Medusa: Managing Concurrency and Communication in Embedded Systems

Thomas W. Barr and Scott Rixner, ***Rice University***

https://www.usenix.org/conference/atc14/technical-sessions/presentation/barr

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Medusa: Managing Concurrency and Communication in Embedded Systems

Thomas W. Barr
Rice University

Scott Rixner
Rice University

Abstract

Microcontroller systems are almost always concurrent, event-driven systems. They monitor external events and control actuators. Typically, these systems are written in C with very little support from system software. The concurrency in these applications is implemented with hand-coded interrupt routines. Race conditions and other classic pitfalls of implementing parallel systems in shared-state programming languages have caused catastrophic, and sometimes lethal, failures in the past.

We have designed and implemented Medusa, a programming environment for microcontrollers using the actor model. This paper presents three key contributions. First, the Medusa language, which is derived from Python and Erlang. Second, an implementation that runs on systems several orders of magnitude smaller than any other actor-model system previously described. Finally, a novel bridging mechanism to extend the domain of the actor-model to hardware. Combined, these innovations make it far easier to build complex, reliable and safe embedded systems.

1 Introduction

This paper presents the design, implementation, and evaluation of Medusa, a high-level language for embedded microcontrollers. Medusa integrates hardware and software messaging to provide a robust, easy to use concurrent programming environment ideally suited for small microcontrollers.

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. However, the event-driven nature of these systems leads to concurrency and synchronization issues that are notoriously difficult to manage, even in large scale systems with system software support [10, 28, 29]. In embedded systems, the programmer is largely unaided in dealing with these complex issues. Designing better programming systems for small scale microcontrollers is becoming critically important as these devices proliferate.

At best, embedded systems utilize a real-time operating system (RTOS) to help manage concurrency. These systems provide primitive, low-level mechanisms for thread scheduling, synchronization, and communication [2, 5, 9, 18]. However, 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. This is difficult and error-prone.

Medusa is a programming language and run-time system for developing concurrent microcontroller-based systems designed to address these issues. Medusa is based on the actor model of concurrency [15, 1]. The actor model solves the fragmented control flow and shared state problems inherent in traditional approaches to event-driven programming [10]. The Medusa programming model and run-time system utilize and expand upon these ideas for small embedded systems.

This paper presents three practical contributions. First, it presents a new actor-based programming language that is implemented as a small set of extensions to Python, a popular and expressive programming language. Medusa's 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.

Second, we present an implementation of the Medusa system as a set of extensions to Owl, our open-source embedded Python implementation [4]. 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. 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.

Finally, this paper presents a novel bridging mechanism to extend the domain of the actor model to the hardware. This mechanism is fast; accepting, converting and storing an external event takes less than 4 microseconds. Further, these bridges simplify one of the more difficult challenges of embedded systems software development: dealing with concurrent interrupts. By presenting the exact same abstraction for communication both among software threads and hardware, the programming model is more uniform, making it easier to design and implement robust and reliable embedded systems.

The following section discusses background and related work. Section 3 describes the Medusa language itself. Sections 4 and 5 describe the implementation of the messaging, bridging and toolchain systems of Medusa in detail. Section 6 presents a quantitative analysis along with real-life applications built with Medusa. Finally, we conclude in Section 7.

2 Background and Related Work

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. While these tools help the programmer find problems, they do little to simplify the task of writing and maintaining low-level embedded software.

These tools have evolved to perform static analysis to detect specific, common bugs. One of the most common themes of these systems is the detection of data races [17, 27]. We believe, however, that the best way to prevent these common mistakes is to preclude them in the programming environment. Recent research efforts have started to move in this direction by introducing new, higher-level programming languages and new programming models to small systems. Medusa does exactly this. It extends our open-source managed run-time to directly support the actor model of event driven programming. In the Medusa system, these static analysis tools become unnecessary because *it is impossible for a Medusa programmer to introduce any of these types of bugs.*

2.1 Embedded managed run-time systems

Early commercial embedded run-time systems, such as the BASIC Stamp [20], were quite primitive. As such, they never moved far past educational uses. Similarly, academic projects were largely focused on extremely small 8-bit devices [22].

Since then, microcontrollers have grown in size and capability, so they can support more capable run-time systems. The Java Card environment allows a small subset of Java to run on 16-bit microcontrollers [6]. The recent availability of 32-bit microcontrollers has allowed for the creation of much larger virtual machines, such as Squawk for Java [26] and Owl for Python [4]. The open-source Owl toolchain and virtual machine serves as the base for the Medusa system. In total, the Medusa Virtual Machine consists of 24,200 lines of C.

