A LUA VIRTUAL MACHINE FOR RESOURCE-CONSTRAINED EMBEDDED SYSTEMS

Alex de Magalhães Machado, Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration
Federal University of Santa Catarina
P.O.BOX 476, 88040900, Florianópolis, Brazil
{alex, guto}@lisha.ufsc.br

ABSTRACT

Embedded systems and high-level languages usually belong to different worlds. It is not easy to fit a language runtime environment to a strongly constrained embedded platform. This porting mostly causes a loss of functionalities. High-level languages need large support from lower-level hardware or software so they can work. This paper describes our work to port the Lua Virtual Machine (LVM) to the Embedded Parallel Operating System (EPOS), making possible the execution of applications written in a high-level language such as Lua on embedded systems. The final system with support for all Lua libraries has less than 155 KB of size, which makes it suitable for a large amount of embedded systems. More than that, our tests showed that LVM runs faster on EPOS than on a Linux distribution.

KEYWORDS

Embedded System, Virtual Machine, High-Level Language, Lua

1. INTRODUCTION

High-level languages provide a vast set of abstractions to ease the development of applications. These abstractions are only possible with a considerable runtime support by the operating system and the hardware. Virtual machines are increasingly providing this type of support, although they require considerably more resources than usual, often impacting applications performance beyond the acceptable. Embedded systems can hardly afford such resources (Koshy et al, 2009). Sensor networks, for example, are a domain characterized by resources that are orders of magnitude smaller than what ordinary virtual machines require (Levis and Culler, 2002).

Therefore embedded systems and high-level languages usually belong to different worlds. It is not easy to fit the runtime environment of a programming language to a strongly constrained embedded platform. This porting mostly causes a loss of functionalities (Caracas et al, 2009). Additionally, languages also have tools specially designed for general-purpose computers, and some abstractions are useless or less important in embedded systems, and their support could be simplified. Our ultimate goal is to understand the environment's behavior, and then remove any unnecessary overhead. Without that overhead, the use of high-level languages in embedded systems would become reality. We want to change this scenario, by analyzing all the abstractions between application and hardware, and trying to reduce the gap between them.

This work specifically describes the adaptations in the Lua Virtual Machine (LVM) for the Embedded Parallel Operating System (EPOS). Existing embedded virtual machines are usually a subset of a desktop virtual machine. These embedded virtual machines normally do not provide features that are hard to implement on resource-constrained embedded systems. This approach eventually impacts execution time, since the virtual machine was not previously designed for this environment (Caracas et al, 2009). However, we decided to work on Lua because the LVM is a lightweight virtual machine written in C, and it is portable as far as it concerns the hardware and operating system support for libc (Ierusalimschy et al, 2007). Nonetheless, the LVM normally requires support to some functionalities that are not used by any Lua application running on EPOS, such as software localization, modules loading, and access to environment variables and external commands. These features were removed. To give a shape to the remaining features,

we defined a Lua profile, in which we create a set of available resources for the embedded version of LVM. This Lua profile aims to help Lua developers to understand the functionalities LVM provides for embedded Lua applications.

The rest of the paper is organized as follows. Section 2 describes related works on interoperability between high-level languages and embedded systems. Section 3 describes the support required by LVM, and what EPOS could provide. It also describes how we solved the problems of functionalities LVM needed but EPOS could not provide. Section 4 summarizes our Lua Profile, and Section 5 presents the evaluation of our embedded LVM in terms of size and performance. Section 6 concludes the paper and describes future works.

2. RELATED WORK

High-level languages are becoming more popular in embedded systems because they provide useful programming abstractions such as object-orientation and multi-threading (Ishikawa and Nakajima, 2005). However, embedded systems are characterized by great architectural diversity. This can be tackled by virtual machines, at the expense of impacts in performance. Various techniques have been developed in order to reduce this performance gap between native and virtual execution environments (Koshy et al, 2009).

