.2 Medusa language

The Medusa language is a backwards-compatible, extended version of Python that directly supports the actor model. This is analogous to Scala, a similarly extended version of Java. It includes several key features derived from Erlang: light-weight threads, messaging, pattern
matching, and atoms.

While Scala and Erlang themselves already support the actor model, they require too many resources for small embedded systems. The Squawk Java run-time system can run on more capable embedded systems, but there is no evidence that it is light-weight enough to run efficiently on a microcontroller with less than 100 KB of RAM. In contrast, Owl is a light-weight Python run-time that has been demonstrated to be capable of running large programs on microcontrollers. Therefore, it was a natural starting point for Medusa.

Note, however, that using the Medusa language is entirely optional in the Medusa system. The key features of the Medusa system, lightweight threading and messaging, are available to Python programmers. However, there is significant benefit—in terms of scalability, reliability, and maintainability—to eschewing mutable state from Python to build strictly actor-based functional programs. Medusa’s new syntax is described formally in Figure 4.2 and described in detail in the rest of this section.

4.2.1 The Medusa language

In Medusa, actors are implemented as light-weight threads. A context switch takes roughly the same amount of time as the execution of a Medusa function call. The interface for creating a new thread in Medusa is extremely simple, using thread.spawn. Once the new thread is running, it is also simple to send messages to the spawned thread. Any immutable object can be sent as a message. For example:

```python
new_thread = thread.spawn(function)
new_thread.send(1)
new_thread.send((1, "foo", 2.1, True, None))
new_thread.send((1, "foo", 2.1, (1, 2, 3)))
```

A message can contain any combination of integers, floating-point numbers, booleans, None, strings, and tuples. Furthermore, atoms, which will be discussed shortly, can also
bind_op = expr, '<-', expr;
atom = ''' , ( all characters - '''), ''';
recv_stmt = 'recv', ':', case_suite;
switch_stmt = 'switch' expr ':' case_suite;

case_suite = NEWLINE, INDENT,
  case_stmt, (case_stmt), DEDENT;
case_stmt = 'case', expr ':' suite;

Figure 4.2 : The new syntax of the Medusa language in EBNF.
be part or all of a message. Medusa provides a library for converting thread objects to and from thread IDs, examining the pending messages to a thread and determining the current thread ID.

Once the actor has been started, it receives a message using the `recv` statement, shown in Figure 4.4. The interpreter matches against each `case` block sequentially. If the message matches the first case, the message will be consumed, the print statement will be executed and the `recv` block will finish. Note that code does not “fall through” into other case blocks. If the message fails to match any block, the message will be deferred for later handling, and the system will wait for a new message.

```
recv:
  case 1:
    print "received 1"
  case 2:
    print "received 2"
```

Often, the programmer will not know the exact message an actor is expected to receive. Medusa allows this through pattern matching, where a portion of the message is specified and other parts are stored as variables:

```
recv:
  case ("fire!", 1, temperature):
    print "engine one on fire!"
  case ("fire!", 2, temperature):
    print "engine two on fire!"
```

A pattern can contain immutable values plus any number of variable names. If a variable is already bound, the message will only match if value in the message is the same as the value bound to the variable.

When a message is received, each element in the message is compared against the first pattern. For example, assume the above actor is sent the message ("fire!", 1,
The first two elements of the message match the pattern. The system then assigns the values from the message to the unbound variable names. In this case `temperature` is assigned the value `1205`. If the pattern does not match all of the bound elements of the message, all unbound variables remain unbound. Literal values, bound variables and unbound variables can appear in any order in a pattern. Additionally, patterns and messages can be arbitrarily complex; patterns are matched with a deep comparison. Finally, the keyword `Any` acts as a wildcard and can be used in a pattern to match any value. This is useful when a part of a message does not need to be saved.

Alternatively, consider the case where the block is sent the message `("fire!", "apu", 1, 1205)`. In this case, the engine type portion of the message ("apu") does not match the bound variable `kind`. As such, the pattern match fails, no value is assigned to `temperature`, and execution continues with the next case.

In addition to `recv` blocks, Medusa allows pattern matching with the new match operator (`<-`):

```
data = ("baz", 42)
("baz", number) <- data
```

In this example, the pattern `("baz", number)` matches against `data`. The value `42` is stored to the variable `number`. If the pattern matching had failed, however, an exception would have been thrown, stopping the execution of this thread with an error.

In addition to complex patterns, as shown above, the match operator can be used with the trivial pattern: `a <- 32`, pronounced “a gets thirty-two”. This enables single assignment, guaranteeing that a variable will never change value, providing support for functional programming.

In the examples so far, strings have been used to distinguish message types. Statically typed languages, such as Scala, use actual object types instead. Dynamically typed lan-
```python
import thread

def start():
    EchoThd <- thread.spawn(echo)
    EchoThd.send((mytid(), "hello"))

    recv:
        case (EchoThd.tid(), Msg):
            print Msg
        EchoThd.send('stop')

def echo():
    recv:
        case (FromTid, Msg):
            FromThd <- thread.get_thd(FromTid)
            FromThd.send(mytid(), Msg)
        return echo()
        case 'stop':
            print "Stopping", mytid()
```

Figure 4.3: A short example of Medusa syntax. The main thread sends a message to a child thread which then echos it back.

guages, such as Erlang, use “atoms”. Internally, atoms are handled more efficiently than strings, as they are merely textual representations of a unique identifier. Medusa supports atoms using the backtick delimiter:

```medusa
    case ('fire', 'apu-one', temperature):
```

Finally, Figure 4.3 shows an example that combines all of these elements. A main thread spawns an actor that echoes messages back to the sender. The main thread then sends a message to the echo server, waits for the proper response, then stops the server with the 'stop' atom. After receiving a message, the echo actor recursively calls itself to handle the next message. The Medusa compiler automatically optimizes this tail-call so no stack space is used.
4.3 Implementation

One of the key elements of the Medusa system is its message passing capabilities. Threads only communicate via messages, and messages only contain immutable data. Therefore, threads cannot corrupt each other’s state. Further, the message passing infrastructure avoids head-of-line blocking by allowing selective receipt of messages. Messages are received using pattern matching, allowing out-of-order delivery of the messages that the thread wants to receive.

In this model, threads wait for, receive, and respond to messages. These messages are software events created by other threads in the system. This model is ideally suited to event-driven systems. Since the programming model matches the structure of such event-driven applications, it simplifies the design, implementation, and maintenance of concurrent software.

The two key mechanisms of the Medusa system that enable efficient, light-weight messaging are pattern matching and mailboxes. This section describes these mechanisms in detail and briefly discusses messaging from Python contexts.

These mechanisms are implemented as a set of extensions to Owl. Specifically, it is built using a small number of new bytecodes, object types, and a modified Python compiler. These extensions are completely backwards compatible. Existing code runs unmodified. Python and Medusa code can even run at the same time on the same device. They consume 3KB of compiled code, less than 10% of the overall size of our virtual machine.

4.3.1 Pattern Matching

Pattern matching is integral to the Medusa messaging system. It allows threads to concisely specify the messages that they are ready to receive. Pattern matching is implemented by
combining Python’s existing comparison support with a new “unbound variable” object.

For example, consider this pattern match operation:

\[(12, a, b) \leftarrow (12, 3, 4)\]

The Medusa compiler compiles this almost exactly as if it were a standard comparison:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Bytecode</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOAD_CONST</td>
<td>(12)</td>
</tr>
<tr>
<td>3</td>
<td>LOAD_NAME_UNBOUND</td>
<td>(a)</td>
</tr>
<tr>
<td>6</td>
<td>LOAD_NAME_UNBOUND</td>
<td>(b)</td>
</tr>
<tr>
<td>9</td>
<td>BUILD_TUPLE</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>LOAD_CONST</td>
<td>(12)</td>
</tr>
<tr>
<td>15</td>
<td>LOAD_CONST</td>
<td>(3)</td>
</tr>
<tr>
<td>18</td>
<td>LOAD_CONST</td>
<td>(4)</td>
</tr>
<tr>
<td>21</td>
<td>BUILD_TUPLE</td>
<td>(3)</td>
</tr>
<tr>
<td>24</td>
<td>COMPARE_OP</td>
<td>(\leftarrow)</td>
</tr>
</tbody>
</table>

For variable loads on the left hand side of the pattern match, the compiler emits `LOAD_NAME_UNBOUND` instead of `LOAD_NAME`. When the program executes `LOAD_NAME_UNBOUND` on the name `a`, it looks up the variable `a` to see if it is bound. If it is, it loads its value and pushes it onto the stack, exactly as `LOAD_NAME` would do. If it is not, instead of raising a name error exception, it creates a new `UnboundLocal` object as a placeholder for the future value for the variable named `a`. Similarly, the variable `b` is looked up, and either its value or a new `UnboundLocal` is pushed on the stack. For the sake of this example, assume that `a` was previously bound to 3 and `b` is unbound.

When execution reaches `COMPARE_OP`, the virtual machine will compare the top two objects on the stack:

\[(12, 3, [UnboundLocal for name "b"]\))
(12, 3, 4)

`COMPARE_OP` for the bind operator starts by creating an empty dictionary to store un-
bound objects with their new values. Then, it performs a nested comparison on the two tuples just as it would in standard Python.

It starts with the first element, compares it, then moves on. Finally, with the last element, it compares the literal value 4 with the unbound object [UnboundLocal for name "b"]. This comparison always succeeds, since \( b \) does not yet have a value. The virtual machine adds an entry to the dictionary associating the name of the unbound local with the value it was compared with. If the value is not going to be used, a special Any object can be used, instead of an unbound variable, to match anything. If every element in the comparison matches, this means that the pattern has matched. At this point, each of the entries in the unbound objects dictionary can be committed into actual variables.

A receive statement consists of a sequence of these patterns. As a message arrives, an attempted match is made against each pattern in the receive statement. If a match fails, the next pattern is tried. If a match succeeds, the variables are bound and the code block associated with that pattern is executed.

An unbound variable will match any Medusa object of any type. Therefore, while not dictated by the messaging system, the first element of a message is often an atom. This effectively allows the message to contain information about itself, given that Medusa, like Python, has no static type system that would allow the programmer to express the type of data that should be matched. This initial atom can be used to identify the expected format of the rest of the message and can be used to ensure that patterns only match data of the expected type.

4.3.2 Mailboxes

In Medusa, each thread has a pair of queues to support message passing: one for incoming messages (the mailbox queue) and one for deferred messages (the deferred queue). When
recv:
  case 1:
    print "received 1"
  case 2:
    print "received 2"

Figure 4.4: Basic Receive Statement.

a thread sends a message, it appends a reference to the message object in the destination thread’s mailbox. Messages cannot contain mutable objects, so they do not need to be copied. When a thread executes the `recv` statement, the virtual machine first moves any messages from the deferred queue into the front of the mailbox, using the `UNDEFER_MSG` bytecode. Then, it removes the first message from the mailbox, using the `RECV_MSG` bytecode, and attempts to match it against each pattern, in order.

If none of the patterns match, the message is appended to the end of the deferred queue, using the `DEFER_MSG` bytecode. The next message is removed from the mailbox, and the process repeats. Finally, if the mailbox is empty (either because the thread had no pending messages or because all the pending messages were deferred), the thread blocks, waiting for another message. This allows the scheduler to execute other threads.

The key to this process is that when messages from the deferred queue are moved back into the mailbox, they remain in the order of their arrival. If a given message cannot be handled by an actor in its current state, a later message may reconfigure the actor to be able to handle the older message. It can then be handled and removed from the mailbox. This prevents deadlock.

An example of this process is shown as Medusa source code in Figure 4.4 and as compiled bytecode in Figure 4.5. The first two bytecodes move all the messages that might have previously been deferred back into the mailbox queue (at the front, and in the order that
Figure 4.5: Compiled bytecodes for the program in Figure 4.4. The program receives one message, checks it against one pattern, then compares it against a second if the first pattern fails to match.
they arrived) and then receives the first message from the mailbox and places it on the stack. The next block of code (bytecode offsets 2–10) performs the first pattern match against the pattern “1”. The message is duplicated, the constant 1 is placed on the stack, and they are swapped (because the message must be on the top of the stack). The COMPARE_OP bytecode then actually does the matching and places True or False on the stack depending on whether the match was successful or not.

