Lynq: A Lightweight Software Layer for Rapid SoC FPGA Prototyping

Jonathan Dechelotte, Dominique Dallet, Jeremie Crenne
IMS, Univ. Bordeaux, Bordeaux INP
Bordeaux, FRANCE
{first.last}@ims-bordeaux.fr

Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
tessier@umass.edu

***Abstract*—Modern FPGAs include a diverse collection of heterogeneous processing elements including microprocessors. However, in many cases, specialized knowledge is required to integrate processing elements, IP hardware cores, memory interfaces and interconnects together. Xilinx recently released PYNQ, an open-source framework to enable interactive testing, rapid design iteration, and fast prototyping on SoC FPGAs. In this paper we present Lynq, Lua for Lynq, a lightweight software layer for rapid SoC FPGA prototyping on Xilinx Zynq devices. We evaluate the performance and energy efficiency of the new software and assess hardware integration efficiency versus competing approaches. It is shown that we outperform Python implementations with PYNQ even when a JITed version of Python is available. Run-time speedups between $3.2\times$ and $4.9\times$ are shown with an energy improvement of $2.5\times$ to $4.8\times$ versus PYNQ. System bootup is achieved in less than 10 ms which fits time-critical application requirements.**

***Keywords*-SoC FPGA, Rapid prototyping, Zynq, JIT, Lua, PYNQ**

I. Introduction

Programming embedded applications is challenging as optimum performance cannot be reached without tailoring an application to the targeted architecture. Heterogeneous SoC FPGA systems provide effective platforms for embedded applications as evidenced by recent industry acquisitions [1], [2]. To date, most embedded systems research for SoC FPGAs has focused on the optimization of individual hardware and software approaches rather than their seamless integration. Software efficiency is achieved with the use of highly-tuned libraries while hardware development is facilitated by the use of high-level synthesis (HLS) to incorporate hardware accelerators into end-user applications. Flexibility, high-performance and ease-to-use are critical to the wide-scale deployment of SoC FPGAs to allow for their broad use by engineers without specialized hardware knowledge.

To address the issue of fast, efficient hardware-software integration on FPGA SoCs, we present Lynq, Lua for Zynq, a lightweight open-source LuaJIT-based [3] software layer targeting ARM processor architectures. Lynq makes it easier for designers of embedded systems to take advantage of SoC-specific features in their applications while achieving high computational performance, energy efficiency and boot time that are comparable to hand-crafted integrations. Lynq makes specific key contributions over existing platforms:

- It runs on top of Xilinx device drivers and libraries.
- The system combines a high-speed interpreter and a just-in-time compiler.
- Lynq provides an application programming interface (API) to expose the ARM architecture and FPGA fabric to the user.
- The system consumes $2.5\times$ to $4.8\times$ less energy than PYNQ in our experimental setup.
- Lynq allows booting from an SD card in 10 ms, taking advantage of its compact implementation and limited LuaJIT compile time.

The rest of the paper is organized as follows: Section II gives an overview of related work. Section III details Lynq software and hardware design decisions. Our experimental approach is described in Section IV and experimental results are assessed in Section V. Finally, Section VI concludes.

II. Background

A number of approaches exist that make SoC FPGA development easier. Software optimization and automated hardware accelerator generation using high-level synthesis abstract away the complexities of an SoC design. This use of high-level programming languages to accelerate development has been an on-going effort for decades.

As part of this effort, Xilinx recently released PYNQ [6] to support rapid application development. PYNQ makes use of the Python programming language and overlay bitstream configurations. The PYNQ API extends common Python libraries and packages to support access to programmable fabrics through memory-mapped I/O (MMIO) and direct memory access (DMA). As a result, designers can develop applications without designing hardware accelerators and associated device drivers. PYNQ also provides a Python interface to allow access to intellectual property (IP) blocks in the programmable logic (PL). The blocks are controlled by Python running in the processing system (PS). They can be dynamically loaded into the Zynq PL at run-time to provide the functionality required by a software application.

Although useful, PYNQ has limitations. Schmidt *et al.* [9] evaluated the relevance of PYNQ for rapid application development by studying its performance impact and the location of associated bottlenecks. A compass edge detection algorithm was implemented in Python resulting in a running

	Ref.	Language	License	Architecture			Platform				Compatibility *	
				x86	x64	ARM	Win.	Lin.	OSX	Standalone	JIT	PL
Python	[4]	Python	PSF	●	●	●	●	●	●	○	○	○
PyPy	[5]	Python	MIT	●	●	●	●	●	●	○	●	○
PYNQ	[6]	Python	BSD	○	○	●	○	●	○	○	○	●
BITS	[7]	Python	n/a	●	○	○	○	○	○	●	○	○
Lua	[8]	Lua	MIT	●	●	●	●	●	●	○	○	○
LuaJIT	[3]	Lua	MIT	●	●	●	●	●	●	○	●	○
(this work)		Lua	MIT	○	○	●	○	○	○	●	●	●