On high-end embedded systems like phones, managed run-time systems are nearly ubiquitous. The Android system runs on top of Dalvik, a bytecode virtual machine and the iPhone runs on top of the Objective-C 2.0 automatic garbage collection and reflection runtime.

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[1] 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 requirements of Medusa.

2.2 Actors

The actor model [15, 1] 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 [10].

The actor model was first described in 1973 by Hewitt, Bishop and Steiger [15] 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 [7], laws for

[1] http://www.erlang.org/faq/implementations.html

actor systems [14], commitment [13] and formal analysis of divergence and deadlock [1].

Many programming languages have been constructed to directly support the actor model, including Act [24], Erlang [3], Scala [25] and Go.[2] 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) [16] and efficient systems for generating and passing object references [12]. Many of these languages, most notably Go and Erlang, contain threading implementations that are lighter than operating system threads [3, 11] but are still more resource intensive than Medusa. Go and Scala require more than an order of magnitude more memory than Medusa.

TinyOS is another microcontroller programming environment that proposes a new programming model. In this *split-phase model*, programs are divided into blocks that start long-running operations, but do not wait for them to complete. Instead, they are notified that operations have succeeded through callbacks [23]. 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 [19].

Scala and Erlang both provide a messaging layer that is similar to Medusa's software-to-software messages; they can transport complex objects and have sophisticated pattern matching abilities. Candygram, a Python library, provides similar facilities. However, it does not include new syntax for messaging and pattern matching nor lightweight threads. On a desktop computer, Candygram threads were measured to require 3.5 KB of memory each, several hundred times what is required in Medusa.[3] The Go language provides a messaging layer that is easy to access from low-level code but does not provide composite object messaging or pattern matching.

3 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. Medusa includes several key features derived from Erlang: light-weight threads, messaging, pattern matching, and atoms.

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:

```
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)))
```

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

Figure 1: Basic Receive Statement.

Once the actor has been started, it receives a message using the `recv` statement, shown in Figure 1. 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.

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 the 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, 1205)`. 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.

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

[2] http://golang.org/

[3] http://candygram.sourceforge.net/

```
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 2: Medusa echo program.

```
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 languages, 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:

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

Finally, Figure 2 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 Implementation

The Medusa system is implemented as a set of extensions to our previously-published embedded Python system, Owl [4]. 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. While our implementation was designed around the existing Owl system, our design should be generalizable to implementing lightweight messaging and pattern matching in any bytecode virtual machine.

4.1 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) <- (12, 3, 4)`

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

```
Offset  Bytecode           Argument

     0  LOAD_CONST         (12)
     3  LOAD_NAME_UNBOUND  (a)
     6  LOAD_NAME_UNBOUND  (b)
     9  BUILD_TUPLE        3

    12  LOAD_CONST         (12)
    15  LOAD_CONST         (3)
    18  LOAD_CONST         (4)
    21  BUILD_TUPLE        (3)

    24  COMPARE_OP         (<-)
```

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 unbound 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.

4.2 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 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.

The implementation of the receive process is illustrated in Figures 1 and 3. Figure 1 shows a simple receive block with more than one case. Figure 3 shows the compiled bytecode for this basic receive statement.

```
Offset  Bytecode           Argument

     0  UNDEFER_MSG
     1  RECV_MSG

     2  DUP_TOP
     3  LOAD_CONST         (1)
     6  ROT_TWO
     7  COMPARE_OP         (<-)
    10  POP_JUMP_IF_FALSE  21

    13  LOAD_CONST         ("received 1")
    16  PRINT_ITEM
    17  PRINT_NEWLINE
    18  JUMP_ABSOLUTE      44

    21  DUP_TOP
    22  LOAD_CONST         (2)
    25  ROT_TWO
    26  COMPARE_OP         (<-)
    29  POP_JUMP_IF_FALSE  40

    32  LOAD_CONST         ("received 2")
    35  PRINT_ITEM
    36  PRINT_NEWLINE
    37  JUMP_ABSOLUTE      44

    40  DEFER_MSG
    41  JUMP_ABSOLUTE      1

    44  POP_TOP
```

Figure 3: Bytecodes for code in Figure 1.

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 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

```
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(m1tid, m2tid):
    recv:
        case (m1tid, 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(m1tid, m2tid)
```

Figure 4: Simple Messaging Example.

and ample opportunity for compiler optimization for special cases.