If the first match fails, the POP_JUMP_IF_FALSE bytecode skips to the next pattern match (bytecode offsets 21–29), which is nearly identical in this case. If the match succeeded, the pop jump bytecode will simply pop True off the stack and fall through. The associated code (bytecode offsets 13–18) will execute, printing the string “received 1” and jumping to the end of the block (bytecode offset 44) which simply pops the original message off the stack.

If both matches fail, the message will be deferred (bytecode offset 40) and the code will jump back to the receive (bytecode offset 1). If there is another message, it will try to match that message again. If not, the RECV_MSG bytecode will block waiting for the next message to arrive and the thread will yield the processor.

Notice that the bulk of the work here is done by the compiler with only a few specialized bytecodes. This provides significant flexibility to use these mechanisms and ample opportunity for compiler optimization for special cases.

### 4.3.3 Example

Figure 4.6 shows a simple example of how messaging works. In this example, two monitor threads are spawned which simply wait for some event to occur. These events could be anything and are not relevant to the example. When the monitor threads detect an event, they send a message back to the main control thread. Here, the messages are simple strings,
def monitor1(controller):
    # wait for event 1 to occur
    controller.send(me().tid(), "alert!")
    monitor1(controller)

def monitor2(controller):
    # wait for event 2 to occur
    controller.send(me().tid(), "alert!")
    monitor2(controller)

def init():
    # spawn two monitors to send me alerts
    m1 <- thread.spawn(monitor1, me())
    m2 <- thread.spawn(monitor2, me())

    # process events
    event_loop(m1.tid(), m2.tid())

def event_loop(mltid, m2tid):
    recv:
        case (mltid, msg):
            # process alert from m1
        case (m2tid, msg):
            # process alert from m2
        case Any:
            print "Unexpected message!"

    # Always return to the event loop
    event_loop(mltid, m2tid)

Figure 4.6: A simple event loop written in Medusa that uses pattern matching to determine which event was received.
but they could be arbitrary data. The main control thread runs an event loop waiting for messages from the monitor threads.

When the main control thread receives a message, pattern matching is used to determine what to do. The incoming message is first matched against the pattern \((\text{m1tid}, \text{msg})\). In this tuple, the first variable, \text{m1tid}, is already bound, whereas the second, \text{msg}, is not. So, this pattern will match any incoming message that is a tuple of two elements with the thread ID of the m1 thread. If this match fails, the next pattern, \((\text{m2tid}, \text{msg})\), will be tried. If either of these matches succeed, the \text{msg} variable will be bound to the data from the second element of the tuple in the message. The last case with the pattern \text{Any} will match any other message, ensuring the mailbox does not fill with unexpected messages. If information about the unexpected message is desired, the pattern could be a single unbound variable which will match any object (including a tuple). This structure frees the programmer from worrying about the order of message arrival. The loop will process messages as they arrive, and there is a clear and easy way in which to specify how to process each message. Finally, \text{event_loop} is reinvoked to process the next message. The Medusa compiler optimizes this tail call.

While this example shows the elegance of the messaging system, mailboxes are designed to provide further control over message receipt to the programmer. Imagine, instead, that it is only useful to process messages from the second monitor after receiving a message from the first monitor. The out-of-order delivery mechanism of the mailbox makes this possible. Consider the revised event loop in Figure 4.7. In this case, no matter what message arrives first, the main control thread will wait for a message from the first monitor. All other messages, either from the second monitor or unexpected messages, will be deferred.

Once a message from the first monitor is received, it will be processed. The next \text{recv}
def event_loop(m1tid, m2tid):
    recv:
        case (m1tid, msg):
            # process alert from m1
    recv:
        case (m2tid, msg):
            # process alert from m2
        case Any:
            print "Unexpected message!"
    # Always return to the event loop
    event_loop(m1tid, m2tid)

Figure 4.7: By using more than one recv block, a thread may handle one type of high-priority messages regardless of low-priority messages that may be received before it.

block will then be executed. In that case, the first thing that happens is that all received messages that were previously deferred waiting for a message from the first monitor will be “undeferred”. So, if a message from the second monitor had arrived first, it will now be received and processed immediately.

These simple examples demonstrates how the messaging implementation enables flexibility for the programmer. Further, note that the mailbox maintains messages in the order that they arrived. The programmer can reorder these messages only by the structure of recv blocks and patterns.

4.3.4 Deadlock and finite message bounds

The actor model in general is resilient to deadlock. Traditional notions of deadlock, where two threads wait on one another to free locks, are impossible in Medusa since there are no locks. However, it is possible for the system to fail if a large number of messages of one type are sent to a thread that is only receiving a message of a different type. This is true of
any actor system with finite message queues [81]. In Medusa, the system will stop with an out-of-memory exception.

The Medusa environment has tools to help diagnose and fix these potential issues. The virtual machine keeps track of the largest number of pending messages that are ever held at once in each thread’s mailbox. Additionally, it tracks the average number of pending messages in the mailbox each time the thread receives a message. The programmer can view these values for any threads at any time. If either value is particularly large, there is the potential for mailbox overflow. In that case, the application should probably be restructured.

### 4.3.5 Messaging and Python

Using the new Medusa syntax is optional in the Medusa system. Actors can be programmed in traditional Python, with mutable state (although only immutable objects may be sent in messages). Python and Medusa threads can coexist in the same program, communicating with each other and running at the same time.

They send messages and spawn threads using the `send(msg)` and `thread.spawn` calls described earlier. The `recv()` call returns the oldest message in that thread’s mailbox. If no message is available, `recv()` blocks, descheduling the thread until a message is available. In this way, the `recv()` Python function call has similar semantics to the Medusa `recv` statement. However, since Python lacks pattern matching, the programmer must manually interpret the contents of the message.

### 4.4 Medusa Infrastructure

For the Medusa system to work, there is some additional necessary infrastructure, including a light-weight threading system and a compiler.
The threading system facilitates the actor model and makes it possible to create, destroy, and schedule large numbers of threads on a small embedded system. The compiler utilizes the Medusa hardware mechanisms to implement communication and bridging, as described in the previous sections.

4.4.1 Medusa Threads

In Medusa, thread state is all stored within the heap. A thread consists of a thread id, message queues, the thread’s activation records, and the thread state (active, blocked, etc.). There is no stack in the system, so all activation records are allocated on the heap and contain a “back” pointer to the previous record. This architecture allows for extremely lightweight thread creation, deletion, and scheduling. If there is space on the heap, a new thread can quickly be allocated and its initial activation record is setup based on the function that is passed to the spawn call. The thread is then immediately ready to go. On average, it takes about 59\(\mu\)s to create a thread on a 50 MHz Cortex-M3 processor. It also makes context switches quickly: about 129\(\mu\)s on average to context switch while running 100 threads.

As all state is contained within the heap, context switch overhead is incredibly low. To swap threads, the interpreter simply needs to switch its reference from one thread context to the other, and then it can immediately start executing bytecodes from the new thread. No other bookkeeping or state management is needed (other than to place the old thread back onto the run queue).

A thread object itself is minimally 176 bytes. As messages are queued for the thread, the message queues will grow to contain references to each message. As messages are not copied, only the 4 byte references need to be stored within the thread object’s queues. The thread object must also refer to a linked-list of activation records. The size of each record
is dependent on the program, but is at least 40 bytes.

### 4.4.2 Task registry

Erlang provides a convenient facility for associating a thread with a name. Once registered, other threads can send the registered thread a message through simple syntax. Medusa provides a similar facility for associating a particular thread with an atom. The programmer can then send messages to the registered thread by calling the `send` method on that atom:

```py
new_thread = thread.spawn(function)
thread.register(new_thread, 'task')
'task'.send('message')
```

This is useful in single processor systems, but it becomes critical in the distributed systems that will be discussed in Chapter 6.

### 4.4.3 Medusa Compiler

The Medusa compiler is based on the Python compiler that is included as part of the standard library of Python 2.7. This served as a relatively easy-to-modify compiler that generates standard Python bytecode. Many bugs were fixed and the C-based parser was replaced with the Python-based parser from the PyPy project*. This compiler was tested to show that it properly compiles large Python code, including the large applications discussed in the previous chapter.

The compiler was modified to treat atoms as first-class data types for the language. They are parsed, converted into constants and loaded into memory images exactly as integers, strings, floating-point numbers and tuples are. Pattern matching was implemented as a special case of the comparison operator while unbound locals are generated by the

*http://pypy.org/
virtual machine. The unbound locals described earlier are generated automatically by the virtual machine, all with very little modification required to the compiler. Finally, support was added for the new `recv`, `switch` and `case` statements. All together, the modified compiler consists of 9,900 lines of Python code.

The compiler and virtual machine were also modified to support a basic form of tail call optimization. After bytecodes have been assembled, the compiler detects locations in the output code where a child function is called and the result immediately returned from the parent function. For these child function calls, instead of emitting the `CALL_FUNCTION` bytecode, it emits the new bytecode `TAIL_CALL_FUNCTION`. This is executed by the virtual machine identically to the `CALL_FUNCTION` bytecode, except that the back pointer from the grandparent’s activation record to the parent’s activation record is replaced with a reference to the child’s activation record. When the child returns, it will return its result directly to the grandparent. Meanwhile, the parent’s activation record can be garbage collected. This allows tail recursion of arbitrary depth in a bounded amount of memory.

Since the virtual machine does not reuse activation records, it can optimize both recursive and non-recursive calls. This is important in the actor model since individual functions represent the state an actor can be in. If an actor repeatedly transitions between two states, a traditional tail recursion optimization would eventually overflow. This cannot happen in Medusa.

4.5 Evaluation

To evaluate the Medusa system, we constructed and measured the performance of several microbenchmarks and several embedded applications. This chapter presents those results that measure the performance of the messaging subsystem. The next chapter combines messaging and I/O and presents more complete results for realistic applications. These
4.5.1 Messaging performance

To measure the speed of message passing, two simple benchmarks were constructed, shown in Figure 4.8. Both spawn a configurable number of worker nodes that receive a message, then forward it on to a specified node. In the broadcast/collect benchmark, a broadcaster node transmits one message to each worker nodes. They then forward their message to a single collector node. In the chain benchmark, the head node transmits only to the first worker node. That node forwards it on to the second worker node and so on. Finally, the last worker node forwards the message to the collector node.

For both benchmarks, the time between the first and last message sent is timed. These results are shown in Figure 4.9, normalized to the number of messages sent in each benchmark. At light load, the time required to prepare and send a message, switch contexts to the recipient and finally receive the message is around 600 microseconds for both bench-
marks. This is less than the time required for five function calls. As memory pressure increases with more threads, this time increases to around 1000 microseconds. The additional threads create more garbage to be collected, slowing overall progress through the program. The broadcast/collect benchmark is more complex in implementation, so it runs into memory pressure somewhat earlier.

Figure 4.9: Average time to send and receive a message for different benchmarks.
Microcontrollers are primarily used to control physical systems. The controller monitors a system using a suite of sensors and changes the system with a suite of actuators. This can be structured naturally as an event-driven system. When there are no inputs, the system need not do anything. When there is an input, the system responds to that input appropriately. One type of input could be a timer, to allow the system to perform periodic tasks even when there are no external events. Traditionally, there are two mechanisms that can be used to detect events: polling and interrupts. Both of these mechanisms have limitations. Polling is conceptually simple, but it is computationally expensive and admits the potential to miss events. Interrupts use fewer CPU cycles but is very tricky to implement properly.

Using the same messaging system introduced in Chapter 4, we present a third option: message bridging. As events are detected by the microcontroller, they are automatically converted to Medusa messages that can be detected by software threads. These messages are indistinguishable from software messages and can be handled with all the same pattern matching infrastructure described in Chapter 4. This mechanism is fast; accepting, converting and storing a external events takes less than 4 microseconds. Moreover, it is extremely simple to use. Programmers simply wait for messages in separate threads and the system handles all synchronization automatically.

This chapter discusses bridges in detail. Sections 5.1 and 5.2 discuss polling and inter-
rupts, the status quo of embedded I/O. Section 5.3 introduces bridges and discusses their implementation. Section 5.4 presents quantitative analysis of our implementation of bridges.

