Porting Lua to the XINU Operating System

Theodore Sudol*
Mentor: Dr. Dennis Brylow

Abstract

Adding a scripting language to the Embedded XINU Operating System enables real-time modification of the OS and potentially improves development processes through faster development times and easier debugging. The Lua scripting language was chosen for its portability, small size, fast execution time and expressive power. Porting Lua to XINU involved implementing new sections of the XINU C standard library and configuring Lua for the limitations of the embedded platform. As a result, core Lua functionality is usable in XINU C programs. However, the Lua standard libraries are not yet functional, preventing full use of Lua's capabilities. Future work, such as integrating the Lua interpreter with the XINU shell, can build on this foundational project to further enhance XINU.

1 Introduction

The Embedded XINU operating system is an "academic" operating system: Designed for use in college-level courses, it is smaller and less featured than a commercial operating system. This makes it easier for students to learn and understand the concepts presented with XINU and provides ample opportunities for professors to assign the implementation of new features as coursework. Embedded XINU, given its primary deployment on Linksys WRT54GL routers, is particularly suited for Operating Systems, Networking and Programming Language courses, and it is also used for embedded systems research [MSCS 2011].

XINU is developed using the C programming language and MIPS Assembly language. All development or research for this operating system follows a similar work flow: First, the new feature is designed and programmed on a normal computer. It is then integrated with the standing XINU code. Because XINU is deployed on Linksys routers, a cross-compiler is used to compile and link the code. Unlike most compilers, which turn high level language code (such as C programs) into the machine code of whatever computer they are running on, cross-compilers are designed to transform the high-level code into machine code for a different platform. Once XINU has been compiled with the new feature in place, the operating system is deployed onto one of the Linksys routers. The new feature can then be tested; the tests would have been written at the same time as the feature so they too could be compiled into the operating system kernel. If the new feature fails its tests or new test cases are thought of, the system must be taken offline and the work flow restarts. This process leads to frustratingly slow development as compilation and deployment consume more and more time. The ability to prototype and test features dynamically while the system is deployed could shift time back to simply developing features and tests. A solution to this problem is the implementation of a scripting language for XINU.

A script is a small program used to automate or simplify tasks. The languages used to write scripts tend to be interpreted instead of compiled. That is, instead of compiling into an executable file, a program called an interpreter executes the instructions in a source code file on the fly. Because there is no need for recompilation, development is can proceed directly from programming to testing. Consider how this could change the XINU work flow. To begin developing a new feature, the developer launches the interpreter on the deployed XINU system. Programming and testing the feature then occurs in the interpreter until the developer is satisfied, at which point the new feature can be solidified with either a C program or as a script. This way, the number of iterations of the development cycle can be reduced, and proofs-of-concepts and prototypes for either C or the scripting language can be created. Due to the scripting language, the problem of the slow development process is reduced if not outright resolved.

Of course, this is not the only advantage a scripting language can bring. What may take several dozen lines of boilerplate code in C may be only a few lines in a scripting language. Instead of only being able to use the imperative

*This work was supported by the National Science Foundation under grant #CNS-1063041.

or procedural programming style of C, object-oriented or even functional programming paradigms become available; such languages are called "multi-paradigm". Scripting languages help produce solutions that may be simpler or more elegant than those created with other programming languages.

Based on these considerations, I chose to port the Lua scripting language to Embedded XINU. Lua is a lightweight, multi-paradigm language designed for embedding with other languages [Ierusalimschy 2003]. Thanks to this design goal, it has a small memory footprint—a full implementation uses less than 250 KB of memory—and has an excellent system for embedding and extending with C [Lua.org 2012]. These two features mesh well with XINU, which is written in C and, as an embedded operating system, must be usable on systems where memory is a limited resource. Also, Lua is highly portable thanks to a large number of configurable features and can be compiled on any platform with a full C standard library. A more detailed discussion of Lua and its features can be found in section 2.1. Lua is a small and powerful language that works well in the limitations set by XINU and the embedded systems.

The goal of this project was to integrate the Lua interpreter, which uses a register-based virtual machine to execute scripts, with the Embedded XINU shell. This would allow the use of Lua as a stand-alone language and as an embedded tool in C programs. Porting Lua involved configuring the language and resolving incompatibilities between the language and XINU's implementation of the C standard library, which is discussed in section 2.2.

