Design and Implementation of an Embedded Python Run-Time System

Thomas W. Barr Rebecca Smith Scott Rixner
Rice University
{*twb, rjs, rixner*}*@rice.edu*

Abstract

This paper presents the design and implementation of a complete embedded Python run-time system for the ARM Cortex-M3 microcontroller. The Owl embedded Python run-time system introduces several key innovations, including a toolchain that is capable of producing relocatable memory images that can be utilized directly by the run-time system and a novel foreign function interface that enables the efficient integration of native C code with Python.

The Owl system demonstrates that it is practical to run high-level languages on embedded microcontrollers. Instrumentation within the system has led to an overall system design that enables Python code to be executed with low memory and speed overheads. Furthermore, this paper presents an evaluation of an autonomous RC car that uses a controller written entirely in Python. This demonstrates the ease with which complex embedded software systems can be built using the Owl infrastructure.

1 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.

Modern microcontrollers 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 [1, 3, 8, 11]. 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.

This paper presents one mechanism for doing so: an efficient embedded Python run-time system named *Owl*. The Owl system is a complete Python development toolchain and run-time system for microcontrollers that do not have enough resources to run a real operating system, but are still capable of running sophisticated software systems. These microcontrollers typically operate at 50–100 MHz, have 64–128 KB of SRAM, and have up to 512 KB of on-chip flash. One example of such a microcontroller is the ARM Cortex-M3. ARM predicts that in 2015, the market for these Cortex-M class microcontrollers will be around 18 billion units [7]. In contrast, Gartner, Inc. predicts that 404 million x86 processors will ship in 2012 [9].

Owl is a complete system designed for ARM Cortex-M microcontrollers that includes an interactive development environment, a set of profilers, and an interpreter. It 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 (p14p).[1]

We have developed a comprehensive set of profilers and analysis tools for the Owl system. Using the data from these static and dynamic profiling tools, 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. The Owl toolchain we developed includes simple tools to program the microcontroller on Windows, OS X, and Linux. We have also added two native function interfaces, stack protection, autoboxing, a code cache, and many other improvements to the run-time system. Furthermore, the toolchain and run-time system have been re-architected to eliminate the need for dynamic loading.

The Owl system demonstrates that it is possible to develop complex embedded systems software using a high-

[1] `http://code.google.com/p/python-on-a-chip/`

level programming language. Many software applications have been developed within the Owl system, including a GPS tracker, a web server, a read/write FAT32 file system, and an artificial horizon display. Furthermore, Owl is capable of running a soft real-time system, an autonomous RC car. These applications were written entirely in Python by programmers with no prior embedded systems experience, showing that programming microcontrollers with a managed run-time system is not only possible but extremely productive. Additionally, Owl is used as the software platform for Rice University's r-one educational robot [15]. A class of twenty-five first-semester students programmed their robots in Python without the interpreter ever crashing.

The cornerstone of this productivity is the interactive development process. A user can connect to the microcontroller and type statements to be executed immediately. This allows easy experimentation with peripherals and other functionality, making incremental program development for microcontrollers almost trivial. In a traditional development environment, the programmer has to go through a tedious compile/link/flash/run cycle repeatedly as code is written and debugged. Alternatively, in the Owl system a user can try one thing at the interactive prompt and then immediately try something else after simply hitting "return". The cost of experimentation is almost nothing.

Microcontrollers are incredibly hard to program. They have massive peripheral sets that are typically exposed directly to the programmer. As embedded systems continue to proliferate and become more complex, better programming environments are needed. The Owl system demonstrates that a managed run-time system for a high-level language is not only practical to implement for modern microcontrollers, but also makes programming complex embedded applications dramatically easier.

2 Related Work and Background

Early commercial attempts to build embedded run-time systems, such as the BASIC Stamp [12], required multiple chips and have not been used much beyond educational applications. Academic projects have largely focused on extremely small 8-bit devices [10, 13]. 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 [6]. While there were some small proof-of-concept applications developed for it such as a basic web-server [16], the limited computational and I/O capabilities of smartcards rendered building large applications in Java Card impractical [17].

While Android uses an interpreter, Dalvik, that runs on embedded systems, it has very different design goals than these projects [5]. 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.