5.1 Polling

Polling can be used to detect events. Every so often, the controller looks at, or “polls”, external inputs of interest. If the external state has changed, an event is processed. Consider a very simplified model of a microwave oven controller. The code to monitor the door could be written as follows:

```python
def door_handler(newstate):
    if newstate == OPEN:
        # turn oven off
        # turn light on

    else if newstate == CLOSED:
        # turn light off

def event_loop(door, door_handler):
    doorstate = door.read_switch()

    while True:
        olddoorstate = doorstate
        doorstate = door.read_switch()

        if doorstate != olddoorstate:
            door_handler(doorstate)

    ...
```

There is only one event that can happen here: the door changes state. In a real implementation, there would be additional if statements checking additional events, and these checks may be throttled by only performing them if a minimum time has passed. In this simplified example, each time through the loop, the controller checks the state of the door. If it has changed, then the `door_handler` is called to respond to the state change. This is
the basic approach taken by programmers of earlier embedded systems that run high-level, procedural programming languages. While it is logically simple, even this example has some basic flaws:

1. Event handlers can delay each other. Once the event loop has called a handler, there is no way for a higher priority event (such as opening the microwave door) to preempt it.

2. It is possible to miss events. While opening and closing the door are likely to take long enough that a change in state will always be detected, that may not be the case for inputs that change state quickly. Such inputs must be polled frequently enough that they will be detected. Hardware buffering of events mitigates this problem, but does not eliminate it.

3. Constantly polling when events do not occur is wasteful of processor resources and power. With this approach, every input must be polled periodically, regardless of how often real state changes occur.

The event handlers could be run in separate threads. This could possibly mitigate the first two problems to some extent. If the thread in charge of the door handler is given high priority, it could be scheduled immediately whenever the polling thread detects a door state change. The polling thread would also need to be given high priority in order to detect such changes quickly. The polling itself could also be separated into threads depending on how often each input needs to be polled. This cannot guarantee neither that event handlers will not be delayed nor that events will not be missed. With careful scheduling, however, the probability of problems can be reduced.

However, the multiple threads will need to coordinate through shared state. Since the program thread may be reading or writing from those shared locations, the interrupt and
program threads must communicate using thread-safe techniques. These are difficult and error-prone to use, which can lead to catastrophic synchronization bugs. These bugs are unpredictable, only occurring under very specific situations, and can often be virtually impossible to reproduce. This is well known to be challenging on traditional computer systems [79, 80] and is particularly difficult on embedded systems [19].

The third problem is in direct conflict with the second. To reduce the cost of polling, one must poll less frequently. To reduce the probability of missing an event, one must poll more frequently. Even if the events are buffered in hardware, polling less frequently will increase response latency. An appropriate balance may be impossible to find.

5.2 Interrupts

As a compromise, the microcontroller includes hardware to monitor inputs for changes. This hardware runs even when the core is busy with other work or sleeping. When a change is detected, the hardware interrupts or wakes the core and calls an interrupt service routine (ISR). When that routine returns, the core either continues its original work or returns to sleep. Returning to the simple microwave controller, the door switch could cause an interrupt whenever it’s state changes, which would then invoke (probably indirectly) the door_handler code.

While interrupts can save time and energy, they are still problematic. Systems written in C use hand-coded interrupt service routines (ISRs). In effect, this creates a multi-threaded program: one interrupt handler thread and one program thread. The interrupt handler thread must send data back to the program thread so that the program can respond to the event. This must be done by writing to and from shared locations in memory. Again, this is difficult and error-prone.

Further complicating the task of hand-writing ISRs is the fact that they must execute
Figure 5.1: Pushbuttons have a tendency to “bounce”, or oscillate between outputs when their state changes. (http://www.protostack.com/blog/2010/03/debouncing-a-switch/)

quickly. If a second interrupt event happens when the ISR is already running, the ARM Cortex-M3 interrupt controller sets a hardware flag to mark that an interrupt is pending. When the first ISR returns, the hardware immediately re-executes the ISR to handle the new event. However, if a third event had happened, the new event would be lost. In effect, only a single event can be buffered while the ISR is running. This makes interrupts problematic for use with high level languages on embedded systems. It is difficult, if not impossible, to use any run-time management within an ISR.

An example makes this challenge more clear. Consider detecting when a button attached to a microcontroller is pressed. Conceptually, this should be simple. A microcontroller’s general purpose I/O (GPIO) can read the digital value of a pin connected through the switch to a voltage source. However, physical realities mean that most pushbuttons “bounce”, oscillating between open and closed several times whenever they are pushed (Figure 5.1). This requires “debouncing” either through hardware or software. In software,
code will detect only the steady state of the button, ignoring bouncing.

Using polling, this operation is relatively simple (Figure 5.2). However, as discussed in the previous section, this implementation is problematic. Specifically, while the button is bouncing and the processor is waiting in the call to sys.sleep, no other thread can make progress.

Using interrupts would solve this problem, however the implementation is almost comically complex. This is because the timer and the input need to be synchronized. Appendix A shows an implementation of a non-blocking, interrupt-based debouncing algorithm using the SafeRTOS C environment. It is 141 lines long and uses two semaphores. There are nearly a limitless number of easy to make and difficult to debug errors that are possible in this code.

Interrupt handling is so notoriously difficult on microcontrollers that at least one microcontroller, the Parallax Propeller, was designed without any form of interrupting at all. The Propeller has eight cores sharing a public RAM array. To wait for an external event, each core can halt until a pin transitions. This means that each event a user wants to monitor requires an entire core to be reserved for it [82].

5.3 Bridges

Medusa introduces a new technique for managing I/O on embedded systems called bridging. It avoids shared state between interrupt handlers and the main program exactly as Medusa solves this problems between multiple software threads: message passing. In effect, bridging makes the hardware agents that monitor for events into actors themselves. These actors run all the time, do not interfere with threads running on the core and send messages when events occur instead of modifying shared state.

Bridges replace the manual process of sharing data with an ISR with a standard, thread-
# debounce.py
# wait-free debounce algorithm in Python

import simplegpio, time

BUTTON = simplegpio.Input('B4', simplegpio.PULLUP)

def get():
    # wait for the button to be pressed
    while BUTTON.get() == False:
        pass

    # wait for the button to stop bouncing
    time.sleep(.2)

    # return the state to the programmer
    return BUTTON.get()

Figure 5.2: Blocking debounce in Python. This program is conceptually simple, but the time spent in line 14 while the program is sleeping blocks other threads from making progress.
safe interface. With bridging, ISRs are extremely simple and all communication happens through a single function call. Further, the programmer cannot introduce a synchronization bug or race condition. Messages from bridges are indistinguishable from other messages and can be received through the `recv` statement, like any other message.

Additionally, the bridge interface used by ISRs is extremely fast. It does not allocate memory off the heap, so it never has to run the garbage collector. The ISR to deliver general-purpose I/O (GPIO) events through a bridge deterministically completes in 187 cycles on our system, less than 4 microseconds. This minimizes the possibility that events will be lost. Overall, bridges eliminate the two biggest challenges of one of the hardest parts of embedded programs: dealing with interrupts.

5.3.1 Implementation

At its core, a bridge is a producer-consumer ring supporting safe asynchronous communication between two endpoints. In between the endpoints, the virtual machine converts Python or Medusa types to and from C types. With an inbound ring, an ISR calls the bridge code to produce one or more bytes of data. Later, the virtual machine automatically consumes data from this ring and delivers the data to a subscriber. Alternatively, data in an outbound ring is produced by a user thread, then consumed by C code and an interrupt routine.

Each peripheral that will need to either send or receive data from a Medusa thread will have a bridge, denoted by a bridge number, assigned at VM compile time. If a peripheral needs bidirectional communication, it will be programmed with two bridges, one for each direction. The user then initializes these bridges at run-time. For each bridge, the user specifies how many bytes each message will contain and how many messages should the ring should hold. For example, consider a bridge to deliver GPIO messages to a Medusa
thread. Each event can be described in five bytes, four to represent the 32-bit port number and one to represent the status of all eight bits on any given port. Therefore, the bridge is set up to send five byte messages:

```python
b <- bridge.create(GPIO, # bridge number
            32, # number of messages
            5) # size of each entry
```

The programmer then sets up the underlying hardware to trigger interrupts on GPIO level transition. Then, the user sets a tag to be included with every message. In this case, the tag is the atom `gpio` which can be matched by a pattern. Finally, the user specifies the subscriber to the bridge, which will receive any data sent to it. In this case, the subscriber is the current thread:

```python
# init the hardware interrupts
interrupt.IntEnable(interrupt.INT_GPIOD)
gpio.GPIOIntTypeSet(button.port,
            button.pin,
            gpio.GPIO_BOTH_EDGES)
gpio.GPIOPinIntEnable(button.port,
            button.pin)

# set the tag and subscriber
b.setTag('gpio')
b.subscribe(me())
```

The GPIO interrupt handler (Figure 5.3) constructs these five byte messages and passes them on to the virtual machine for handling. First, it clears the pending interrupt, then reads the GPIO port. It concatenates these values into a single block of size `MSG_SIZE`, then calls `bridge.produce`. This function takes the bridge number, a pointer to the message and the message size.

First, the virtual machine checks that the user has actually initialized the bridge corresponding to the bridge number passed to `bridge.produce`. Then, it checks that the
```c
#define ALL_PINS 0xff
#define MSG_SIZE sizeof(unsigned long) + 1

void GPIOInterruptHandler(unsigned long port)
{
    uint8_t values;
    uint8_t message[MSG_SIZE];

    /* clear all the interrupts for this port */
    GPIOPinIntClear(port, ALL_PINS);

    /* read the value of the port */
    values = GPIOPinRead(port, ALL_PINS);

    /* pack the data into a five byte message */
    memcpy(message, &port, sizeof(unsigned long));
    message[sizeof(unsigned long)] = values;

    /* send it to the subscribers */
    bridge_produce(GPIO_BRIDGE, &message, MSG_SIZE);
}
```

Figure 5.3: The GPIO interrupt service routine. When the microcontroller detects a change of the input on a particular GPIO port, this routine is called. It reads the new value of the input into a port, then sends that value to the virtual machine.
provided message size matches the message size selected by the user at bridge initialization time. Finally, if the bridge ring is full, it returns an error. In this case, the error is ignored since there is no way to reject a GPIO event. Unlike a serial port, buttons and switches don’t have flow control.

On the VM side, the system checks that the bridge is initialized and is not full. It then copies the message into the ring. At this point, the interrupt handler returns and the processor resumes executing whatever bytecode it was interpreting when the interrupt occurred.

When that bytecode completes, the virtual machine will consume any new messages out of the bridges by copying them into a Python/Medusa string. The tag specified by the user is combined with the string into a tuple, and that tuple is sent to the subscribing thread. The thread scheduler then marks the subscriber as runnable, and bytecode execution continues. It is critical to realize that this portion of the bridge code is not running inside an ISR, so does not delay or interfere with the execution of other interrupts.

Once the bridge is set up, this process is entirely transparent to the subscriber. That thread simply receives messages, tagged with the specified tag, and acts upon them, just as it would had it received a message from another thread.

This is a very simple implementation for GPIO events, but it could be extended to include more information about the system state when the GPIO event occurred. For example, the bridge could include the exact time in the message. To do this, the message size is increased and the value of a timer is copied into it. The rest of the interrupt handler is unchanged.
5.3.2 Chronograph

A central timer, or “chronograph”, can also be implemented using bridging. A thread can request that a message be sent to it at some later time. The chronograph then determines which thread will get the next timer wakeup, configures a hardware timer, and waits to receive the message from it. The scheduler automatically suspends the chronograph thread and any threads waiting on a message from it. When the hardware timer goes off, the scheduler wakes up the chronograph, which in turn wakes up and sends a message to the subscribing thread.

The chronograph also provides a `sleep(time)` function that stops the current thread for `time` milliseconds. Internally, this function uses the same hardware timer and bridge as the rest of the chronograph. Using this facility, a thread that is sleeping is automatically suspended until it is time to wake up.