1.1 Related Work

The eLua (or "embedded Lua") project aims to bring Lua to a wide variety of embedded systems [Marinescu and Sutter 2011]. eLua runs on the "bare metal" of a system with no operating system. In fact, the project is organized by the different processors that eLua can run on instead of by the operating systems that can use it. eLua allows a Lua script (with possible C integration) to work on any of the supported systems without modification. However, there is no implementation of eLua for the Linksys WRT54GL routers, which is the primary platform for Embedded XINU.

2 Methods

2.1 The Lua Scripting Language

As mentioned in section 1, Lua is a lightweight, multi-paradigm scripting language with a history of use in embedded programs. It was originally developed in 1993 by the Computer Graphics Technology Group at the Pontifical Catholic University of Rio de Janero, Brazil, and it is currently maintained by the Lablua group at the same institution [Lua.org 2012]. In the following years, Lua found use in the game development industry where it is used in conjunction with a low-level game engine (which controls things like in-game physics, sound and graphics) to create a full game. Beyond its use with embedding and extending, Lua is small, fast and portable.

An embedded language is generally used in other programs to provide additional functionality. For example, Lua is designed for embedding in C programs. It capitalizes on this by providing tools that cover C's weaknesses:

> "Lua is a tiny and simple language, partly because it does not try to do what C is already good for, such as sheer performance, low-level operations, or interface with third-party software. Lua relies on C for those tasks. What Lua does offer is what C is not good for: a good distance from the hardware, dynamic structures, no redundancies, ease of testing and debugging. For that, Lua has a safe environment, automatic memory management, and great facility to handle strings and other kinds of data with dynamic size" [Ierusalimschy 2003].

To further this end, Lua has a concise and consistent C API that is easy to get started with and use. The central idea is a Lua state, which is similar in concept to a thread of execution and all the data associated with it. The Lua state maintains a stack that is used to push and pop values and arguments to functions in the program. Using the stack and the Lua state is straightforward, especially since the documentation clearly shows how each function in the API interacts with the stack. By design, Lua is one of the simplest solutions when it comes to embedding.

Another advantage to Lua is its extensibility. That is, it is easy to use third-party libraries—even those written in other languages like C, C++ and more—to add new functionality to Lua. For example, a function that is slow in Lua and causes a bottleneck in the script's execution time can be rewritten in C to remove that bottleneck and speed up the script as a whole. Extending Lua turns it from a small scripting tool into a language that can be used in any situation.

The size of a full Lua implementation was another factor in choosing it for this project. An embedded platform has a limited amount of memory available, especially when compared to the resources available to a normal workstation

PC. Fortunately, Lua is very small: At only 20,000 lines of ISO C source code, the compiled library (which is the core of the language) is only 243 KB. The interpreter itself, which is used to execute stand-alone Lua scripts, is another 182 KB when compiled with all of the Lua standard libraries [Lua.org 2012]. Lua's memory footprint is tiny, even when considered in the context of an embedded system.

Lua also performs well with another important programming language metric: execution speed. The Computer Language Benchmarks Game, a standardized suite of benchmark programs for various languages, shows just how fast Lua can be [Fulgham 2012]. The benchmarks are ten programs that can be implemented in many different languages. For each language, the average execution speed for the ten benchmarks are divided by the fastest execution time of all the languages[1] to find that particular language's median execution time. The C programming language, for example, has a median execution time of 1.30. In comparison, Lua has a median execution time of 28.53. Comparisons with the other scripting languages that have been benchmarked—such as Ruby 1.9, PHP, Python 3 and Perl—show that Lua is the fastest scripting language[2]. Its reputation of speed is well deserved.

Possibly the most important consideration is how portable Lua is. With no modifications, Lua can be compiled on any platform with a full C implementation [Lua.org 2012]. Even better, Lua is OS independent; it has no features that rely exclusively on one operating system. These two traits have brought Lua onto a wide variety of platforms: "Lua runs on all flavors of Unix, Windows, mobile devices (running Android, iOS, BREW, Symbian, Windows Phone), embedded microprocessors (such as ARM and Rabbit, for applications like Lego MindStorms), [and] IBM mainframes" [Lua.org 2012]. The eLua project mentioned in section 1.1 shows the wide variety of platforms that Lua can be used on. The portability of Lua is obvious and well-established.