Arduino[2] is a simple microcontroller platform that has been widely adopted by hobbyists and universities. It has shown that there is great interest in making programming microcontrollers easier. However, Arduino focuses on raising the abstraction level of I/O by providing high-level libraries, while Owl raises the abstraction level of computation with a managed run-time system.

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. The p14p system is a portable Python VM that can run with very few resources and supports a significant fraction of the Python language. Similarly, eLua is a Lua run-time system that runs on ARM microcontrollers.[3] The overall objectives and method of operation of eLua are very similar to p14p, so they will not be repeated here.

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.

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.

In p14p, native C functions can be wrapped in a Python function. This allows arbitrary C functions to be

[2] `http://arduino.cc/`

[3] `http://www.eluaproject.net/`

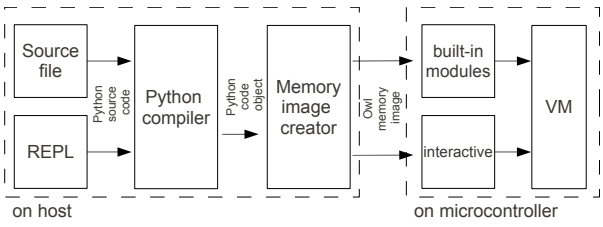

Figure 1: The Owl toolchain.

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.

3 The Owl Toolchain

This section describes the Owl toolchain, as shown in Figure 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 4.

3.1 Code sources

The Owl toolchain supports all of the functionality of p14p. Furthermore, it provides an interactive prompt over USB and drivers for Windows, OS X, and Linux. It also provides several additional capabilities that do not exist in p14p. Most notably, Owl includes a bootloader so that once the virtual machine has been programmed in flash, no C compiler or programming tools are needed by the user to program Python code onto the microcontroller. Directly from the Python interpreter on the host computer, the user can:

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 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.

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 [14]. 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.

To accomplish this, the Owl system architecture 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 references 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.

4 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.

4.1 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.

4.2 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 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

```
/* 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 2: Body of autowrapped native function.

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.

4.2.1 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 2 shows the C code required to wrap a call to the native function with the prototype:

```
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 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 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 [4].

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.

4.2.2 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`[4], 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 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 4.2.1. This tool reads the names

[4] `http://sourceware.org/libffi/`

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.

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.

5 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 because 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. Furthermore, the Owl run-time system also includes a memory analyzer, which would not even be possible for conventional C programs. Additionally, 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.

All of these profilers operate transparently to the user and store all data directly within the Python heap. This works 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.

5.1 Memory Analyzer

A managed run-time system imposes structure on the memory within the system. Everything in the heap is a Python object with an object descriptor that includes its type and size. Furthermore, all data is stored within the Python heap, including stack frames and other global

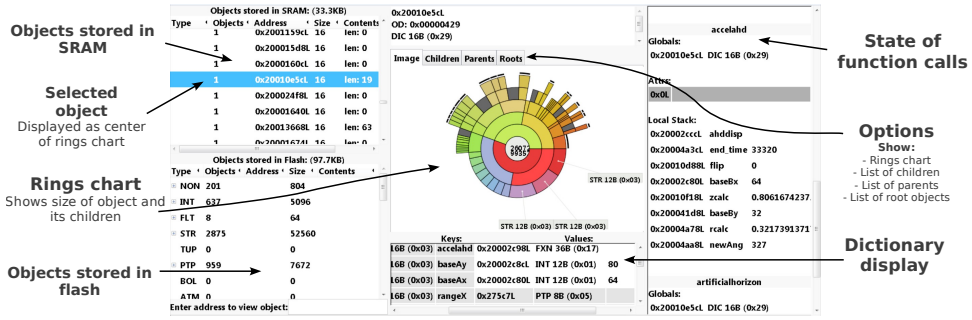

Figure 3: Owl memory analyzer.

and local data. Therefore, it is possible to build tools that can analyze the entire memory space of a Python program. Such tools would actually be impossible to build for C applications, due to the existence of pointers, the fact that data is scattered through the address space, and the lack of a well-defined structure to objects in memory.

The Owl system includes a novel memory analyzer, shown in Figure 3, that provides the programmer with complete access to the entire memory space on the microcontroller. No other system has such a capability. This is accomplished by transferring the entire contents of the Python heap (which is only tens of KB on a Cortex-M3 microcontroller) to the host. The host can then parse the heap and present information about all Python objects that have not yet been garbage collected. Objects in flash that are referenced from the heap can be requested and transferred as well.