5.3.3 Debouncing in Medusa

Medusa makes the implementation of the non-blocking debouncing algorithm shown in Appendix A dramatically simpler (Figure 5.4). The debounce function listens for events from either the timer or the input. When it detects that the button has changed state, it starts the timer and records the new state of the button. From there, it waits for the button to bounce again or for the timer to expire.

The behavior of the Medusa program is largely similar to that of the C program, but the implementation is much more clear. To the programmer, it is effectively a blocking implementation. The debounce function just waits for new messages while the virtual machine continues running other threads. When an event happens, it is automatically converted to a new thread and the debounce algorithm resumes. *It is less than one eighth the length of the*
C program and just only twice as long as the blocking, Python implementation. It requires no locking and it is impossible to introduce a bug that would corrupt data or block other threads from progressing.

5.4 Evaluation

This section evaluates the complete Medusa system using software and hardware messages. It presents both microbenchmarks and several realistic embedded applications, demonstrating how Medusa works in the real world. We compare applications that use the Medusa systems of messaging and bridging with procedural, polling-based applications that run on the standard Owl system.

5.4.1 I/O Latency

The simplest I/O benchmark for Medusa monitors an input pin for a signal and raises an output pin in response. The baseline for this test uses polling. It spins in a loop, reading the input line, writing its state to the output line and yielding back to the scheduler. The other version of this program uses bridges and interrupts. When the pin changes state, an interrupt triggers the bridge interrupt handler described in Section 5.3.1. It sends a message to a thread, which then raises the output pin. This experiment was run using both standard and priority bridges. These use a modified version of the scheduler that executes a blocked thread immediately when it receives a message from a bridge. To compete for time, the microcontroller runs a heapsort benchmark repeatedly in a background thread. I/O latency was measured using another Stellaris microcontroller and confirmed with a factory-calibrated Tektronix TDS 220 digital oscilloscope. Both benchmarks were tested 500 times each. The distribution of these response times are plotted in Figure 5.5.

In the polling implementation, the thread reading the input must wait until it is sched-
```python
# debounce.md
# wait-free debounce algorithm in Medusa

import simplegpio, chrono, thread, registry

BUTTON = simplegpio.Input('B4', simplegpio.PULLUP)
BUTTON.enable_interrupts()

def debounce(current_state, bouncing):
    recv:
        case ('gpio-change', port_state):
            new_state = bool(port_state & (BUTTON.pin))

            if not bouncing:
                # this replaces our existing timer.
                chrono.one_shot(10, 'steady')

            return debounce(new_state, True)

        case 'steady':
            'main'.send(('button', current_state))
            return debounce(current_state, False)

def main():
    recv:
        case ('button', True):
            print "down"

        case ('button', False):
            print "up"

thread.spawn(debounce, (False, False))
thread.register('main', main)

Figure 5.4: Non-blocking debounce in Medusa.
Figure 5.5: Distribution of I/O latency for polling and bridges.
uled before it can detect a change. The probability that the GPIO will happen across the 10 ms that the background thread is running is uniform. However, approximately 25% of the time, the background thread triggers the garbage collector, which can last upwards of 85 ms. Once the garbage collector has finished, the polling thread will run and can respond to the input.

The behavior is similar with the standard bridge benchmark. The bridge delivers a message to the I/O thread, which will receive the I/O message once it is scheduled. Again, the garbage collector may interrupt the background thread, slowing response. Note, however, that the bridge implementation will not lose events unless its buffer fills. The bridge interrupt handler will still run during garbage collection, queueing messages. Compare this to polling, which can miss short events during garbage collection. This effect becomes more dramatic as more threads are active in the system competing for processor resources. The best performance comes with the priority bridge. Here, the I/O thread does not have to wait for the background thread to complete its timeslice, so latency is less than 2 ms 80% of the time.

5.4.2 Streaming data

Many peripherals such as GPS receivers and ultrasonic rangefinders send periodic updates as bursts of data over a serial port. In a GPS unit, update messages are sent once per second in messages ranging from twenty to eighty bytes in size. This presents a problem for applications that use polling to monitor events. While the hardware serial port (UART) on the microcontroller does have a buffer, it too small to hold a complete message. If a program is not polling when a message is sent, the hardware buffer will overflow and the user will not receive the complete message. If another thread is running during the message transmission, the polling thread may miss the entire transmission. If this happens, only the
16 bytes at the end of the transmission, the data in the hardware buffer, will be received. Additionally, the user program must be able to pull bytes out of the UART buffer at least as fast as they are being received.

Figure 5.6 shows this behavior. A transmitter sends 50 byte bursts to a receiver. The receiver records the bytes either using a polling loop or with bridges. The first experiment (solid line) shows that even when the controller is doing nothing but polling the UART, the software cannot keep up with the transmission rate after 9600 bits/sec. When the polling loop has to compete for time with a background task, it loses data after 4800 bits/sec. Eventually, both of these experiments are so much slower than the transmission rate that the only data that can be recovered are the 16 bytes in the UART hardware buffer.

Compare this to an application that receives bytes using bridges and interrupt handlers. In this case, whenever traffic is received, the interrupt handler places the data in a bridge for the receiving thread. Then, that thread is activated by the scheduler and can pull the data from its message queue. In this case, the background task is never interrupted when data is not being sent, and the application can receive the complete 50 byte message at any tested transmission rate.

Similar results are seen with real-life applications. We constructed a simple example that receives and parses data from a GPS receiver. An application using polling could receive and interpret messages up to a transmission rate of 9600 baud. When the GPS transmitted any faster, all messages received are incomplete, and therefore unusable. When the polling thread runs in competition with another thread, no complete messages were received at any data rate. However, with bridges, the GPS can run at its full speed (57600 baud) even with a background thread. Since the GPS thread only runs when it is actively processing a message, the background thread ran at 93% of its native speed.
Figure 5.6: Data loss rates for different transmission rates.
5.4.3 Event-driven Systems

The previous sections have shown the efficacy of the Medusa mechanisms for communication and concurrency. The ultimate motivation for these mechanisms is to build better structured event-driven systems. To demonstrate that these mechanisms are both practical and usable in such systems, we built several embedded applications, including an autonomous car and a traffic-light controller.

Autonomous car

The autonomous car demonstrates that Medusa can coordinate concurrent tasks and peripherals into a cohesive embedded system. The electronics from an off-the-shelf RC car (an Exceed RC Electric SunFire Off-Road Buggy) were replaced with a 9B92 microcontroller and associated peripherals. The car is controlled entirely by the microcontroller. An ultrasonic range finder, GPS receiver, and three-axis gyroscope connected to the microcontroller transmit feedback from the car’s surroundings, while connections to the car’s motor and steering servo provide control of the car’s movements.

Figure 5.7 shows the design of the car application. Each box in the figure represents an actor (Medusa thread). Each oval represents an actor state (Medusa function). Dotted arrows represent state transitions within an actor. Solid arrows represent messages being sent from one actor to another.

The GPS connects to the microcontroller over a UART. Every second, the GPS sends a packet of data that contains control information and its current position. After each byte of the packet is sent, the microcontroller triggers an interrupt. This signal is caught by the specific UART’s bridge interrupt controller, converting the byte from the GPS into a software message. A GPS driver thread subscribes to these messages being sent from the
Figure 5.7: Diagram of Event-driven Autonomous Car.
GPS via the UART bridge. When it has received an entire message, it converts the message from a string of bytes into a Medusa message with numeric longitude and latitude. These messages are sent to a navigation controller. This thread maintains a list of waypoints and sends turn commands to the master thread described below.

The motor controller and rangefinder also connect to the microcontroller via UARTs, using a similar combination of bridges and driver threads. The gyroscope connects over the I2C bus, also through a bridge to its own driver module. Finally, the application uses the chronograph to wait for an exact period of time without spinning to read the clock.

The car is controlled by a master thread that collects messages from the GPS, rangefinder and gyro and sends messages to the servo and motor controller threads. This master module runs a two-term, PI feedback controller that makes sure the car drives straight, even over varying terrain (or the author’s foot). Meanwhile, it receives turn commands from the navigation thread. Finally, it monitors the rangefinder to avoid hitting obstacles.

Note that the nine threads in this system have no shared state whatsoever. All communication is done through messages. Furthermore, all communication from the hardware takes place through bridges. This means that none of the peripherals are polled. Each driver thread waits to receive a message from its interrupt bridge and is descheduled when there is no data to process. When new data is available, it is rescheduled and execution continues.

During execution, this application sends an average of 315 messages per second, 243 of which come from interrupt bridges and 72 that come from other software threads. The system is idle 52.4% of the time, i.e. all threads are waiting on data from external sources. This allows all messages to be dealt with promptly. On average, there are less than 1.1 messages queued whenever a thread receives a message. The maximum number of messages queued in a thread’s mailbox at one time was 40, in the thread that receives bytes from
the GPS receiver. These 40 bytes correspond to a single GPS sentence which was likely received while the VM was busy with an uninterruptable task like garbage collection. As soon as that task finished, the GPS driver thread resumed and read the entire message.

This example was originally written as a single-threaded, event-loop based program in Python. While conceptually simple, the concurrent nature of the peripherals proved to be very difficult. The event-loop has to run very slowly and be tuned very carefully to ensure that the control loop has consistently updated data and that input events are not missed. Adding a feature becomes harder as the program grows more complex. Suppose the programmer wants to trigger a periodic event. On program start, a global variable is set storing the next time the event needs to happen. Each time through the control loop, the clock is polled and compared against that global variable. If it is time, the programmer executes the event and resets the global variable. Each feature like this slows down the event loop, degrading performance of everything else in the system. Moreover, if anything else in the event loop takes a long time, the periodic event will be triggered late. Writing the program in C using interrupts and locks would be even more difficult and extremely error-prone due to the large number of events coming from both hardware and software components that need to be synchronized.

The Medusa program is comparatively simple. A periodic task can be implemented by calling the chronograph’s sleep function (see Section 5.3.2), performing the task, and repeating:

```python
def periodic():
    chrono.sleep(5000)
    do_task()
    return periodic()