A full examination of Lua's features is beyond the scope of this paper and would be better served by referring to the *Programming in Lua* book[3] or the Lua 5.2 Manual[4]. Instead, I will discuss two of the most important features of Lua: the basic data types and the standard library.

2.1.1 Lua Data Types

There are eight basic data types in Lua: nil, Boolean, number, string, function, userdata, thread and table. The first four are straightforward:

nil The `nil` type represents the absence of value. If a variable is not set to any value when created, it will be set to `nil`.

Boolean Boolean variables are either `true` or `false`. They are used in conditionals or as flag values. `nil` is also considered false, and any value other than `nil` or `false` is true. The number zero, the empty string (`"`") and the empty table (`{}`) in particular are treated as true values.

number Lua numbers are usually double-precision floating point type numbers that can represent any real number. Lua does not differentiate between integers and floating-point numbers as other languages do, and instead just uses one data type to handle both. Because the underlying data type for numbers can be configured when Lua is compiled, few assumptions about Lua numbers can be made. However, most implementations do use the `double` type.

string A string is a sequence of characters. There is no special data type to represent just a single character, so `"a"` and `'a'` are equivalent, as are `"Hello"` and `'Hello'`.

The remaining data types require a little more depth.

Functions Functions are blocks of callable code that can be passed different values as arguments, like functions in any other language. Lua treats functions as first-class values. That is, any function can be stored in a variable, and this is actually the behavior the Lua exhibits behind the scenes when a function is declared. Functions are created using the `function` keyword:

[1] The language with the fastest average execution time is Intel Fortran.

[2] The next closest is JRuby, an implementation of the Ruby language for the Java Virtual Machine, with a median execution time of 34.52.

[3] Available for Lua 5.0 for free at www.lua.org/pil/#1ed. The most up-to-date edition, which was written for Lua 5.2, is available through Amazon.

[4] Available for free at www.lua.org/manual/5.2/. Note that there are several differences between the last version, 5.1.4, and 5.2. The manual does list the changes between versions.

```
function hello()
    print("hello!")
end
```

This is actually equivalent to:

```
hello = function()
    print("hello!")
end
```

Both notations are completely valid in Lua. Because they can be stored in variables, functions can be passed around in the program like any other variable. This allows for more complex scripts that can use functions in a wide variety of ways. For example, the `map` function, which applies a function (passed as an argument) to a list of values, is almost trivial in Lua. Anonymous functions can also be created by using the second function syntax mentioned above. These are primarily used to pass small functions as arguments to eliminate the need of creating a whole new function that will only be used for that one call. Because of their first-class nature, functions in Lua are very useful and powerful.

Userdata The userdata type is an interesting example of the Lua C API. Userdata are used when embedding Lua in C to create more complex data types. They are essentially raw blocks of memory that can be configured and manipulated by the programmer. Using a Lua construct called metatables, userdata can be given behaviors for common operations such as addition or multiplication. The data found inside the userdata can be defined from a C struct. Userdata can be thought of as a more powerful version of the normal C struct.

Threads Threads are the data type used to represent Lua's coroutines. Coroutines (also called "cooperative multithreading") are independent threads of execution. While they are somewhat similar, coroutines should not be confused with operating system threads. Coroutines can be used on any platform—even if that platform does not support multithreading. Coroutines should be thought of as a way to organize the control flow of a program and separate different conceptual processes and not as a multithreading tool.

Tables Lua comes with one built-in data type, the table. Tables are collections of key-value pairs, where any data can be a key or a value. Because of this, tables are a flexible data type that can be treated as any other complex data type. For example, a normal array is simply a table with numeric keys (or indices, as they would be called in this case) while a dictionary structure would use strings as keys. The types of the keys do not need to all be the same, so any of the data types listed above (including tables) can be used together in one table. A very simple object-oriented programming (OOP) scheme can be implemented by taking advantage of this fact.

First, a table that represents the full class is created. Any class-level variables are stored in this table using strings as keys. Thanks to syntactic sugar, `someTable["stringKey"] = someValue` can be written as `someTable.stringKey = someValue`, which looks exactly like field access in a C++ or Java class. Any class methods can also be stored in this table by using the `function someTable.someFunction()` idiom, which stores the function as a key in the table. The constructor for new variables of this class is created this way. The constructor would create a table as a local variable, assign any instance variables with values are passed in as arguments. Instance methods are added to the table using the same syntax as the class methods. Once the initialization is complete, the constructor simply returns the table it defined, and thus a new instance of the class is created. This OOP scheme is straightforward and easy to implement where needed.