The Owl memory analyzer operates by building and traversing a directed graph of the memory objects stored in both the heap and flash. Two tables, one for objects stored in SRAM and one for those in flash, display all objects organized by type. An object can be selected from these tables or looked up by address in order to display additional information about that object, both textually and graphically. Additionally, the memory analyzer can trace the stack frames to show the scope of the objects. These features allow the programmer to easily view how objects are stored, see which objects exist at a given point in the programs execution, and identify costly objects.

5.2 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 5.

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.

5.3 Python and VM Profilers

The three profilers that measure execution time—the Python line number profiler, the call-graph profiler and the VM bytecode profiler—work in fundamentally similar fashions. A hardware timer fires periodically, stopping the interpreter. The profiler determines which part of the user code the interpreter is currently running. This information is then recorded in a set of counters inside the Python heap. In the case of the line number and call-graph profiler, the profiler records the location of execution inside the user's program. In the VM bytecode profiler, the currently executing bytecode is recorded. After the profiler is turned off, the host can read these records and present them in a human-readable format.

The Python line number and call-graph profilers are useful for writing high-performance Python code. However, the VM profiler is extremely useful in determining bottlenecks in the virtual machine itself. For instance, it revealed that a significant portion of the execution time of most programs was spent looking up variable names.

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, as discussed in Section 7.

```
1  for location in self.route:
2    ...
3    atTarget = False
4    while not atTarget:
5      ...
6      if curtime > range_update_time:
7        ... # Read range finder;
8            # get distance to nearest obstacle
9        if dist_to_obstacle < RANGE_MAX:
10         ... # Set motor proportional and servo
11             # inversely proportional to distance
12       ...
13       range_update_time = curtime + RANGE_PERIOD
14
15     if curtime > loc_update_time:
16       ... # Read GPS; get current location,
17           # heading, and distance to destination
18       if dist_to_goal < ERROR_MARGIN:
19         ... # Set motor to min speed
20         atTarget = True
21       else:
22         ... # Set motor proportional to dist_to_goal
23       loc_update_time = curtime + LOC_PERIOD
24
25     if curtime > heading_update_time:
26       ... # Calculate degrees to turn (deg_to_turn)
27       heading_update_time = curtime + HEADING_PERIOD
28
29     if curtime > gyro_update_time:
30       ... # Read gyro; update steering and integral
31       if dist_to_obstacle == RANGE_MAX:
32         ... # Set steering to gyro's recommendation
33       gyro_update_time = curtime + GYRO_PERIOD
34
35 ... # Stop the car when it reaches the final location
```

Figure 4: Python event loop for autonomous RC car.

6 Applications

This paper demonstrates the Owl system on the Texas Instruments Stellaris LM3S9B92 (9B92), an ARM Cortex-M3 microcontroller that operates at up to 80 MHz, has 96 KB of SRAM, and has 256 KB of flash. In the experiments, the 9B92 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), and an autonomous RC car (using the gyroscope, GPS, range finder, steering servo, and motor controller).

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 StellarisWare 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 5 were used to measure and improve performance.

6.1 Autonomous RC Car

The autonomous RC car demonstrates the capability, flexibility, and ease of use of the Owl system. The electronics from an off-the-shelf RC car (the Exceed RC Electric SunFire Off-Road Buggy) were replaced with a 9B92 microcontroller and associated peripherals. The car is controlled entirely by the microcontroller.

The code skeleton in Figure 4 shows the main event loop running on the car to implement a feedback controller that performs GPS-based path-following with obstacle avoidance. 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.

First, if the range finder detects an obstacle in lines 7–8 of the code, the controller translates the distance to that obstacle into a proportional motor speed and an inversely proportional steering servo setting. Second, the controller uses the GPS location to calculate the distance to its destination in lines 16–17. If the car has reached its destination, the controller slows the motor and updates to the next destination. Otherwise, it scales the motor's speed proportionally to the distance and calculates a goal heading. Finally, the a proportional-integral controller steers the car, using the gyro's feedback to control the rate of turn and the GPS' heading calculation to determine the degrees to turn.