There are different approaches for virtual machines: virtualizing real hardware, virtualizing intermediate program representation and virtualizing bytecode interpretation (Costa et al, 2007). We are focusing our work in virtual bytecode interpretation. This set of virtual machines can be classified in two classes. The first class targets middleware by position between operating system and applications. They are called middleware level virtual machines. The second replaces the entire operating system. They are called system level virtual machines (Costa et al, 2007). Since LVM does not replace the operating system, it is a middleware level virtual machine. In fact, EPOS is our operating system.

High-level languages have abstractions that require special support from the virtual machine (Koshy et al, 2009), but using virtual machines in embedded systems has some pros and cons. They enable higher levels of abstraction, but they also have to care about the execution model. LVM has a straightforward execution model, which did not have to be changed. LVM is a register-based virtual machine, unlike most of the current middleware level virtual machines for bytecode interpretation. This type of virtual machine can be implemented to be faster than stack-based virtual machine. This approach has been rewarding, since programs require far less instructions in order to execute a task (Shi et al, 2005).

The Mote Runner virtual machine is an interesting approach because it was created from scratch and it supports more than one language (Caracas et al, 2009). However, its implementation removes some features from its supported languages, e.g. threads, floating point arithmetic, some data types, multi-dimensional arrays, and introduces the event-driven programming paradigm, hence causing a negative impact on the application portability. Mote Runner only supports strictly-typed languages, unlike our approach (Caracas et al, 2009). Our approach does also remove some features, but only those that would not be used on an embedded system, such as software localization, dynamic module loading, and access to environment variables.

There are other virtual machines, such as VM* and EarlGray, with different approaches on Java Virtual Machines. VM*, specifically, focuses on optimizing a Java Virtual Machine for wireless sensor networks (Koshy et al, 2009). However, most of its optimizations are already used in the LVM or change the language's functionalities, hence impacting its portability. EarlGray is a Component-based Java Virtual Machine, therefore focusing on modularization (Ishikawa and Nakajima, 2005). This modularization is achievable through the use of EPOS, considering LVM and its libraries as components (Schulter et al, 2007). Scylla and KESO are other virtual machines aiming efficiency and low overhead, but they have very different ways to solve these problems. In fact, Scylla is a system level virtual machine (Stanley-Marbell and Iftode, 2000), and KESO is a tool that compiles Java bytecode to C source code, the system's native language (Stilkerich et al, 2006).

Some of these approaches are unsuitable for small embedded devices, and others make significant changes to the supported languages, requiring applications to be designed specifically for the host system, therefore impacting portability.

3. EPOS SUPPORT FOR LVM

This section describes in details our approach to port the LVM to EPOS. LVM is written in C and compiles as a C++ library with minor changes (Ierusalimschy et al, 2007). That is how we use it on EPOS. Therefore, its portability relies only on the underlying hardware and operating system.

EPOS is a framework designed to guide and provide architecture transparency to the development of scenario independent component families that can be used in different environments through applying aspect programs (Schulter et al, 2007; Fröhlich and Schröder-Preikschat, 2000). It is designed for embedded hardware and it provides most of the support Lua needs to work. Nevertheless, Lua uses C standard libraries that are not present in EPOS. We could add these libraries to the environment and most of it would work properly, but we would be adding unnecessary code. We therefore chose to implement only the functionalities Lua needs. The next subsections address every functionality we had to handle in order to allow full support for Lua applications on EPOS.

3.1 Character, Memory, and String Handling

Character Classification functions are used in LVM inside lexical analyzer routines and also for pattern matching. Since these functions were not available in EPOS, they were implemented. This implementation was simple and took into consideration the ASCII character set.

EPOS already had a header called string.h, which defined some Memory and String handling functions. Lua uses most of these functions and a few more. Since they are frequently used, they were implemented inside the file string.cc. The only function that we did not implement was strcoll, although it was used in the LVM. The calls to this function were replaced by a call to the strcmp function. We will discuss more about this subject in the next subsection.

3.2 Localization, Time, and Date

The main use for software localization tools in the LVM is to provide software localization for the Lua applications. However, the current locale can also be used to inform the lexical analyzer what is the current representation for the decimal point. The Lua Operating System Library provides the setlocale function for Lua applications. This function simply calls the C standard library. We could not find a utility for software localization in EPOS, so we removed this feature. This removes the setlocale function from the Lua library. Besides, we removed the support for locale in every function that originally used it, such as String and Time handling functions.