4.3 Example

Figure 4 shows a simple example of how messaging works. In this example, two monitor threads are spawned which wait for arbitrary events. When they receive a message, they send a message back to the main control thread. Here, the messages are simple strings, 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 `(m1tid, msg)`. In this tuple, the first variable, `m1tid`, is already bound, whereas the second, `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, `(m2tid, msg)`, will be tried. If either of these matches succeed, the `msg` variable will be bound to the data from the second element of the tuple in the message. The last case with the pattern `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

```
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 5: Prioritizing Messages.

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, `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 5. 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 `recv` 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 demonstrate 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.

4.4 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 [8]. 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.

5 Bridge architecture

Current embedded run-time systems, such as the Owl system that Medusa is based upon, require polling to detect hardware events. Applications must repeatedly check to see if an event has occurred inside peripheral hardware. In contrast, C programs often use interrupts, which requires the use of shared, mutable state.

Medusa introduces a novel third option by *bridging* the domain of the actor model to hardware. Bridges allow interrupt service routines (ISRs) to communicate with software threads. ISRs written using bridges are extremely simple, often just a few dozen lines of C, and all interaction with the virtual machine happens through a safe, secure interface. This interface makes it impossible to introduce a synchronization bug or race condition. With bridges, hardware modules and software threads communicate using the same message passing system.

5.1 Programming with Interrupts

Inside the microcontroller, there are hardware agents that monitor the state of peripherals constantly, even when the core is busy with other work or is sleeping. These agents detect external events that can be selected by the programmer. A user may want to detect level changes on a particular general-purpose I/O (GPIO) port, incoming serial data or a completed analog to digital conversion. When a selected event occurs, the hardware interrupts normal program execution and calls a user function.

Systems written in C use hand-coded interrupt service routines (ISRs) to respond to these events. 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. 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.

5.2 Interrupt Bridging

Medusa solves these two problems with a technique 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-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 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. 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

```
#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 6: The GPIO interrupt service routine.

on any given port. Therefore, the bridge is set up to send five byte messages:

```
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:

```
# 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 6) 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.

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.

5.4 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.

6 Evaluation

To evaluate the Medusa system, we constructed and measured the performance of several microbenchmarks and several embedded applications. We compare applications that use the Medusa systems of messaging and bridging with procedural, polling-based applications that run on the standard Owl system. These experiments were run on the Texas Instruments Stellaris LM3S9B92, an ARM Cortex-M3 microcontroller operating at 50 MHz, with 96 KB of SRAM and has 256 KB of flash.

6.1 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. On average, it takes about 59*us* to

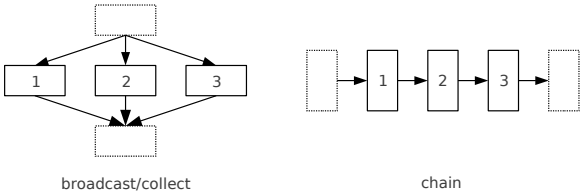

Figure 7: The two message benchmarks: broadcast/collect and chain. The number of workers (3 in the figure) is configurable.

create a thread on a 50 MHz Cortex-M3 processor. It also makes context switches quickly: about 129*us* on average to context switch while running 100 threads.

6.2 Messaging performance

To measure the speed of message passing, two simple benchmarks were constructed, shown in Figure 7. 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 node. 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 8, 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 benchmarks. 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.

6.3 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. 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 9.

In the polling implementation, the thread reading the input must wait until it is scheduled 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.

6.4 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. 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 10 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

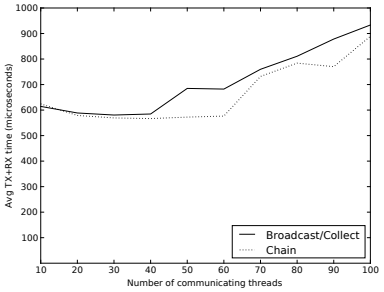

Figure 8: Average time to send and receive a message for different benchmarks.

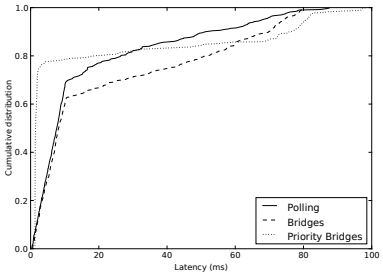

Figure 9: Distribution of I/O latency for polling and bridges.

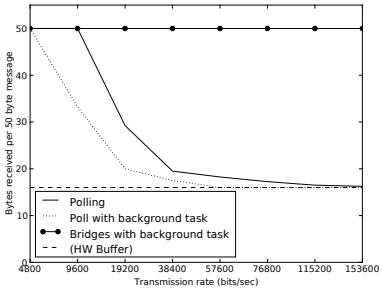

Figure 10: Data loss rates for different transmission rates.

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.

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.

6.5 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.

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.