thread.spawn(periodic)
```

Since the chronograph uses hardware timers that do not need to be polled, this task *does*
not impact others while it is waiting to run. The many components in a large program just wait for data to be available, process it, and send it on to other components. The bridge, messaging and scheduling systems synchronize everything automatically.

**Traffic light controller**

An undergraduate research assistant constructed a simple traffic light controller as another demonstration of a complex event-driven system in Medusa. This system is intended less as a model of a real traffic intersection, and more as a motivating example that puts together all of the concepts discussed in this chapter.

The bridge interface receives interrupts from the GPIOs connected to three buttons, as described in Section 5.3. These interrupts are converted to messages to three other actors in charge of these buttons. There is a “change” button, which requests an immediate change in direction of the lights and two “emergency” buttons simulating the remote control of lights by emergency vehicles. A “light controller” actor responds to the four input events and sends the appropriate messages to the individual actors in charge of the actual lights.

The overall state machine for this system is very complex and there are many different events that can occur across the system. Expressing this as an event-driven system with the actor model is relatively straight-forward. Writing this same program for an embedded system in C would be much more difficult and error prone due to the enormous amount of synchronization required. The result would be a single complex state machine with interrupt handlers and locks to ensure correctness. Even using Python would not help that much, as it would be likely that many input events would be missed if the program were not crafted very carefully.

Other demonstration applications have also been built using Medusa such as a distributed traffic-light controller that runs on a microcontroller and a multi-threaded web
Figure 5.8 : Diagram of Event-driven Traffic light controller.
server that runs on a port of Medusa to a standard x86 Linux system. These examples show that the Medusa system is flexible and easily programmable.

5.5 Conclusions

Building reliable concurrent systems has long been a challenging endeavor. Higher level programming languages like Python can significantly improve programmer productivity by automatically managing resources and raising the level of abstraction, but they do little to improve the safety and reliability of concurrent systems. Programmers still need to worry about synchronizing access to shared state, leading to difficult-to-find non-deterministic bugs.

Fortunately, the programming languages and computer systems communities have developed techniques that solve these issues in larger computer systems. Functional, actor-based programming is used everywhere from telephone switches to the Facebook messaging platform. Medusa brings these advances to small systems for the first time. It introduces a new, highly accessible, actor-based programming language based on Python. Actors can be implemented either as pure functional modules or using mutable state, techniques familiar to any Python programmer. Second, it shows that these modern programming language concepts can be used on platforms that are much smaller than previously thought: platforms where safety and reliability are critically important. Finally, it introduces a novel bridging system that makes it much simpler and safer to process events at the time that they happen, combining the performance of interrupts with the programming simplicity of blocking I/O.
CHAPTER 6

The Hoot Distributed Programming System

This chapter presents Hoot, a proof-of-concept system that demonstrates that message passing is a practical and easy-to-use model for building distributed embedded systems. It allows messages to be sent between separate microcontrollers just as they are sent between different threads. By doing this, the programmer can treat a collection of microcontrollers as a single parallel system. The fact that threads are running on separate hardware is largely irrelevant. Hoot is built on top of the facilities already available in Medusa and Owl, requiring virtually no modification to the virtual machine. In fact, Hoot is written entirely in Python and Medusa. It can run on both microcontrollers and traditional desktop computers. While this means that Hoot is relatively slow, it demonstrates the power of the Actor model in distributed, as well as concurrent, systems.

So far, this thesis has considered embedded applications that run on a single processor in a single virtual machine. While programming individual controllers is certainly important, virtually all embedded systems are comprised of several processors in a distributed system. A modern car is a clear example of this. A large number of processors communicate over a Controller Area Network (CAN), each responsible for a variety of tasks such as entertainment, safety, or engine management. Each controller has its own set of peripherals, handling some tasks autonomously and coordinating on others.

Other embedded systems are distributed in less obvious ways. For example, a robotic
vacuum may only have one primary processor, but that processor may communicate over ad-hoc protocols to application-specific integrated circuits (ASICs) like motor controllers or radios or sensors. While these ICs are not directly programmable, they are internally still computers. They can be anything from simple 8-bit processors to the same 32-bit Cortex-M controllers that run Owl. Often, the program running on the primary processor will write to configuration registers on the peripheral processor over a serial bus such as SPI or RS-232. The program running on the peripheral processor will then read from and write to these registers as its runs its own control loop.

Unfortunately, this arrangement has exactly the same concurrency challenges discussed in Chapters 4 and 6. Specifically, the programmer of the primary processor must carefully synchronize access to different registers on the peripheral processor to ensure correct behavior. The fact that the programmer cannot generally change or even see the program on the peripheral processor makes this even more difficult that it otherwise would be. At best, the programmer will have to test each write to ensure success. At worst, the programmer will simply have to wait between operations and hope that writes succeed.

Fortunately, the same solution discussed in the previous two chapters is applicable to this problem as well: message passing. In fact, Erlang, the programming environment that inspired Medusa, does exactly this [83]. Messages can be exchanged not only between software threads on the same system, but also between threads running on different systems.

Section 6.1 discusses the design of the Hoot system. Section 6.2 discusses tools built around Hoot, including a remote programming facility. Finally, Section 6.3 compares Hoot to other messaging and serialization formats including JSON and Google Protocol Buffers.
6.1 Design and Implementation

Hoot is a relatively simple system since it relies on the technologies discussed in the previous three chapters. Instead of using a new encoding format, messages are composed into packed tuples that double as both a container and a serialization format. They are sent to the communications layer as a Medusa message. The object in memory is then copied out of memory, byte for byte, and sent over a network.

On the other end, the message is sent from the network interface into the communications layer via a bridge. It then either forwards it on through another network or stores it directly on the Owl heap. Finally, the object is sent to a Medusa thread (Figure 6.1).

Since this communications layer is primarily written in Python, it can run on both microcontrollers and traditional computer systems (Figure 6.2). Large portions of Hoot share the same source on both platforms, shown in double boxes in Figures 6.2 and 6.1. This allows a Python program, running in CPython 2.7, to send a message to a Python or Medusa application running on a microcontroller. It can also take advantage of the error handling and exception control built into Owl. Hoot relies on this for all transmission and link failures. If a message is undeliverable, dropped, or corrupted, Hoot throws an exception. It makes no attempt to redeliver the message or restart the link.

More critically, another student is currently researching techniques for detecting and recovering from failures in embedded systems.

6.1.1 Destination objects

Embedded systems use a wide variety of very different communications methods with very different characteristics. Routed networks are used in Ethernet networks. Broadcast networks are used in CAN and I²C. Point-to-point channels are used in UART (RS-232) and
Figure 6.1: Data flow in Hoot/Medusa.
SPI. Some of these networks support automatic routing, some require manual routing, and some have no routing or addressing capability at all.

Hoot does not enforce any particular approach here: it runs on top of existing networks and uses their supported configurations. If it is built on top of a point-to-point system, messages have no address headers. If it is built on top of a bus or routed topology, messages will use the headers already included in that network’s packet layout.

To send a message, the user program must first request a destination object from the hoot module. This module is written in Python and is available from Python programs on the desktop and microcontroller as well as Medusa programs. This request selects the network and any parameters necessary to specify a particular destination. For an Ethernet network, this is the MAC address. On a TCP/IP network, it is the IP address. Point-to-point
networks only have one destination, the device on the other side of the link. They have no parameters. To a programmer, these objects look just like a Medusa thread object: they have a send method and can be registered as tasks (Section 4.4.2):

```python
rangefinder <- hoot.destination("uart")
programmer <- hoot.destination("usb")
navigation_controller <- \n    hoot.destination("ether:00 00 00 00 00 01")

navigation_controller.send('test-message')
thread.register(navigation_controller, 'nav')
```

Many times, a single embedded system will have a mixture of different networks (Figure 6.3). To use such a system, the user requests a destination object for an entire “path”. The user specifies a network and parameter for each network along the path from one controller to the next, finally ending up at the destination:

```python
# from "source" to "dest"
controller <- hoot.destination( \n   "ether:00 00 00 00 00 01,can:1")

# from "dest" to "source"
controller <- hoot.destination(\n   "can:3,ether:00 00 00 00 02")
```

Internal to each virtual machine, Hoot simulates two networks: the debugging console and the task module. Any message sent via the debugging console will be printed as output on that controller. The task module takes as its parameter the name of a Medusa task, routing all messages sent to it to that task:

```python
programmer_console <- hoot.destination("usb:debug")
controller_task = hoot.destination( \n    "ether:00 00 00 00 00 01,can:1,task:main")
```
Figure 6.3: An embedded system with several different networks.
6.1.2 Message switch

When Hoot generates a path to a destination, it generates and stores an encoded form of the path. The encoded path starts with one byte specifying the network type for that hop. If the path requires any parameters, they will be stored directly after the network byte. This repeats for every network along the path. For example, the path specified as "ether:00 00 00 00 00 01,can:1,task:main" is encoded as:

| E | 00 | 00 | 00 | 00 | 00 | 01 | C | 01 | T | m | a | i | n | 00 |

When a user sends a message via a destination object, Hoot copies the object out of memory and concatenates it with the complete path. It then sends that string to the message switch:

| E | 00 | 00 | 00 | 00 | 00 | 01 | C | 01 | T | m | a | i | n | 00 | (bytes of object)...

The switch removes the first byte from the string (in this case, the specifier of the first network in the path) and delivers the rest of the string to the transport module designated to handle messages on that network.

6.1.3 Transport modules

When the message arrives at a transport module, it starts by splitting off the encoded parameter stored at the start of the string of bytes. For example, the Ethernet transport removes six bytes, one for each byte in the destination MAC address.

It then transmits the bytes leftover over the link using the parameter. For a byte-oriented link like TCP/IP or a serial port, it must delimit the message so that the receiver can tell when one message ends and another begins. For a packet-oriented link like Ethernet, the module is responsible for dividing the message up into packet-sized units for transmission and reassembling it at the other end.
When the message is read and reassembled at the other end of the network, it is sent to that controller’s message switch where it is processed for the next hop. Again, the first byte specifies the transport, 'C' for CAN, and the next byte specifies the destination, address 01.

'C'  01  'T'  'm'  'a'  'i'  'n'  00  (bytes of object)...

The task module and debugging console are special forms of transport modules. Since they are always at the end of a path, they do not hand the decoded message back to the message switch. Instead, they read their parameter (in this case, the null-terminated name of the task) and load the rest of the message into memory. They allocate a chunk off the heap and copy the contents of the message straight into the chunk. The only necessary processing is to preserve metadata in the object header necessary to maintain the integrity of the heap.

'T'  'm'  'a'  'i'  'n'  0  (bytes of object)...

Since CPython uses a different object structure in memory, this process differs on the desktop. It reads objects by passing them to the heap inspector described in Section 3.4. This turns the objects back into Python objects, where they can be used by a Python program.

### 6.2 Applications

The complete Hoot stack is functional and easy-to-use. An example of a complete Hoot program is shown in Figure 6.4 and 6.5. It creates a message on the desktop, transmits it to a microcontroller which then sends a response back to the desktop. It is only a few lines long and took the author less than ten minutes to write and test. This section also discusses two much more complex applications, a distributed version of the autonomous model car and an interactive programming facility.
# -------------------------------------
# echo.py
from hoot import Hoot, Atom

h = Hoot()

# build a route to the microcontroller
destination = h.destination('usb,task:demo')

# pack the message
out_obj = (Atom('ping'),)

# send it
destination.send(out_obj)

# wait for the response
print(h.msgs.recv())

Figure 6.4: A Hoot program, written for Python. It communicates with a microcontroller running the program in Figure 6.5.

Unfortunately, the complete Hoot stack is also quite slow. Round-trip time through the entire stack, from Python to Medusa over USB and back again, takes on average 30 milliseconds for even a small packet. Some packets can take over three times this long (Figure 6.6). This is orders of magnitude slower than the underlying transport. However, the design of Hoot is fundamentally sound and there is significant opportunity for optimization.

6.2.1 Autonomous car

The principal result of this chapter is to demonstrate the simplicity of building distributed embedded systems in a message-passing environment. To demonstrate this, the autonomous model car project was modified to run under Hoot. The navigation module,
# echo.md

```python
import hoot
import hoot_usb
import hoot_tasks
import thread

# register us as the "demo" task
thread.register(me(), 'demo')

# set up the route back to the desktop
desktop <- hoot.destination('usb,task:main')
thread.register(desktop, 'desktop')

def loop():
    recv:
        case ('ping',):
            'desktop'.send('pong')

    return loop()

loop()
```

Figure 6.5: A Hoot program, written for Medusa. It communicates with a desktop computer running the program in Figure 6.4.
Figure 6.6: Data flow in Hoot/Medusa.
and its interface to the GPS, is split off onto a second microcontroller. The rest of the
application runs on the first microcontroller, sending and receiving message to the naviga-
tion microcontroller over Ethernet. In this example, no higher level networking stack, like
TCP/IP, is used. Messages are sent as raw Ethernet frames.

Since logically separate modules already run in separate threads and communicate via
messages, separating the code onto separate microcontrollers requires no modification of
control loops or program logic. The only thing that needs to be changed is initialization
logic to set up destinations and start threads. These changes consist of 41 lines of code out
of a total of 893 lines for the entire program and are shown in their entirety in Figure 6.7.

6.2.2 Interactive programming

One of the most popular features of the original Owl project is the interactive development
facility. Users have repeatedly said that it is one of their favorite features. However, users
often complain about inability to stop a statement once it has been issued. Fixing this has
been very difficult due to the fact that the interactive prompt is written in a single thread.
The microcontroller receives a code object, runs it, then starts waiting for a new code object.
When a user’s statement is actually running, the virtual machine cannot receive data from
the programmer.

Medusa and Hoot makes fixing this trivial. The existing code was modified, both on the
desktop and the controller, into a multi-threaded program where programs and responses
are sent as messages. This was a surprisingly simple task. The code on the microcontroller,
now written in Medusa, is only 90 lines of code. The modified version of the desktop code
actually shrunk significantly, from 514 lines of Python to 308 lines.
# in car.md (microcontroller #1)