The range finder can update every 49 ms, the gyro every 10 ms, and the GPS every second; the controller utilizes these sensors fully by requesting updates every 60, 30, and 1000 ms, respectively.

6.2 Microbenchmarks

A small subset of the Computer Language Shootout[5] 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.

`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

[5] `http://dada.perl.it/shootout/craps.html`
`http://shootout.alioth.debian.org/`

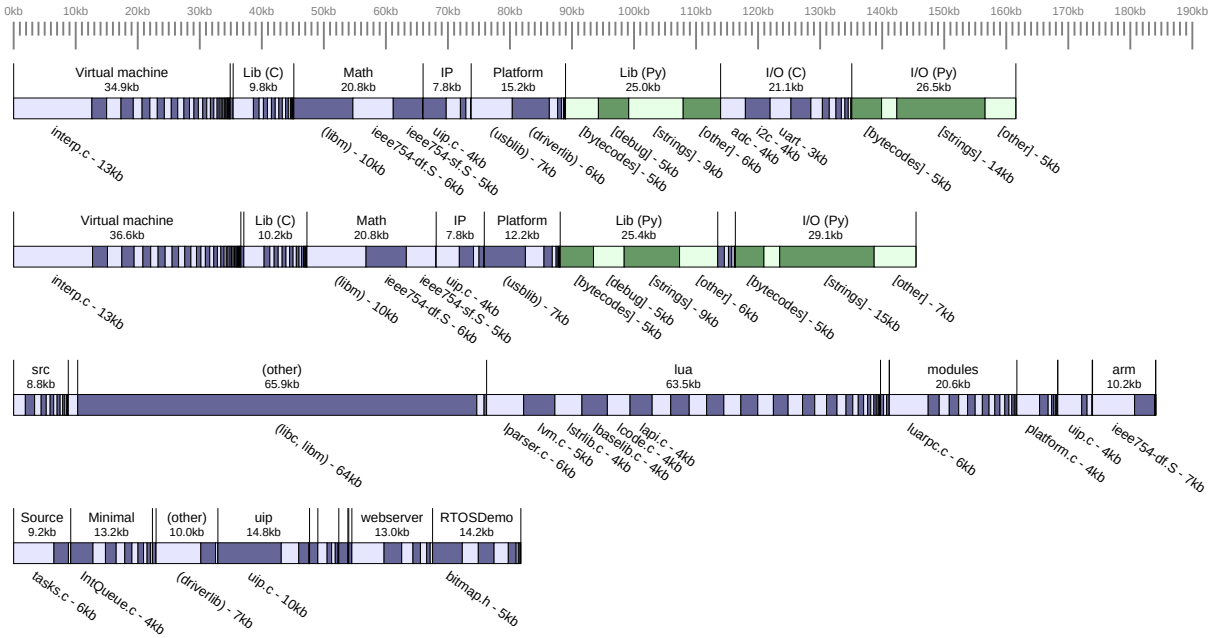

Figure 5: Static binary analysis, from top to bottom, of the Owl virtual machine using wrapped functions, using the foreign function interface, of the eLua virtual machine and of an example program using SafeRTOS.

a floating-point benchmark that simulates the motion of the Sun and the four planets of the outer solar system.

7 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 runtime performance. Finally, we show that using a loaderless architecture uses four times less SRAM than a traditional system.

7.1 Static binary analysis

Figure 5 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 runtime 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