The Lua Operating System Library provides the following manipulation and formatting functions to Lua applications: 'clock', 'date', 'difftime', and 'time'. Lua uses C library functions in order to achieve that. EPOS provides the following classes for these purposes: Real-Time Counter, Date, Clock, and Chronometer. We implemented these functions using the classes above. Our implementation differs from the original in that it does not take into account the current locale, as we already discussed.

Our Chronometer class counts how much time passed since the last call to its 'start' function. In order to do so, it uses an architecture-dependent function called 'time_stamp'. We used the Chronometer class to implement the 'clock' function. Our Real-Time Counter class uses its machine-dependent functions to implement some of the other functions.

3.3 General Purpose Standard Library

The C standard library is used in LVM for various purposes. The realloc function is used for all the LVM memory allocations. It was implemented in the file malloc.cc using the EPOS malloc function. The exit function is used when an error occurs. We now use the EPOS exit function of the Thread class. The strtoul, abs and strtod functions were not provided by EPOS and therefore they were implemented straightforwardly. The getenv and system functions were not implemented. The getenv function is mostly used for loading and building modules in Lua. Currently, the embedded lua virtual machine only executes one Lua script at a time, which is placed inside EPOS application. Therefore, the Lua Package Library was disabled. Besides this

specific use, the Lua Operating System Library also provides these functions to Lua applications. They were not implemented because they use environment variables and external commands, and our Lua profile does not support these functionalities, as we will discuss further.

The Lua Mathematical Library provides the rand and srand functions for Lua applications. The first generates a random number and the second sets the seed. They were implemented in EPOS using the Pseudo_Random class with some minor changes.

3.4 Input and Output

Various file systems were developed for EPOS, but they are not compatible with the way general-purpose computers use file systems. The functions that receive or return files were not implemented. However, we created an FStream class, similar to the OStream class present on EPOS, and declared all those functions there. Hence these functions are not supported, but they could be eventually implemented and their support would be ready to work. The functions responsible for reading from standard input stream were not implemented, since EPOS has no input device by default. This also could be easily implemented the same way of the FStream class. The functions responsible for writing to standard output stream were implemented using the existing OStream class.

3.5 Mathematical Operations

Lua provides the Lua Mathematical Library for its applications. Nevertheless, the underlying hardware does not necessarily support floating-point number representation. We implemented a simple configuration which informs the virtual machine whether the hardware supports or not floating-point representation. If so, these functions would be available for Lua applications. If not so, the Lua Mathematical Library would be reduced to only fixed-point functions.

4. LUA PROFILE

The set of modifications described in the last section only addresses C library functionalities in terms of what LVM needs and what EPOS provides. The work we did in adapting LVM to EPOS, implementing new features and changing others, accomplishes our goal of providing embedded system support for LVM.

Table 1. Lua Library functions. Crossed out words are not present on our Lua Profile, and thus are not available on EPOS

Basic library	Package Coroutine library	String library	Mathematical library	I/O library	OS and Table library	Debug library
assert , error	~~module~~	byte	abs	io.close	clock	debug
collectgarbage	~~require~~	char	acos , asin	io.flush	date	getfenv
~~dofile~~ , ~~loadfile~~	~~cpath~~	dump	atan , atan2	~~io.input~~	difftime	gethook
getfenv	~~loaded~~	find	ceil , floor	~~io.lines~~	~~execute~~	getinfo
getmetatable	~~loaders~~	format	cos , sin	~~io.open~~	exit	getlocal
ipairs , pairs	~~loadlib~~	gmatch	cosh , sinh	io.output	~~getenv~~	getmetatable
load , loadstring	~~path~~	gsub	deg	~~io.popen~~	~~remove~~	getregistry
next	~~preload~~	len	exp	~~io.read~~	~~rename~~	getupvalue
pcall , xpcall	~~seeall~~	lower , upper	fmod	~~io.tmpfile~~	~~setlocale~~	fenv
print		match	frexp , ldexp	~~io.type~~	time	sethook
rawequal	create	rep	log , log10	io.write	~~tmpname~~	setlocal
rawget, rawset	resume	reverse	max , min	~~file:close~~		setmetatable
select	running	sub	modf	~~file:flush~~	concat	setupvalue
setfenv	status		pow	~~file:lines~~	insert	traceback
setmetatable	wrap		rad	~~file:read~~	maxn	
tonumber	yield		random	~~file:seek~~	remove	
tostring			randomseed	~~file:setvbuf~~	sort	
type			sqrt	~~file:write~~		
unpack			tan , tanh			