```python
import nav, hoot, hoot_ether, hoot_tasks

REMOTE_NAV = True
NAV_REMOTE_PATH = 'ether:000000000001,task:nav'

def wait_for_running():
    recv:
        case 'running':
            return

def start(use_nav=True):
    if REMOTE_NAV:
        remote <- hoot.destination(NAV_REMOTE_PATH)
        thread.register(remote, 'nav')
        'nav'.send('start')
        wait_for_running()
    else:
        thread.register(thread.spawn(nav.start), 'nav')
```

# in car.md (microcontroller #2)

```python
import hoot, hoot_ether, hoot_tasks

REMOTE_NAV = True
LOOP_REMOTE_PATH = 'ether:000000000002,task:nav'

def wait_for_start():
    recv:
        case 'start':
            return

def start():
    if remote_nav:
        remote <- hoot.destination(LOOP_REMOTE_PATH)
        thread.register(remote, 'loop')
        wait_for_start()
        'loop'.send('nav-running')

    return run(WAYPOINTS)
```

Figure 6.7: Modifications to the autonomous car project to run in a distributed manner.
6.3 Results

The packed tuple facility inside of Owl was originally written to allow objects to be stored in flash by the programmer in the form that they would be used in. This eliminates the need for a run-time code loader that copies constant objects from read-only Flash to writable SRAM. It saves considerable memory at run-time (Section 3.6). In Hoot, this still saves memory. While the SRAM requirements are not reduced since messages are not stored in Flash, it does save size in the virtual machine itself. The code to serialize and deserialize messages must already included in the virtual machine. The cost of such code should not be underestimated. Serialization formats can be quite complex, requiring complex code to read and write them.

To quantify this, two serialization libraries were ported to the ARM Cortex-M3, open-source C implementations of JSON * and Google Protocol Buffers †. They were compiled with the same toolchain used to build Owl, with the same optimization options.

The Protocol Buffer library consumes 18 KB of Flash and the JSON library consumes 12 KB of Flash. While this could likely be reduced through careful optimization, all of the code required to send and receive Hoot messages is already part of the Owl virtual machine. Therefore, the code size overhead of including support for message encoding in Hoot is zero.

Since Hoot’s message format was not principally designed to be transmitted over a network, it has some inefficiencies. Specifically, objects must be padded to a multiple of 32-bits to fit in Owl’s heap. It also cannot include any built-in compression. Despite this, Hoot messages are comparable in size to dedicated serialization formats.

---

*http://sourceforge.net/projects/cjson/
†https://github.com/protobuf-c/protobuf-c
The size of several messages encoded in different formats is shown in Table 6.1. For this table, messages were encoded using the Python Standard Library and their size counted in bytes. Atoms are not included in this comparison since they are only supported by Medusa. Protocol Buffers use a somewhat different design philosophy than these other formats, so they are not included. Messages are not self-contained representations of the data object encoded. Any given message can only be interpreted within the context of a particular message template defined in a .proto file. In this manner, they are much more similar to a C structure where the data in memory is meaningless without the structure definition to define which data is stored at a particular location.

Medusa stores all integers as 32-bit numbers, which makes it inefficient for coding small numbers compared to text encodings which use a variable number of digits per number. In extreme cases such as the number 1, JSON can encode a message simply as the string "1". Hoot must include an entire 32-bit integer and an entire 32-bit object descriptor, eight times as much data. However, text encoding becomes inefficient for large numbers since each byte can only encode a single decimal digit.

6.4 Discussion

There is a great deal of benefit to treating a complex, intertwined embedded system as a single distributed system. With the correct abstraction, it is much easier for both human programmers and automated tools to reason about such a system. It is possible to produce off-line analysis tools to prove facts about correctness, robustness and error tolerance if those tools can examine the entire system for error. Programmers can move functionality around between controllers, confident that they can see the entire functionality of the complete system in a single, consistent set of source code. The mass of ad-hoc communications protocols between heterogeneous systems spoils this. Even a correctly written
Table 6.1: Sizes in bytes for messages compared in several serialization formats, including Medusa packed tuples.

<table>
<thead>
<tr>
<th>Object</th>
<th>Medusa</th>
<th>Pickle</th>
<th>JSON</th>
<th>BSON</th>
<th>XML-RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bare_string'</td>
<td>20</td>
<td>19</td>
<td>13</td>
<td>26</td>
<td>80</td>
</tr>
<tr>
<td>('none_obj', None)</td>
<td>28</td>
<td>22</td>
<td>18</td>
<td>34</td>
<td>143</td>
</tr>
<tr>
<td>('tuple',)</td>
<td>20</td>
<td>18</td>
<td>9</td>
<td>28</td>
<td>119</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>14</td>
<td>64</td>
</tr>
<tr>
<td>-1</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>14</td>
<td>65</td>
</tr>
<tr>
<td>1.2</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>18</td>
<td>72</td>
</tr>
<tr>
<td>2000000000</td>
<td>8</td>
<td>13</td>
<td>10</td>
<td>14</td>
<td>73</td>
</tr>
<tr>
<td>8.23456789e+23</td>
<td>8</td>
<td>17</td>
<td>14</td>
<td>18</td>
<td>83</td>
</tr>
<tr>
<td>('float', 1.2)</td>
<td>28</td>
<td>23</td>
<td>14</td>
<td>39</td>
<td>155</td>
</tr>
<tr>
<td>('int', 1)</td>
<td>28</td>
<td>19</td>
<td>10</td>
<td>33</td>
<td>145</td>
</tr>
<tr>
<td>('both', 1.2, 1)</td>
<td>36</td>
<td>25</td>
<td>16</td>
<td>45</td>
<td>182</td>
</tr>
<tr>
<td>('long', 1, 2, 3, 4, 5)</td>
<td>60</td>
<td>32</td>
<td>23</td>
<td>62</td>
<td>258</td>
</tr>
<tr>
<td>('nested', (2.2, (1, False, 'foo')))</td>
<td>72</td>
<td>51</td>
<td>36</td>
<td>78</td>
<td>346</td>
</tr>
<tr>
<td>(the numbers from 0 to 99)</td>
<td>808</td>
<td>396</td>
<td>390</td>
<td>805</td>
<td>2971</td>
</tr>
</tbody>
</table>

program on one microcontroller may encounter a race condition when that program is distributed across several controllers. With a poorly designed protocol, it will be difficult or impossible to reason about the system as a whole.

This has resulted in security exploits on everything from cars [10, 11, 12, 13] to tire pressure monitoring systems [14] to hotel and office door locks [15]. Hoot, by itself, does not completely solve this problem. What it does do is provide a framework for existing analysis techniques to be adapted to existing embedded networks. A programmer can write a distributed program as if it were a concurrent program in Medusa. Since destination objects have the same interface as thread objects, the same code can be reused. Modules can be combined or split apart at will without changing the functional behavior of the system. It is much easier to read and understand such a program, both for humans and tools.
One of the central design decisions in Hoot is that it re-uses the data structure of objects in memory for message encoding. This makes message transfer fast and eliminates the need to include expensive ([84, 85]) serialization and deserialization code on the microcontroller. However, it restricts communication to virtual machines that use compatible object formats. Unfortunately, some microcontrollers are simply too small to run Owl. For example, the Atmel ATtiny13 only has 64 bytes of memory. This falls well short of our memory requirements and is almost certainly too small to run a memory allocator, much less any kind of managed run-time.

This chapter presents MiniMedusa, a library that allows very small microcontrollers, devices with only a few kilobytes of memory, to send and receive Medusa messages without a complete virtual machine. MiniMedusa serves to demonstrate that a single message format can span the entire gambit of computers used in distributed embedded systems: everything from large, multi-core, 64-bit chips running Linux to tiny 8-bit devices with little memory beyond their register set.

MiniMedusa is inspired by Google Protocol Buffers. It is structured as a code generator that takes a set of templates and generates executable code to send and receive them. This technique has been popular in internal applications at Google due to its speed and flexibility [86]. However, MiniMedusa goes beyond current implementations of Protocol Buffers...
to generate not only the interface to the messaging library, but also its implementation. This means that the memory requirements of MiniMedusa are incredibly small, less than 1% of comparable libraries. In a small, but realistic, example, the MiniMedusa library only requires four bytes of RAM.

Section 7.1 discusses the design, implementation and use of MiniMedusa. Section 7.2 compares the memory and execution time of MiniMedusa programs to existing serialization libraries.

7.1 Design and Implementation

MiniMedusa provides a limited form of Medusa pattern matching to C programmers. Users specify patterns in a pattern file that is then automatically converted into C code that receives, decodes and transmits Medusa messages. These patterns are specified in a format nearly identical to Medusa source code. They are read by the standard Medusa toolchain into strings of bytes that are compared to an incoming message at runtime. Critically, this generated code has no understanding of the semantic meaning of the message format. It has no code to parse object headers, atoms or packed tuples. It simply compares them, byte-for-byte, with precomputed templates. These templates mark the position of variables inside the templates so data can be extracted.

This limits the flexibility of pattern matching in MiniMedusa. While the data may vary in incoming and outgoing messages, the structure and length of messages must be predefined. For typical applications, however, where this limitation is acceptable, the resulting reduction in code size is dramatic.
7.1.1 Input format

The programmer specifies the types of messages to be sent and received into a pair of pattern files, one for transmitted packets and one for received packets. These files are essentially annotated Medusa source code. Users can include any immutable Medusa object inside of them: atoms, integers, floating-point numbers, strings and tuples. Objects may also be arbitrarily nested inside of tuples:

```plaintext
('a')
('b', "foo")
('c', "foo", 1, -100,("bar", "baz"))
```

Each line in the pattern file is analogous to a case statement inside a Medusa recv block (see Section 4.2).

As in Medusa, the MiniMedusa programmer may also specify unbound variables to be captured when a message is received. Since MiniMedusa targets C, a typed language, the data type of each unbound must be specified as part the pattern. The programmer specifies unbounds in the form `(c_type) var_name`. This specification doubles as a flag to the compiler that a particular portion of a message will be unbound at run-time and may hold any value:

```plaintext
('a', (uint8_t) num)
('b', (int32_t) other_num)
('c', (uint16_t) yet_another_num)
```

If the controller is sent the message (`'a`, 4), the first pattern will match and the number 4 will be saved to the variable `num`. It should be noted that the maximum value of the data type specified in the pattern constrains the values that can match the pattern. In other words, while the message (`'a`, 4) will match the first pattern, the message (`'a`, 400) will not.
Just like Medusa, MiniMedusa supports complex patterns with more than one unbound variable, unbound variables inside of tuples, and a mixture of unbound variables and constants:

```c
('d', (uint8_t) num1, (uint8_t) num2)
('e', 1, (uint8_t) num)
('f', (1, (uint8_t) num2), (2, (uint8_t num2))
```

MiniMedusa also has support for bound variables in pattern matching. Variables are specified just as unbound variables. The only difference in specification is that bound variables must start with a capital letter, a distinction borrowed from Erlang. MiniMedusa compares the received value in a message with the value stored in a global variable. Only if the two values match does the entire pattern match:

```c
('one-unbound', (uint8_t) num)
('one-bound', (uint8_t) Num)
('mixed', (uint8_t) num1, (uint8_t) Num2)
```

Finally, the user can test if no patterns match an incoming message. This emulates the use of the `Any` keyword in a Medusa `recv` block. It should be noted, however, that MiniMedusa cannot distinguish between a valid message that matches no pattern as compared to a corrupt or invalid message.

To avoid storing excess temporary variables, MiniMedusa does not guarantee that variables will be stored atomically. If a portion of a pattern matches, some variables may be clobbered. Of course, in this case, the programmer should not use any of the variables. All elements must either be specified as constants, bound variables or as unbound variables.

### 7.1.2 Template generation

The MiniMedusa toolchain generates a template for each pattern specified by the programmer. Each template is a string of bytes that will be compared to the incoming message at...
run-time. This process is very simple for patterns that have no unbound variables. The toolchain evaluates the text of the pattern inside of a Medusa virtual machine instance and sends the result as a message back to the toolchain.

For example, the pattern `(a, 1)` is translated into this template:

<table>
<thead>
<tr>
<th>object header</th>
<th># of objects</th>
<th># of bytes</th>
<th>object header</th>
<th>len</th>
<th>val</th>
<th>pad</th>
<th>object header</th>
<th>32-bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 06 00 00</td>
<td>02 00 10 00</td>
<td>07 02 00 00</td>
<td>01 00 61 00</td>
<td>01 02 00 00</td>
<td>01 00 00 00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When a message is received, it is compared byte-for-byte with the template. If the entire message matches the template, the message matches the pattern. If any bytes differ, or the message has a different length from the template, the message does not match the pattern.

When the pattern includes unbound variables, the toolchain must determine which bytes are allowed to change between different messages that match the same pattern. It does this by assigning two values to the variable and translating it through a virtual machine instance twice. First, when the variable holds the minimum value, and again when it holds the maximum value. These two patterns are compared and the bytes that differ are marked as “don’t care” bytes in the pattern. For example, given the pattern `(a, (uint8_t num))`, the pattern is translated twice. Once where `num = 0`:

<table>
<thead>
<tr>
<th>object header</th>
<th># of objects</th>
<th># of bytes</th>
<th>object header</th>
<th>len</th>
<th>val</th>
<th>pad</th>
<th>object header</th>
<th>32-bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 06 00 00</td>
<td>02 00 10 00</td>
<td>07 02 00 00</td>
<td>01 00 61 00</td>
<td>01 02 00 00</td>
<td>00 00 00 00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And again where `num = 255`:

<table>
<thead>
<tr>
<th>object header</th>
<th># of objects</th>
<th># of bytes</th>
<th>object header</th>
<th>len</th>
<th>val</th>
<th>pad</th>
<th>object header</th>
<th>32-bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>05 06 00 00</td>
<td>02 00 10 00</td>
<td>07 02 00 00</td>
<td>01 00 61 00</td>
<td>01 02 00 00</td>
<td>ff 00 00 00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The two templates are compared and the bytes that differ (highlighted in the above examples) are replaced with don’t care variables (??):

<table>
<thead>
<tr>
<th>object header</th>
<th># of objects</th>
<th># of bytes</th>
<th>object header</th>
<th>len</th>
<th>val</th>
<th>pad</th>
<th>object header</th>
<th>32-bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>06</td>
<td>00</td>
<td>02</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>07</td>
<td>02</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01</td>
<td>00</td>
<td>61</td>
<td>00</td>
</tr>
</tbody>
</table>

This process is repeated for each unbound variable. When an incoming message is processed, don’t care bytes in the pattern will not be compared against the message. Instead, these bytes will be stored into C variables. This process is explained in detail in Section 7.1.4.

### 7.1.3 Finite state machine generation

Naïvely storing each pattern in the generated pattern matching routines would require a large amount of storage, at least one byte for each byte in each pattern. This is unnecessarily costly because many patterns share common regions that should not be stored multiple times. For example, different messages to a particular peripheral may be similar:

- ('motor-controller', 'set-direction', 'forward')
- ('motor-controller', 'set-direction', 'backward')
- ('motor-controller', 'set-power', (uint8_t) power_level)
- ('motor-controller', 'stop')

Pattern matching with discrete finite automata is well-known to be a good solution to this problem. Such algorithms are very efficient, expending pre-computation time to save space and time in the generated code. Each input character only needs to be examined once and the algorithm runs in constant time with respect to the number of patterns [87].

MiniMedusa implements a DFA pattern matching algorithm by transforming the set of message templates into a single tree, representing all templates. An example is shown
in Figure 7.1, created by combining the templates "abcd", "abef", and "abg". The match process starts at the root, n0. If a node only has one outgoing arc, the generated code compares the incoming byte with the stored arc. If the bytes match, it increments to the next node, proceeding down the tree.

If a node has more than one outgoing arc, it generates a “control node”, a small block of C code that is part of a switch statement. The generated code for n2 is shown in Figure 7.2.

If the incoming byte is either ’c’ or ’e’, the DFA transitions to the appropriate next node to compare the rest of the pattern. If the incoming byte is a ’g’, we know that the match has succeeded. The DFA sets a global variable stopping the search and stores the pattern that matched. If the incoming byte is something else, the match is stopped and signals that no match happened.

### 7.1.4 Saving unbound variables

Bytes that are part of an unbound variable are represented in the tree as a node whose outbound arc is a “don’t care” byte. These nodes are also a form of control node. An example of such an unbound node:

```c
    case n5:
        // store the incoming data
        unbounds.p0.num = unbounds.p0.num << 8;
        unbounds.p0.num |= new_byte;

        // move on
        current_node++;

    break;
```

First, the incoming data is stored in the unbound variable. Since the DFA is run one byte at a time, the data is shifted into the variable as it becomes available. MiniMedusa uses big-
Figure 7.1: A generated search DFA.
```c

case n2:
    if ( new_byte == 'c' ) {
        current_node = n3;
    }

    else if ( new_byte == 'e' ) {
        current_node = n4;
    }

    else if ( new_byte == 'g' ) {
        current_node = STOPPED;
        matched_pattern = PATTERN_ABG;
    }

    else {
        current_node = STOPPED;
        matched_pattern = MATCH_ANY;
    }

    break;

Figure 7.2 : The generated control node for n2 in Figure 7.1.
```
 endian, the byte order from ARM, so the most significant byte is sent first. As subsequent bytes are sent, the entire multi-byte message is stored. Finally, the DFA transitions into the next node to match the rest of the message.

On little-endian systems, the user calls library functions before reading data from or sending data to MiniMedusa to convert data into big-endian format.

7.1.5 Testing bound variables

Internally, MiniMedusa treats bound variables just like unbound variables. They are stored into the unbounds union using the process described above.

At the last node of a particular pattern, the generated code sets the variable MATCHED_PATTERN to specify which pattern actually matched the incoming message. However, if the pattern contains bound variables, the value received must match the value stored locally. The code generator produces an extra line of code for each bound variable in the final node of each pattern:

```c
    case n5:
        // node may match
        if (bound_p1_Num2 == unbounds.p1.Num2)
            if (new_byte == 0x00) {
                current_node = STOPPED;
                matched_pattern = PATTERN_NUM2;
            }
```

7.1.6 Sending messages

The process for sending messages is roughly analogous to receiving them. A template is generated for each message to be sent. Instead of storing this data in a tree, each template is stored as the raw sequence of bytes representing the message. A tree structure is not necessary since the generated code is not trying to do any sort of pattern matching that
would benefit from a DFA is not required. When the user sends a message, the program selects which message is to be sent through a function call. From there, the generated code sends the selected message a byte at a time.

The user can specify variables to be sent in a message using the same pattern format used to specify unbound variables. The generated code includes a union of structures, one for all the variables in each pattern. This is analogous to the structure that stores the values for unbound variables in incoming messages. As the generated code is transmitting a message, it will send values from these structures in lieu of bytes from the pre-computed pattern at their appropriate positions in the message. These positions are computed by translating the message with the maximum and minimum possible value for each variable, the same process to generate input data don’t-care bytes.

7.1.7 MiniMedusa API

The programmer interacts with generated MiniMedusa code a very lightweight API. This interface is designed more for size than elegance.

Whenever a byte is received, the user calls `hoot_rx(uint8_t new_byte)` with the new data. The DFA processes one byte, transitioning one node. It then returns to the user program. When an entire message has been received, the user program queries the current matched pattern by reading from the global variable `matched_pattern`. Any variables that were bound by the process are also stored in global variables.

To send a message, the user calls `hoot_send(uint8_t msg_num)` with the appropriate pattern to send. It then calls `hoot_rx(uint8_t out_byte)` for each byte in the output message to send. When the entire message has been sent, `hoot_send` returns control to the user program.
7.2 Results

To evaluate the MiniMedusa system, a simple program was written in C for the Atmel AT90USB646 8-bit microcontroller on the PJRC Teensy development board. The AT90USB646 has 64 KB of Flash, 4 KB of SRAM and operates at 16 MHz. In 2014, it retailed for $5.00 in large quantities.

The test program reads and writes messages to a simulated communications interface to avoid the overhead of a real peripheral. It reads two types of messages and sends a third in response. A Medusa version of the program is shown in Figure 7.4. Using the same libraries used in Section 6.3, four C programs were written and tested: one using JSON, one using Protocol Buffers, one using auto-generated MiniMedusa, and a final variant of the MiniMedusa program that was hand-optimized for AVR. These applications are similar in functionality and purpose, but the different message formats require each version to use slightly different messages (Figure 7.3). The JSON implementation uses strings in place of atoms and the Protocol Buffer implementation uses an enumeration to specify the message type.

Memory requirements for AVR were calculated using static binary analysis to measure the size of the bss, data and text segments of the compiled program. The program’s stack use was measured by painting the stack at runtime and measuring how much was overwritten. The heap use was measured by modifying avr-libc’s malloc routine to keep track of the highest address of the top of the heap. This is more accurate than simply keeping track of the total heap requested since it corresponds to the minimum heap size required to run the program. Finally, each measurement was measured relative to the overhead of a dummy program that performs no parsing.

Table 7.1 compares the size and speed of the four programs measured. It shows that
# MiniMedusa
'blink', (uint8_t) num
'ping'
'pong'

# JSON
["blink", 3]
["ping"]
["pong"]

# Protocol Buffers
message msg {
  enum Type {BLINK = 1; PING = 2; PONG = 3; }

  required Type type = 1;
  optional int32 num = 2;
}

Figure 7.3: Message formats used by the MiniMedusa, JSON and Protocol Version of the benchmark program.

def loop():
    recv:
        case ('blink', num):
            blink(num)
        case 'ping':
            'cortex'.send('pong')

    return loop()

Figure 7.4: The MiniMedusa example program shown as a standard Medusa program.
### Table 7.1: Memory use and running time for the tested serialization formats on AVR8. Memory usage is listed in bytes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>bss data</th>
<th>text</th>
<th>stack</th>
<th>heap</th>
<th>RAM</th>
<th>ROM</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>json</td>
<td>35 74 10958 57 191</td>
<td></td>
<td></td>
<td></td>
<td>357</td>
<td>11032</td>
<td>596 ns</td>
</tr>
<tr>
<td>protobuf</td>
<td>8 286 16898 267 32</td>
<td></td>
<td></td>
<td></td>
<td>593</td>
<td>17184</td>
<td>264 ns</td>
</tr>
<tr>
<td>minimedusa</td>
<td>4 90 198 0 0</td>
<td></td>
<td></td>
<td></td>
<td>94</td>
<td>288</td>
<td>119 ns</td>
</tr>
<tr>
<td>minimedusa-opt</td>
<td>4 0 290 0 0</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>290</td>
<td>125 ns</td>
</tr>
</tbody>
</table>

Both the auto-generated and optimized MiniMedusa code run twice as fast as the Protocol Buffer code and almost five times as fast as the JSON implementation. MiniMedusa also uses less than a quarter of the required SRAM of either competing format, even without any hand-optimization. By modifying the generated code to place constants in program space, the MiniMedusa parser only requires four bytes: one to record which pattern matched, one to store the unbound variable in the (`blink`, `num`) message, and two to store a pointer into the trees.

Even more striking is the reduction in program size between the message formats. The JSON and Protocol Buffer implementations use parsers that understand each element of the incoming message. This is flexible because it can interpret any arbitrary message but it is very costly. Both parsers scan for tokens inside the input messages, converting each internal object into a C representation on the C heap. This requires a great deal of code, including hundreds of bytes of code from complex routines in the C standard library (Figure 7.5).

On the other hand, the MiniMedusa generated code needs no such process: it simply compares bytes and shifts numerical values into C variables. No parsing is required. This means that the entire MiniMedusa code consumes less than 300 bytes of flash, over a 97% reduction from its competitors.
Figure 7.5: From top to bottom: code size for a dummy program, the un-optimized and optimized MiniMedusa programs and the JSON and Protocol Buffer programs.
7.3 Discussion

Hoot extended the Actor model of programming from a single controller to a network of virtual machines by exploiting packed tuples and the message passing provided by Medusa. MiniMedusa extends the boundary even further to the small controllers that would be otherwise incapable of running even the simplest of virtual machines.

In a system that uses a well-defined message format between devices, a great deal of interface code can be eliminated. Indeed, over a third of the code in the autonomous model car example (344 of 893 lines) is dedicated to encoding and decoding messages for peripherals. In this application, the controller communicates with the servo, the motor controller, the GPS receiver and the rangefinder, each over a completely different interface. Experience has shown that this is one of the most tedious and error prone parts of embedded programming.

MiniMedusa can allow a suitably designed system to send and receive messages directly from a peripheral, through a Hoot network, to and from a particular Actor. The system is completely agnostic to the communications system. It has been tested with both RS-232 serial ports and UNIX pipes.

Finally, by limiting the message interface to a limited number of message formats, MiniMedusa can be much smaller than existing libraries. It can run in only a few bytes of SRAM. This allows the scope of the Medusa system to scale down to very small peripherals. These devices have limited functionality, usually just sending and receiving a handful of messages. MiniMedusa allows them to use an interface library appropriate to that level of complexity.
Conclusions

There are orders of magnitude more embedded microcontrollers in the world than conventional microprocessors, yet they are much harder to program. Programmers of web applications have rich, multi-level stacks that allow them to build presentation layers and relational databases in literally minutes. Scientific and high-performance computing programmers have auto-parallelizing and just-in-time compilers. Students around the world can learn to write interactive games in their web browsers with just a few hours of instruction over the web.

Embedded programmers are forced to use much older technology. Simply determining how much memory a single program uses is a surprisingly herculean task. Standard compilers either cannot report memory usage or can only report static memory usage like code and data, ignoring heap and stack use. Being forced to use such primitive tools makes it difficult for beginners to even start building embedded systems. Expert programmers end up spending more time debugging code than they do inventing and creating. As a result, there is a proliferation of low-level embedded software that is difficult to write, difficult to test, and difficult to port to new systems.

The amazing success of platforms like Arduino shows that there is great demand for tools that make building embedded systems easier. There was no new hardware on the Arduino; flash programmable microcontrollers have been available for decades. The rev-
olution comes in the software. Users can get started quickly and they have access to rich libraries. They don’t need to write assembly or modify C memory allocators. As a result, such systems have allowed huge numbers of people to make new and amazing things. Microcontrollers have moved from a specialized hobby to a trend raved about in the New York Times and the Wall Street Journal [40, 88].

As these software tools become more complicated and more important, they must be developed into serious tools and their properties carefully studied. This thesis has presented Owl, a complete development environment for developing complex embedded systems. The Owl virtual machine contains novel features that allows it to host larger programs than ever before.

Our new programming language, Medusa, fundamentally changes the way embedded systems are written. It demonstrates that Actors, a model of programming designed for much larger computers, is especially well-suited to building embedded systems. Moreover, it shows that the overhead of including support for the Actor model is minimal in embedded run-time systems. For only a few kilobytes, the Owl virtual machine can provide support without sacrificing backward compatibility.

This thesis introduces bridging, a new mechanism for managing I/O in a microcontroller. It bridges the gap between polling and interrupt-based designs. The results presented here show that it is possible to get non-blocking performance with code that is not much more complicated to write than a blocking implementation. In such a system, it is impossible to introduce the sort of deadly race condition that has plagued embedded systems.

Finally, Hoot and MiniMedusa expand the scope of the Actor model from concurrent to distributed systems. Developers can connect a wide variety of computers, from 8-bit microcontrollers to multi-core x86 processors into a cohesive programming environment.
A thread running anywhere can send a message to any other thread in the system just as if it was a local thread. This makes creating large-scale systems such as cars, airplanes, or an entire power grid much easier.

8.1 Future work

The challenge of building large embedded systems is not going away. The embedded systems of the future will be more complex, more concurrent, and more distributed. Even today, thermostats have networking interfaces, washing machines have suites of environmental sensors and home security systems perform real-time computer vision. This thesis has answered some questions about how such systems should be built. The process of completing it, however, has raised many more. Owl will make an excellent platform for much of this research, and I am proud to be able to provide it to other researchers.

One obvious area of research is in security. Currently, Hoot makes no attempt to be secure. Any thread can send a message to any other thread, local or remote. The recipient thread cannot even tell where that message came from. Medusa and Hoot do, however, provide the framework for addressing this deficiency. All messages travel through the virtual machine and can be authenticated and verified in one place. In fact, all messages that are sent to remote destinations can be verified inside of the Python code in Hoot, which will make implementing an authentication scheme easier.

This is an important concern in embedded systems. Many security researchers have identified specific bugs in existing products that lead to vulnerabilities [10, 11, 12, 13, 14, 15]. Comparatively less work has been done to propose systematic solutions to avoiding such bugs. Larger systems have proposed comprehensive analysis and protocol based features to ensure that messages come from where they are supposed to and only permissible actions are carried out by nodes [89, 90, 91]. The web is a massively successful example
of such a system. Browsers communicate with servers over well-defined protocols, HTTP and SSL, and their behavior is exhaustively researched [92, 93, 94, 95]. If the communications protocol between controllers is well-defined, as is the case in Medusa and Hoot, this research can be applicable to microcontrollers.

Another potential arena for future work is in the language design and analysis of systems like Medusa. We designed Medusa so that it would be approachable to beginners and easily comparable to programs written in Python. For this goal, it has been successful. The results in this thesis show that the actor model is especially well suited to building concurrent and distributed embedded systems.

Moving forward, the Actor model is also well-suited to formal verification of programs [96, 97, 98, 99, 65]. The lack of mutable state, both local and shared, makes such verification of a Medusa program potentially much easier than it is for programs written in C. We can and should adapt tools written for systems like Erlang to our environment. This would allow a programmer to prove that their Medusa programs will run within some specifications. For example, it would be possible to prove isolation between tasks. You could show that no matter what happened inside one component, it wouldn’t change the operation of another. This verification would be invaluable in safety-critical systems.

This thesis has focused on execution time and memory use as the constrained resources on a microcontroller, but increasingly, energy is becoming critical. Devices are expected to last longer and longer, even with thinner and thinner batteries. Current research has addressed automatic power management for cellular handsets [100, 101], desktops and notebooks [102, 103] and datacenters [104, 105, 106]. If these developers have support from system software for power management, there is no reason that the developer of a microwave oven or remote control should not. Just as it is possible to create an adaptive scheduler for Medusa, it should be possible to create a power manager as well.
There are virtually no realistic benchmarks available for microcontroller applications. Rich suites of benchmarks are available for supercomputers [107], workstations [108] and even large data-centric embedded applications that run on DSPs [109, 110]. They are derived from programs used in the real world. They are relatively easy to work with and allow researchers to quickly see how their innovations will improve practical applications. Microcontrollers have no such suites. I implore the users of embedded toolchains, the manufacturers of cars and avionics and consumer products, to release as much source code as they can. While this certainly has business implications, it would make it possible for the research community to dramatically improve their productivity in the long run.

Finally, the complete Owl stack has the potential to be used in very large systems. One such system would be an x86-based notebook, something not generally considered to be a distributed embedded system. While the lion’s share of computational power in such a system is provided by the x86 CPU, such a system has a large collection of other microcontrollers that handle other tasks. When the user opens the computer, a microcontroller detects this through a button inside the latch. It then sends a message to other controllers, devices that control power management, display backlight, radios and countless other peripherals. In fact, current versions of the Stellaris microcontrollers used in this thesis were designed for this purpose. It is said that at least one is used in every MacBook and MacBook Pro.

Currently, these controllers are linked in a haphazard network of legacy hardware and convoluted protocols. Medusa and Hoot provide the opportunity to start fresh and connect all of these devices into a single system. Instead of being a loosely connected set of components, the system developer can build a modern computer as a single distributed system. The new capabilities such a design could bring are limitless.

As they continue to grow more complicated, the research community must take mi-
microcontrollers seriously. This research can, at times, be painful. Building a new software development system requires using existing development systems that are far from ideal. However, it can also be extremely rewarding. I have found that it is wildly more fun to see your thesis driving down the hallway as everyone comes out to watch, or seeing your thesis light up the face of a child writing her first line of code than it is to see your thesis result improve a benchmark performance by a few percent. This work has the potential to change the world, too. Potential to let new people build amazing new machines. That potential comes when we realize what microcontrollers really are: a complete computer system.
Debouncing in C

```c
#include "inc/lm3s9b92.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"

#define SWITCH_SYSCTL_PORT SYSCTL_PERIPH_GPIOB
#define SWITCH_GPIO_PORT GPIO_PORTB_BASE
#define SWITCH_GPIO_PIN GPIO_PIN_4

bool current_state, bouncing;
xSemaphoreHandle bouncing_lock;
xSemaphoreHandle output_count;

void init (void);
void __attribute__((interrupt)) timer0a_isr (void);
void __attribute__((interrupt)) gpiob_isr (void);
void spin (void);
void user (void);
void main (void);

void init (void)
{
    /* Start the GPIO and the timer */
    SysCtlPeripheralEnable (SWITCH_SYSCTL_PORT);
    SysCtlPeripheralEnable (SYSCTL_PERIPH_TIMER0);
```
/ Wait for the hardware to come up. */
for (ulLoop = 0; ulLoop < 10; ulLoop++) {}

/* Enable the right modes for the GPIO */
GPIOPinTypeGPIOInput(SWITCH_GPIO_PORT, SWITCH_GPIO_PIN);
GPIOPadConfigSet(SWITCH_GPIO_PORT, SWITCH_GPIO_PIN, GPIO_STRENGTH_2MA,
    GPIO_PIN_TYPE_STD_WPU);
GPIOPinIntEnable(SWITCH_GPIO_PORT, SWITCH_GPIO_PIN);
GPIOPinIntTypeSet(SWITCH_GPIO_PORT, SWITCH_GPIO_PIN, GPIO_BOTH_EDGES);

/* And the timer ... */
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_OS);
TimerLoadSet(TIMER0_BASE, TIMER_A, RESET_VAL);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

/* configure the semaphores and reset values of global variables */
vSemaphorCreateBinary(bouncing_lock);
xSemaphorCreateCounting(MAX_INT, 0);
    current_state = false ;
bouncing = false ;

/* Be sure to enable the interrupts LAST */
IntMasterEnable();
}

/* clear the interrupt. It's a one-shot, so it resets itself */
TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

// acquire the lock and set the global variable
xSemaphoreTake(bouncing_lock);
bouncing = false ;
xSemaphoreGive(bouncing_lock);
}

void __attribute__ (( interrupt )) timer0a_isr (void) {
    /* clear the interrupt */
    TimerIntClear(TIMER1_BASE, TIMER_TIMA_TIMEOUT);

    // acquire the lock and set the global variable
    xSemaphoreTake(bouncing_lock);
bouncing = false ;
xSemaphoreGive(bouncing_lock);
}

void __attribute__ (( interrupt )) gpiob_isr (void) {
    uint8_t values ;

    /* clear the interrupt */
    GPIOPinIntClear(SWITCH_GPIO_PORT, ALL_PINS);

    /* read the value of the port */
    values = GPIOPinRead(SWITCH_GPIO_PORT, ALL_PINS);

    /* read out just the one pin directly into the global var */
    if (values & SWITCH_GPIO_PIN)
    {
        current_state = true ;
    }
else
{   current_state = false;
}

// acquire the lock and set the global variable
xSemaphoreTake(bouncing_lock);
if (!bouncing)
{
    /* start up the timer */
    TimerLoadSet(TIMER0_BASE, TIMER_A, RESET_VAL);
    TimerEnable(TIMER0_BASE, TIMER_A);
}
bouncing = true;
xSemaphoreGive(bouncing_lock);

void spin(void)
{
    while (1) {}
}

void user(void)
{
    void debounced_button = false;

    while (1)
    {
        xSemaphoreTake(output_count, portMAX_DELAY);
        debounced_button = !debounced_button;

        if (debounced_button)
        {
            lib_printf ("down\n");
        }
        else
        {
            lib_printf ("up\n");
        }
    }
}

void main(void)
{
    xTaskHandle xDebounce, xUser;

    /* spin up the user thread */
    xTaskCreate(user, "user", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xUser);

139  init ();
140  spin ();
141  }
APPENDIX B

Autonomous model car
Figure B.1: Autonomous car hardware.
Figure B.2: Autonomous car schematic.

Figure B.3: Autonomous car printed circuit board layout.
Bibliography


[50] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, “Java on the bare metal of wireless sensor devices: the squawk java virtual machine,” in *Proceedings of the*