	Bytecode/Benchmark	artificialhorizon	gps-tracker	car_range	car_gyro	car_range_gyro	car_gps_gyro	car_gps_gyro_range	ackermann	heapsort	matrix	nbody	Running time
Memory	BINARY_SUBSCR	1%	-	-	-	-	-	-	-	9%	11%	4%	7 μs
	LOAD_ATTR	23%	10%	7%	18%	14%	18%	21%	-	5%	-	-	64 μs
	LOAD_CONST	3%	1%	-	-	-	-	-	11%	5%	1%	8%	8 μs
	LOAD_FAST	1%	1%	1%	-	3%	3%	3%	12%	18%	12%	14%	4 μs
	LOAD_GLOBAL	24%	21%	41%	24%	68%	64%	59%	16%	5%	1%	-	75 μs
	STORE_FAST	-	-	-	-	-	-	-	-	2%	3%	6%	4 μs
	STORE_SUBSCR	-	-	-	-	-	-	-	-	2%	-	3%	6 μs
Math	BINARY_ADD	-	-	-	-	-	-	-	3%	2%	-	2%	10 μs
	BINARY_MULTIPLY	-	-	-	-	-	-	-	-	-	3%	16%	11 μs
	BINARY_POWER	-	-	-	-	-	-	-	-	-	-	11%	116 μs
	BINARY_SUBTRACT	-	-	-	-	-	-	-	6%	-	-	3%	10 μs
	INPLACE_ADD	-	-	-	-	-	-	-	-	3%	3%	5%	10 μs
	INPLACE_SUBTRACT	-	-	-	-	-	-	-	-	-	-	3%	11 μs
Control Flow	CALL_FUNCTION	33%	60%	44%	41%	3%	4%	5%	33%	3%	4%	-	148 μs
	COMPARE_OP	4%	-	1%	-	2%	1%	1%	-	7%	-	-	13 μs
	DUP_TOPX	-	-	-	-	-	-	-	-	-	-	3%	4 μs
	FOR_ITER	-	-	-	-	-	-	-	-	-	5%	1%	6 μs
	JUMP_ABSOLUTE	-	-	-	-	1%	1%	-	-	2%	3%	1%	7 μs
	JUMP_IF_FALSE	-	-	1%	-	2%	2%	2%	-	5%	-	-	5 μs
	JUMP_IF_TRUE	-	-	-	-	-	-	-	6%	-	-	-	5 μs
	POP_BLOCK	-	-	-	-	-	-	-	-	-	2%	-	137 μs
	POP_TOP	-	-	-	-	1%	1%	1%	3%	2%	-	-	2 μs
	RETURN_VALUE	-	-	-	-	-	-	-	4%	1%	2%	-	74 μs
	ROT_THREE	-	-	-	-	-	-	-	-	-	-	2%	3 μs
	SETUP_LOOP	-	-	-	-	-	-	-	-	-	1%	-	15 μs
	UNPACK_SEQUENCE	-	-	-	-	-	-	-	-	-	-	7%	12 μs
	(garbage collector)	1%	-	-	-	-	-	11%	-	17%	40%	3%	3-65 ms

Table 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.)

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.

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.

7.2 Performance

This section presents the performance of the Owl runtime system using the profilers discussed in Section 5.3. Table 1 shows the profiling results of the benchmarks and applications described in Section 6. Each column (before the last column) shows one workload. 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 contributer 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 LOAD_GLOBAL bytecode loads objects (including functions) from the global namespace, and is particularly slow due to the large size of this namespace.

	Lookups	Hit Rate	Search len	Avg size
LOAD_GLOBAL	129232	0.88	25	41.2
LOAD_ATTR	174374	0.71	10.4	17.0

Table 2: Profiler results showing how dictionaries are used by the interpreter.

Specifically, in the artificial horizon workload, nearly half of the running time is spent in the LOAD_ATTR and LOAD_GLOBAL bytecodes. Table 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 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 `max()` or `int()`. In other words, the interpreter may have to search multiple dictionaries per name lookup. However, these dictionaries are reasonably small. As Table 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 [2].

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 (BINARY_ADD, etc.) are only significant for the CPU benchmarks. The embedded workloads are dominated by the bytecodes that perform control flow (JUMP_*, COMPARE_OP, etc.).

	Type	Object count	Avg size (bytes)	Total size (bytes)	Fraction of total heap
primitives	None	2	8.0	16	0%
	int	180	12.2	2188	7%
	float	3	12.0	36	0%
	string	29	19.9	576	2%
	bool	2	12.0	24	0%
sequences	tuple	47	15.2	716	2%
	packed tuple	5	8.0	40	0%
	set	1	12.0	12	0%
	seglist	92	16.5	1520	5%
	segment	251	40.0	10048	32%
	list	7	12.0	84	0%
	dict	200	16.3	3252	10%
	xrange	0	0.0	0	0%
OOP	module	22	36.4	800	3%
	class	14	12.0	168	1%
	function	317	36.1	11440	36%
	instance	2	12.0	24	0%
internal	code obj	2	44.0	88	0%
	packed cobj	1	12.0	12	0%
	thread	1	36.0	36	0%
	method	0	0.0	0	0%
	frame	5	94.4	472	1%
	block	2	20.0	40	0%
	(all)	1185	26.7	31592	100%

Table 3: A snapshot of the heap, broken down by object type, for the `artificialhorizon` workload.