Through these modifications, we defined features that EPOS would not support and consequently LVM cannot use. That is the case of OS library functions such as ‘getenv’, ‘execute’ and ‘setlocale’. The ‘execute’ function, specifically, is useful for Lua applications that control larger systems, but this kind of control can be performed outside Lua, with the use of EPOS facilities. We are looking forward to support this ‘execute’ function inside Lua in future works.

However, two problems arise from this scenario. The first is that Lua developers may not know whether EPOS supports an LVM functionality or not. The second is that Lua developers may never need some features in an embedded Lua application, but LVM would still support these features.

Therefore, in order to properly create an efficient and maintainable Lua runtime environment for embedded systems, we need to make clear what LVM provides and what Lua applications need. Even if a feature is supported by EPOS and LVM, it is possible that it will never be used by a Lua application running on EPOS. We solve these two problems creating a Lua Profile for embedded systems, which defines a subset of functionalities that LVM provides for embedded Lua applications.

Table 1 shows all Lua libraries, with all the functions they implement. The features removed in our Lua Profile are crossed out. As we can see, we did not remove many features from the LVM, and therefore our Lua Profile is vast and it is able to support most of the real Lua applications.

5. EVALUATION

We intented to evaluate the overhead LVM caused on EPOS environment in terms of size, and also analyse if the embedded LVM runs on EPOS similarly to a Linux distribution, so Lua developers could expect comparable performance between them.

Lua Test Application

```
function fib(n)
        N=N+1
        if n<2 then
                return n
        else
                return fib(n-1)+fib(n-2)
        end
end

function test(f)
        N=0
        local v=f(n)
        s = string.format("Value: %d, Evals: %d", v, N)
end

n=24
test(fib)
```

Fig. 1: Lua Test Application.

We created a test application in Lua and in C++, EPOS native language. Fig. 1 and Fig. 2 show the source code of the two versions. This simple application calculates the 24th Fibonacci number. An enhanced Lua version of this test application is deployed with the source code of the official implementation of Lua. The Lua version uses a recursive function called fib and then calls the format function of the String library to create a result string. The C++ version uses our own implementation of functions itoa and sprint. We tested these applications on EPOS and on an Ubuntu 9.04 with 2.6.28-19 kernel. Both systems were executed on the

same platform. Therefore, we have four different execution times. All these execution times were measured with an oscilloscope.

As we can see in Fig. 3, the C++ application is faster than the Lua application wherever it is running. Nonetheless, Lua is still very attractive, since its high-level abstractions ease the development of applications. The C++ application was faster on the Linux distribution because of optimizations in the libc. The Lua application, however, was faster on EPOS.

C++ Test Application

```
int N;

int fib(int n)
{
    N++;
    if (n < 2)
        return n;
    else
        return fib(n-1)+fib(n-2);
}

int main()
{
    int n = 24;
    test(n);
    return 0;
}
```

```
void test(int n)
{
    N = 0;
    int v = fib(n);
    char s[30];
    const char * va_list[2];
    va_list[0] = itoa(v,10);
    va_list[1] = itoa(N,10);
    sprintf(s,"Value: %d, Evals: %d",va_list);
}
```

Fig. 2: C++ Test Application.