Lua tables also have a feature called metatables that can be used to formalize more complex data types. Metatables define a table's behavior when used in certain operations. For example, all of the mathematical operators like addition or multiplication can be assigned special behavior for a specific table. When coupled with the above OOP scheme, metatables resemble operator overloading from C++. For example, a Set class that represents mathematical sets (collections of unique numbers) can have union and intersection operations defined by using the addition and multiplication operators. The metatable uses a series of keys called metatable events that are checked whenever a particular operation is performed on a table. If that operation is defined in that table's metatable, that definition is

used instead of the default. Like normal operator overloading, metatables allow for more natural and powerful object interactions.

A few last notes on tables: All tables are indexed from 1 when using numeric indices. While 0 and negative indices can be used, they will be ignored by the table manipulation functions and will not be considered when calculating the length of the table. Furthermore, there is no bounds checking for tables. This means, for example, if a table has only indices 1 through 3 assigned and the program accesses index 4, Lua will not throw an error or otherwise complain. This is actually less of a problem than it may initially seem. Any key that does not explicitly have a value assigned to it will return `nil` when accessed. As long as the programmer is aware of this behavior, it can be used very effectively in Lua scripts.

2.1.2 The Lua Standard Library

There are ten standard libraries in Lua. They range from string and table manipulation to bitwise operations and coroutine creation. These libraries are presented as tables of functions; when used in a Lua script, they can be called using the `tableName.functionName()` syntax. They are automatically loaded in the Lua interpreter and can be accessed when embedding Lua with the C API. The ten libraries are:

Base Library The base library is something of a "catch-all". The basic functions that will likely be used in all Lua scripts are found here, and none of them could be easily shoehorned into one of the other libraries. The library provides functions for the garbage collector, type checking and conversion, basic terminal output, exception handling and assertions, table iteration and metatable access.

Coroutine Library This library is used to create, use and close coroutines. Coroutines are created by developing a function that yields values at certain points. The yield function is provided by the coroutine library, too. The function is wrapped inside a coroutine, which, when run, executes its function until a yield statement is encountered. At that point it pauses execution, and the yield statement returns a value similar to a return statement. When the coroutine is told to run again, it resumes execution from the last yield statement and runs until it encounters another yield. The coroutine library provides all the necessary functions for using coroutines.

Package Library Packages are third-party modules or other external code that can be used in a Lua script. This library handles finding and loading these modules. It also keeps track of all packages loaded.

String Library This library provides string manipulation. Besides common functions that find the length of a string, change the case of its characters, or find substrings, the string library provides functions for pattern matching, conversion of characters to their equivalent byte representations and vice versa, and string formatting similar to the `printf` functions in C. Lua's pattern matching is a light version of regular expressions, which can be used to find groups of characters in a given string. Pattern matching can be used to extract substrings or replace characters in a string. Unlike the other standard libraries, the string functions can be called on strings themselves. That is, instead of using the `string.functionName(someString, args)` format, the function could be called by using `someString:functionName(args)`. The colon function call syntax passes the caller of the function as the first argument of the called function; this is useful for both strings and the object-oriented programming mentioned in section 2.1.1. This syntax is more expressive and natural when using strings and other objects.

Table Library Even though tables can be used in a myriad of ways, the table library is best used for arrays and lists. For instance, the functions used to add or remove elements from a given table only do so with numeric indices. The other functions expect a table where indices 1 to n are assigned values in sequence. Any non-numeric keys or indices below 1 are ignored, and the first index with a `nil` value is considered the end of the table. The two functions used for table iteration have been adapted to this. One function iterates over a table and returns only the numeric indices and their values starting at 1 and proceeding in sequence. The other functions returns any and all key-value pairs in the table, which may not have an established order. These two functions allow any configuration of table keys and values to be used in loops for easier processing.

Mathematical Library As Lua only has the basic arithmetic operators by default, it needs a library to provide the many mathematical functions that are commonly used in programs. The math library provides functions for trigonometry, logarithms, exponentiation, basic ceiling and floor rounding, and random number generation. Some mathematical constants are also provided, such as pi and "huge". Huge is "a value larger than or equal any other numerical value" [Ierusalimschy et al. 2006]. It is essentially a representation of infinity.