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.

7.3 Memory use

Table 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 9B92 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 `segment` object is a portions of a list, and a `function` object points to a code object and the variables in its scope. In contrast, the heapsort bench-

mark 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. 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 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.

8 Conclusions

This paper has presented the design and implementation of Owl, a powerful and robust embedded Python run-time system. The Owl system demonstrates that it is both possible and practical to build an efficient, embedded run-time system for modern microcontrollers. Furthermore, this paper has shown that it is straightforward to implement complex embedded control software in Python on top of such a run-time system.

This paper has also illustrated several key points about embedded run-time systems through careful analysis of Owl. First, a large fraction of the binary consists of support libraries that would also need to be included in a native C executable. 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.

Traditionally, microcontrollers are programmed with hand-coded C or auto-generated code from MATLAB. While this paper has presented many of the advantages of a managed run-time system for microcontrollers, it does not present a comparison to more traditional systems. Such comparisons are an important direction for future work to further quantify the trade-offs of embedded managed run-time systems.

There are orders of magnitude more embedded microcontrollers in the world than conventional microprocessors, yet they are much harder to program. This persists because software development for embedded microcontrollers is mired in decades old technology. As a result, there has been a proliferation of low-level embedded software that is difficult to write, difficult to test, and difficult to port to new systems. The Owl system helps to improve this situation by enabling interactive software development in a high-level language on embedded microcontrollers. We believe this will lead to enormous productivity gains that cannot be overstated.

Acknowledgments

We would like to thank Kathleen Foster and Laura Weber for their contributions to the system and applications, Dean Hall for developing p14p, Texas Instruments for hardware donations, and the anonymous reviewers and our shepherd for their advice on the paper.

References

[1] ANH, T. N. B., AND TAN, S.-L. Real-time operating systems for small microcontrollers. *IEEE Micro 29*, 5 (2009).

[2] BACON, D. F., CHENG, P., AND RAJAN, V. T. The metronome: A simpler approach to garbage collection in real-time systems. In *P. 1st JTRES* (2003).

[3] BAYNES, K., COLLINS, C., FILTERMAN, E., ET AL. The performance and energy consumption of embedded real-time operating systems. *IEEE Trans. Computers 52*, 11 (2003).

[4] BEAZLEY, D. M., FLETCHER, D., AND DUMONT, D. Perl extension building with SWIG. In *O'Reilly Perl Conference 2.0* (1998).

[5] BORNSTEIN, D. Dalvik VM internals. In *Google I/O* (2008).

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

[7] DATABEANS. Smartphones boost microcontroller shipments... `http://www.mtemag.com/ArticleItem.aspx?Cont_Title=Smartphones+boost+microcontroller+shipments+with+Arm+seeing+major+growth`, March 2, 2011.

[8] GANNSSLE, J. The challenges of real-time programming. *Embedded System Programming Magazine* (2007).

[9] GARTNER, INC. Gartner says pc shipments to slow... `http://www.gartner.com/it/page.jsp?id=1786014`, September 8, 2011.

[10] HILL, J., SZEWCZYK, R., WOO, A., ET AL. System architecture directions for networked sensors. In *P. 9th ASPLOS* (2000).

[11] KALINSKY, D. Basic concepts of real-time operating systems. *LinuxDevices Magazine* (2003).

[12] KUHNEL, C., AND ZAHNERT, K. *BASIC Stamp: An Introduction to Microcontrollers*. Newnes, Woburn, MA, USA, 2000.

[13] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *P. 10th ASPLOS* (2002).

[14] LIANG, S., AND BRACHA, G. Dynamic class loading in the Java virtual machine. In *P. 13th OOPSLA* (1998).

[15] MCLURKIN, J., LYNCH, A. J., RIXNER, S., ET AL. A low-cost multi-robot system for research, teaching, and outreach. In *10th DARS* (2010).

[16] REES, J., AND HONEYMAN, P. Webcard: a Java card web server. In *P. 4th CARDIS* (2001).

[17] RIPPERT, C., AND HAGIMONT, D. An evaluation of the Java card environment. In *P. Middleware for Mobile Computing* (2001).