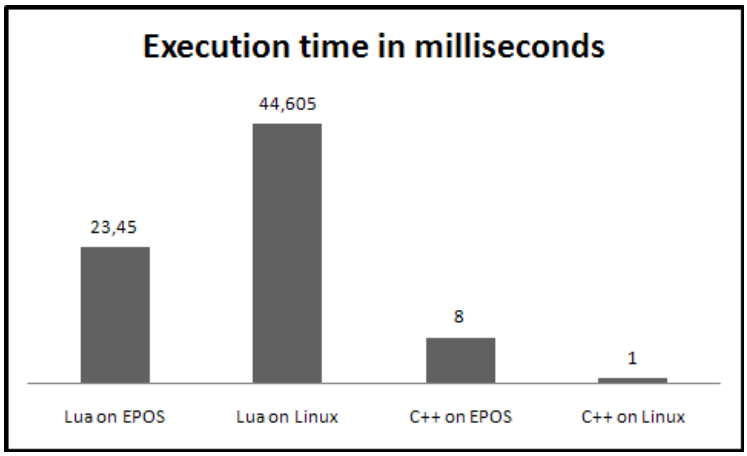

Fig. 3: Execution time in milliseconds of the two applications running on the two different systems.

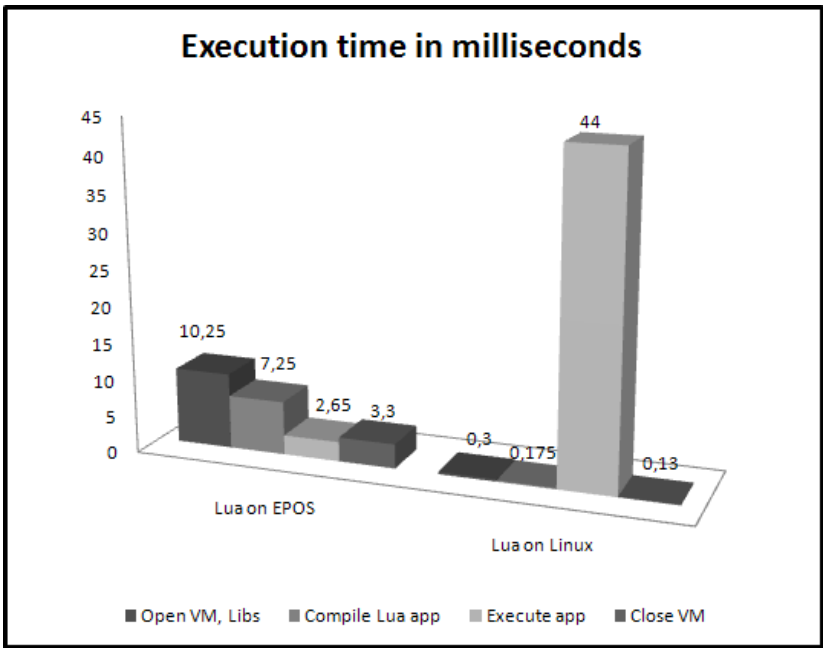

Fig. 4: Execution time in milliseconds of each step of the LVM execution on EPOS and Linux.

Fig. 4 goes beyond and shows how much time it takes to perform each step of the LVM execution. These discrepant times depend primarily on the application, and therefore we cannot say that Lua runs faster or slower on EPOS. In fact, we are showing that the overhead of the LVM execution on EPOS is not much different from the Linux version. For the record, the LVM execution time standard deviation was 0.8857 ms, and the application execution time specifically had 0.1985 ms of standard deviation.