Bitwise Library The bitwise operators found in C are not used in Lua. Instead, they are implemented as functions in this library. Some examples are bitwise AND and OR, bit shifting, and rotating. The library is referred to as `bit32`, which indicates that it is designed for 32-bit integers. All input is transformed from Lua's standard number type to 32-bit unsigned integers automatically.

I/O Library While the base library provides simple output, it is lacking in file and stream operations. The I/O library provides all of these functions, which can be used to open files, read input from a file or stream, and write output to the same. It has a few more options than the equivalent C library. For example, it is possible to read one line (ending with a newline character) from a file at a time with this library. Temporary files are also made available.

OS Library This library provides an interface to the underlying operating system. It is primarily used for date and time operations, though it does have functions for executing shell commands, renaming or removing files, exiting a script early and accessing the computer's environment settings.

Debug Library The debug library is designed for exactly what its name suggests: Debugging a Lua script. It can access the internal workings of a script and the interpreter, providing valuable information for finding serious problems. However, it can be somewhat dangerous; it purposefully violates certain rules set by the Lua language standard. It is a very advanced tool that should only be used for debugging and must be used carefully.

This overview of Lua and its standard libraries sheds some light on the power the language brings to the XINU platform. Lua is a simple language with powerful tools that can be adapted to any situation.

2.2 Porting Lua

While Lua is OS independent and relies only on a full C implementation, as mentioned in section 2.1, actually porting it to the Embedded XINU operating system was not a straightforward task. The process was split into two phases: configuring Lua for the new platform and resolving any dependencies.

2.2.1 Configuration

The configuration of Lua is centralized in a simple C header file called `luaconf.h`. This configuration file defines a large number of constants and functions that are used throughout Lua. Some of them are for platform- or OS-specific implementations while others control compatibility with older versions of Lua. The most important definitions are those for the basic data types that Lua uses. For example, Lua uses the `luanumber` type to refer to all numbers. In practice, `luanumber` is just another name for the common `double` floating-point type. Because the Linksys routers that XINU is deployed on do not have a floating-point processor, `luanumber` was defined as an `int` (integer) type instead. This was as simple as changing one line in the header file and making sure that two basic functions used for type-casting numbers wouldn't break. Fortunately, `luaconf.h` is logically laid out, has basic markers to aid searching, and is very well commented, so finding these little side effects was a simple matter. Configuring Lua was a simple and straightforward matter that was made easier by Lua's design.

2.2.2 Dependencies

As previously mentioned, Lua requires a C compiler with a full implementation of the language. Unfortunately, XINU does not have a full implementation of C. Instead of just configuring and compiling, the missing features that Lua required had to be found and reimplemented as part of the XINU C standard library.

First, the missing dependencies had to be discovered. Simply running the Lua library through the XINU C compiler and keeping track of the errors was not a feasible solution. Instead, the linker's cross-reference table, which it writes in order to keep track of the symbols needed by a program, was used to find the functions and definitions that were included from libraries external to Lua. The cross-reference table was accessed by passing the `--cref` option to the linker at compile time. To find the complete list of dependencies, Lua was compiled on a Redhat Linux system, as this would compile correctly and find all of the dependencies. The linker produced a table like so:

fopen	/lib64/libc.so.6
	liblua.a(loadlib.o)
	liblua.a(lauxlib.o)

The first entry of this table is the name of the symbol, which in this case is the `fopen` function from the `stdio.h` C header. The right-hand column shows where the definition of the symbol can be found (`libc.so.6`) and where `fopen` is used in the Lua source code (`loadlib` and `lauxlib`). (Not every entry in the cross-reference table followed this format. Many of the entries were just lists of functions that were not used in the Lua code.) A simple Lua script extracted the functions used by Lua along with where in the Lua source code they are used. This script created another list that showed which functions each Lua source file needed. The list of needed functions was determined by comparing the list created by the scripts with the XINU standard library.

The second step was to actually implement these functions. Further analysis of the list revealed that not all of the functions needed could be implemented by XINU. For example, all of the math functions that required floating point numbers, such as the trigonometry functions, could not be implemented because of the lack of floating-point processors in the Linksys routers. Furthermore, it was pointless to implement some of the other functions, such as the error number tracking headers. They provided little additional functionality to Lua and were easily removed. In the process of removing these dependencies, some of Lua's standard libraries were heavily reduced, particularly the OS and math libraries. Those two libraries had most of the functions that XINU simply could not support. That left only the dependencies that were actually important to Lua.