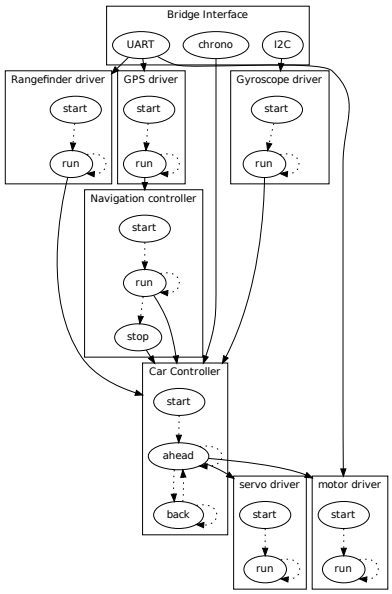

Figure 11: Diagram of Event-driven Autonomous Car.

Figure 11 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 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 5.4), performing the task, and repeating:

```
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 it 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.

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 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.

7 Conclusions

Building reliable embedded systems has long been a challenging endeavor. In the 1980s, the Therac-25, a radiation therapy device, suffered several failures due to a race condition between the main program and a hand-coded interrupt service routine. As a result, at least six patients were overdosed and three died. [21] Unfortunately, not much has changed since then. 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[4] caused widespread blackouts in 2005. More recently, the suspect systems in the Toyota unintended acceleration case used shared-state and hand-coded interrupt handlers. [5]

Medusa is designed specifically to advance the state-of-the-art in this area. Pure functional programming with an efficient messaging system leads to much simpler concurrent programming. That is why such systems are used everywhere from telephone switches to the Facebook messaging platform. Medusa demonstrates that it is possible to take the best ideas from these systems and implement them on resource-constrained systems. Further, Medusa is the first embedded language run-time system capable of integrating hardware interrupts into the software messaging system in a seamless manner.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] T. N. B. Anh and S.-L. Tan. Real-time operating systems for small microcontrollers. *IEEE Micro*, 29(5), 2009.

[3] J. Armstrong. Erlang - a survey of the language and its industrial applications. In *In P. symposium on industrial applications of Prolog (INAP96)*, 1996.

[4] T. W. Barr, R. Smith, and S. Rixner. Design and implementation of an embedded python run-time system. In *P. 2012 USENIX ATC*, 2012.

[5] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of embedded real-time operating systems. *IEEE Trans. Computers*, 52(11), 2003.

[6] Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, Boston, MA, USA, 2000.

[7] W. D. Clinger. Foundations of actor semantics. Technical report, Cambridge, MA, USA, 1981.

[8] E. D'Osualdo, J. Kochems, and C. H. L. Ong. Automatic verification of erlang-style concurrency, 2013.

[9] J. Gannssle. The challenges of real-time programming. *Embedded System Programming Magazine*, 2007.

[10] P. Haller and M. Odersky. Event-based programming without inversion of control. In *In Proc. Joint Modular Languages Conference (2006), Springer LNCS*. Springer, 2006.

[11] P. Haller and M. Odersky. Actors that unify threads and events. In *P. 9th international conference on Coordination models and languages*, COORDINATION'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *P. 24th European conference on Object-oriented programming*, ECOOP'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] C. Hewitt. Coordination, organizations, institutions, and norms in agent systems ii. chapter What Is Commitment? Physical, Organizational, and Social (Revised). Springer-Verlag, Berlin, Heidelberg, 2007.

[14] C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress'77*, 1977.

[15] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *P. 3rd international joint conference on Artificial intelligence*, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[16] F. Huch. Verification of erlang programs using abstract interpretation and model checking. In *P. fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, New York, NY, USA, 1999. ACM.

[17] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *J. Log. Program.*, 13(2-3), July 1992.

[18] D. Kalinsky. Basic concepts of real-time operating systems. *LinuxDevices Magazine*, 2003.

[19] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys*, volume 9, pages 127–140, 2009.

[20] C. Kuhnel and K. Zahnert. *BASIC Stamp: An Introduction to Microcontrollers*. Newnes, Woburn, MA, USA, 2000.

[21] N. Leveson and C. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7), july 1993.

[22] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *P. 10th ASPLOS*, 2002.

[23] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[24] H. Lieberman. Thinking about lots of things at once without getting confused. Technical Report 626, Massachusetts Institute of Technology Artificial Intelligence Laboratory, May 1981.

[25] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausane, 2004.

[26] 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 *P. 2nd international conference on Virtual execution environments*, VEE '06, New York, NY, USA, 2006. ACM.

[27] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. Softw. Eng.*, 6(3), May 1980.

[28] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2), Feb. 1973.

[29] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *P. 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[4] http://www.securityfocus.com/news/8412

[5] http://www.nhtsa.gov/UA/