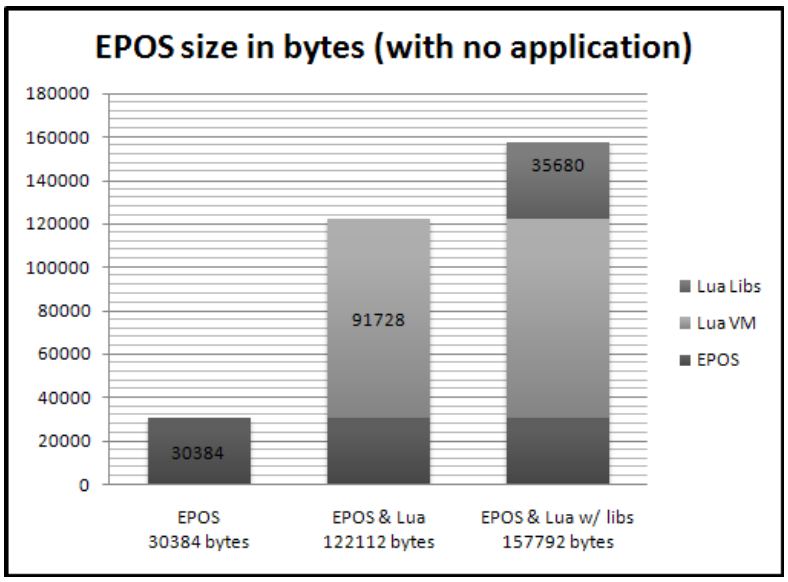

Fig. 5: EPOS size in bytes, without any application.

Fig. 5 shows that EPOS has less than 32KB of size. The LVM, without its libraries, has size of approximately 90KB. Hence the EPOS size with basic Lua support is approximately 120KB. The libraries size is approximately 35KB. Theses sizes do not count the application size, which varies depending on the system.

6. CONCLUSION

Although some embedded systems often do not have 120KB available just to support the system, this size is comparable to those of our related works. In fact, some other high-level languages virtual machines have much bigger sizes (Ishikawa and Nakajima, 2005). Also, the LVM code in EPOS is really lighter than in Linux, but our tests show that we can improve performance and also reduce even more its size. That shall be the next step.

Our final EPOS/LVM system is able to execute a high-level language application, with full support to all high-level abstractions Lua already provided for desktop applications. More than that, we achieved to support a high-level language without restricting its flexibility, without disabling its portability, and most of all, without making the system unsuitable for embedded systems.

Finally, our approach showed us common aspects in adapting high-level languages for embedded systems, such as internal communications between virtual machine and operating system and libraries support by the virtual machine. We are looking forward to generalize these steps in order to ease the adaptations of virtual machine high-level languages for embedded systems.

REFERENCES

Caracas, A. et al, 2009. Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices. *Sensor Technologies and Applications. SENSORCOMM '09. Third International Conference on*, pp. 117-125.

Costa, N. et al, 2007. Virtual Machines Applied to WSN's: The state-of-the-art and classification. *Systems and Networks Communications. ICSNC 2007. Second International Conference on*, pp.50-50.

Fröhlich, A. A. and Schröder-Preikschat, W., 2000. Scenario Adapters: Efficiently Adapting Components. *In Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*. Orlando, USA.

Ierusalimschy, R. et al, 2007. The evolution of Lua. *In HOPL III: Proceedings of the third ACMSIGPLAN conference on History of programming languages*. New York, NY, USA, pages 2–1–2–26. ACM Press.

Ishikawa, H. and Nakajima, T., 2005. EarlGray: A Component-Based Java Virtual Machine for Embedded Systems. *Object-Oriented Real-Time Distributed Computing. ISORC 2005. Eighth IEEE International Symposium on*, pp. 403-409.

Koshy, J. et al, 2009. Optimizing Embedded Virtual Machines. *Computational Science and Engineering. CSE '09. International Conference on*, pp. 342-351.

Levis, P. and Culler, D., 2002. Maté: a tiny virtual machine for sensor networks. *In International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95.

Schulter, A. et al, 2007. A Tool for Supporting and Automating the Development of Component-based Embedded Systems. *in Journal of Object Technology*, Vol. 6, No. 9, Special Issue: TOOLS EUROPE 2007, p. 399-416.

Shi, Y. et al, 2008. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 4, No. 4, p. 1-36.

Stanley-Marbell, P. and Iftode, L, 2000. Scylla: a smart virtual machine for mobile embedded systems. *Mobile Computing Systems and Applications, Third IEEE Workshop on*, pp. 41-50.

Stilkerich, M. et al, 2006. OSEK/VDX API for Java. *In Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*. San Jose, California, p.4-es.