Implementing most of these dependencies was not difficult. Most of the functions were well documented enough that the basic mechanism for them could be determined just by reading the documentation. The most difficult exception to this was the `setjmp.h` header. `Setjmp` and `longjmp` are a pair of functions that store the state of the processor—represented by its registers—in an array. `Longjmp` is used to reset the processor back to the state that was stored in the array by `setjmp`. This particular header is not widely used, and most documentation for it focuses on how to use it without many clues to how it actually works. After much research, the solution of using a pointer to an array of integers to store the processor state was eventually found. Fortunately, most of the dependencies were easier to resolve than this, and Lua was soon compiling on XINU.

2.3 Testing

Several test cases were written for this project. As mentioned in section 2.2.2, the functions written to fulfill missing pieces of the XINU C library each had a test case written for them. Once the full Lua library was compiled, a test case specifically for embedding Lua in C was written. The test case creates a Lua state, performs some arithmetic and string operations with it, and closes it. While this undoubtedly sounds straightforward, actually getting the test case to work was not.

The first major problem was found in `lua_newstate`, the function that actually creates a new Lua state. It was eventually traced to a function that initializes memory-sensitive segments of the state. If that function does not return an OK signal, the state is closed prematurely and the test case fails. To protect against faults that would prevent the state from being closed correctly, the aforementioned `setjmp` facilities were used. The array used to store the processor state, called a jump buffer, was stored in a struct along with the status code the function would return. If the status code there was an OK signal (represented by the integer value 0), the memory-sensitive function would also return an OK signal. However, the function actually returned a large negative value when tested, such as $-2,147,153,304$. Some investigation eventually revealed that this value when translated to hexadecimal was the exact memory location of the function call to the memory-sensitive function. Further testing was carried out by rearranging the source code, which changed the position in memory of that function call and revealed that the large negative error code was in fact the memory location of the function call. Further investigation revealed that the implementation of `setjmp` was faulty. There are 29 processor registers that needed to be saved, and the implementation stored them in an array of 29 integers. However, the values were stored in array locations 1 to 29 instead of 0 to 28—that is, the last register was written to a memory location outside of the array. The last register was the return address register and the memory location it was written to was the memory location for the status code! When the processor state was stored, the return address was written into the status code, so the status code became $-2,147,153,304$. The test case for `setjmp.h` never indicated that this problem could occur. When `setjmp` and `longjmp` were rewritten to correctly store the registers, the problem of the mysterious status code disappeared and the Lua state initialized correctly.

The second major problem was found in the Lua string table. The string table is used by the Lua state to keep track of all strings used in the program. Since the number of strings can change wildly, the string table is initialized to 32 entries and is doubled in size whenever the number of strings approaches the current size of the table. This can happen fairly quickly because Lua initializes the string table with several strings that it always uses during execution. In the XINU implementation of Lua, the table resize function, called `luaS_resize`, was the common point of failure

for the several iterations of the test case. The error was consistent in that it was always the same instruction that caused the router to crash. It was eventually found that entries in the string table were invalid pointers. This situation could only arise due to a faulty initialization method for the entries or memory corruption at some point after initialization. Interestingly, the problem never appeared when the test case was run as a part of the full XINU test suite, and there were several cases where the individual test case for Lua would run correctly after the full test suite was run. Despite several attempts to diagnose the problem, the source of this string table error has not been found. The current solution is a series of conditionals that check if the string table entry, which is a pointer to a struct, and its constituent parts are within the valid bounds of the XINU heap. If the entry is found to be invalid, it is skipped and the next entry is used. This solution, while crude, catches most of the faulty table entries and prevents XINU from crashing. Thanks to this fix, the full Lua test case was able to run as part of the full suite and by itself.

Once the full test was running, a more complex test was started. This test would check the Lua standard libraries to perform more complex tasks and verify that the libraries were working. However, the test quickly ran into a snag. Upon calling the function that opened the standard libraries for a Lua state, the state would throw a "PANIC" signal and forcibly close itself, thus failing the test. XINU and router were not affected, and the test case could be run again immediately with similar results. The cause of this panic is unclear. Investigations showed that only some of the standard libraries would cause the problem, but there was no real indication of why these particular libraries failed. In particular, the base library, coroutine library, table library, I/O library and OS library were known to cause the Lua panic consistently, though the remaining libraries occasionally caused the problem. It is possible that the string table problem mentioned above is related to this standard library problem, but it is hard to verify this connection. Until this problem is resolved, work on Lua has stagnated.

2.4 Installing the Lua Interpreter

With the core language working successfully, porting the Lua interpreter to XINU became feasible. The interpreter would enable the use of Lua as a stand-alone language that did not have to be embedded in C. The lack of the Lua standard libraries was discouraging, since most of the power of Lua is locked in those libraries. Still, having a working interpreter would be useful.

It was decided that the interpreter, which uses the core Lua C library to execute Lua scripts on a register-based virtual machine, should be made accessible through the XINU shell. The XINU shell has several functions for monitoring the system and is used to run the test suite or individual test cases mentioned in section 2.3. Adding the Lua interpreter to the shell would allow the user access to Lua at any time while using XINU. The current source code for the interpreter, which resides in the source file `lua.c`, was modified to fit the standard set for shell commands. The new interpreter code was then linked to the existing shell code and the command "`lua`" was added to the list of commands for the shell. With this simple process, the Lua interpreter was added to the XINU shell.

Unfortunately, simply adding the command did not mean that the interpreter would actually work. Upon calling the interpreter, the router crashed with the same string table error outline in section 2.3. It is possible that the Lua interpreter will work as it should once the string table error is resolved. In the meantime, the Lua interpreter is a XINU shell command in name only.

3 Results

The core of the Lua scripting language was successfully ported to Embedded XINU and could be used as an embedded language in XINU C programs. Test cases were designed for both embedding Lua in C and for testing the XINU C library functions that were created while porting Lua, and these cases have shown the system to be fairly stable. However, persistent memory errors in Lua have prevented further work, and being unable to use the Lua standard libraries limits the complexity and usefulness of the scripting language. An unfortunate consequence of these errors is the lack of a working Lua interpreter in XINU. Though the goals of this project were not completely met, a solid foundation has been laid for bringing the Lua language to the XINU platform.

3.1 Future Work

There are several projects that could use the groundwork established by this research. The primary goal of fully porting Lua to XINU still stands, and some of the Lua dependencies that were removed for practicality reasons could

be restored. Fully integrating the Lua interpreter with the XINU shell and adapting it to the particulars of the platform, such as by allowing Lua scripts to be read in from the user's computer over the serial port, would greatly enrich both Lua and XINU. Due to the ease with which Lua can be configured and changed, a new standard library designed for use with XINU could easily be added to the language. This new library could provide any functions that would be commonly used while developing for XINU. For example, functions for working with the network interface of the Linksys router could prove useful. If XINU is ported to platforms with more complex capabilities than the routers, platform-specific implementations of XINU Lua could be created so that Lua is not limited to the lowest common denominator of XINU platforms. Perhaps XINU will be brought to a platform with floating-point capabilities; in that case, the Lua floating-point math operations could be restored. Finally, Lua itself should be used in XINU. Parts of XINU that deal with dynamic data or strings—which are Lua's strengths—would benefit greatly from using Lua in their implementation. The XINU shell, perhaps, could be rewritten as a C program with embedded Lua. These projects would take advantage of the Lua foundation that is now part of Embedded XINU.

References

[Lua.org 2012] LUA.ORG, 2012. About Lua. www.lua.org/about.html

[Fulgham 2012] FULGHAM, BRENT. The Computer Language Benchmarks Game. www.shootout.alioth.debian.org/

[Ierusalimschy 2003] IERUSALIMSCHY, R. 2003. Programming in Lua, 1st ed. www.lua.org/pil/#1ed

[Ierusalimschy et al. 2006] IERUSALIMSCHY, R., DE FIGUEIREDO, L.H., and CELES, W. 2006. Lua 5.2 Reference Manual. www.lua.org/manual/5.2/

[Marinescu and Sutter 2011] MARINESCU, B. and SUTTER, D. 2011. The eLua Project. www.eluaproject.net/

[MSCS 2011] MARQUETTE UNIVERSITY MSCS DEPARTMENT, 2011. Embedded XINU. www.xinu.mscs.mu.edu/Main_Page