* JIT = just-in-time compiler, PL = programmable logic

Table I: State of the art implementation solutions and features comparison for SoC FPGA integration

time that is $334\times$ slower than the C version. A speedup of $11.5\times$ over the C version was observed if OpenCV libraries are used. Combined with a hardware accelerated core, the speedup increases to $30\times$. The result shows that the benefit of Python is limited if performance is a concern. The choice of Python as a glue language for hardware interfacing facilitates fast prototyping, but Python implementations also tend to reduce the achievable performance of the platform due to the use of interpreted code. Rigo *et al.* [5] proposed PyPy as an option over Python. PyPy is an alternative implementation which uses an interpreter and a just-in-time (JIT) compiler to achieve better performance. However, PyPy implements a restricted subset of the Python language and must include dependencies at build-time. In this paper, we introduce Lynq, an open-source system integration software. Lynq is a software development environment based on the Lua high-level programming language. It provides a lightweight and low-overhead alternative to PYNQ with similar high-level hardware interfacing services and access to the programmable fabric. Lynq relies on the LuaJIT just-in-time compiler to execute user applications rather than a Python interpreter which drastically improves performance. Lynq runs standalone on the ARM Cortex-A9 processor architecture. We summarize the features of our system versus the previous solutions discussed in this section in Table I.

III. Lynq Overview

The lightweight software layer currently targets Xilinx Zynq-7000 series SoC FPGAs, although it could be adapted for other SoCs. Lynq's execution environment was designed with two major goals in mind. The first one was to provide a lightweight software interface between hardware and the user application. The second design goal was to make available a high-level API to allow interactions between the Zynq PS and PL. This interface facilitates rapid prototyping and development. To execute a user program, Lynq relies on three underlying software layers, 1) a customizable board support package (BSP), 2) a Lua jitted compiler LuaJIT, and 3) a Lua API. In the following subsections, we give a detailed description of this software stack.

A. Lua

To create an efficient interface between software and hardware, we use Lua [8], a lightweight, portable and embeddable scripting language. Lua supports procedural, functional and data-driven programming, object-oriented programming and data description. The language combines procedural syntax with data description constructs based on associative arrays and extensible semantics. Lua is also dynamically typed, runs by compiling and interpreting bytecode with a register-based virtual machine (VM), and has automatic memory management with incremental garbage collection.

B. LuaJIT

To improve performance execution time, Pall [3] proposed LuaJIT, a JIT compiler for the Lua programming language. A JIT compiler turns bytecode into architecture instructions that can be executed by the target processor. LuaJIT is a tracing JIT rather than a method JIT. It works by discovering, identifying and optimizing linear sequences of bytecodes (traces) during execution. While executing the bytecode, the LuaJIT interpreter records execution information in LuaJIT bytecode and intermediate representation (IR) is emitted from that bytecode. The compilation phase of JIT execution calls architecture-specific translation hooks to generate machine code from the recorded IR. Further iterations execute the compiled code by using a vector of pointers to generated assembly stubs, indexed by the bytecode instructions they implement. LuaJIT has been carefully hand-crafted for performance and outperforms most interpreted and jitted languages [10]. Our study points also in that direction.

C. Modifications to LuaJIT

Lua and LuaJIT were not originally designed to work as standalone, but rather were positioned to run on top of Windows, Linux, BSD, OSX or POSIX OS. Specifically, the underlying C library for Lua and LuaJIT depends on subroutine calls for operating system services. Because some of the system calls (syscalls) needed by LuaJIT are not provided when an OS is not present, they must be implemented to provide links to subroutines. Minor changes have been made

to LuaJIT source code to support standalone Lua program execution on 32-bit and 64-bit ARM Cortex-A9 processor architectures. Due to a lack of privileges in user mode, cache flushes and invalidations on the ARM must be done via a syscall. This syscall has been implemented in software and the toolchain provides a cache synchronization function that uses it. The LuaJIT source code has been modified accordingly to make use of this function.

D. Lynq Lua API

One contribution of Lynq is the definition of a Lua API. The Lynq Lua API is a high-level user interface that allows for programming the ARM processor and managing the FPGA fabric.

The API manages data transfers between the Lua environment in the PS and the IP blocks in the PL. It also extends Lua standard libraries and packages to include support for general purpose input/output (GPIO), timers, interrupts, DMA, socket, MMIO access, central processing unit (CPU) control and FPGA bitstream programming. Internally, the Lynq API is implemented with a set of wrappers written in C which encapsulate BSP libraries and device drivers provided by Xilinx. It exports functions, methods and constants as modules that can be later loaded in a Lua script.

Lynq BSP is based on two software libraries: FatFS for manipulating files and lwIP to provide a network interface. Many legacy and common applications rely on OS services such as file systems and sockets. Lynq is a limited environment that provides these essential services. File manipulation is one of the most basic operations an OS must provide. Although standalone, LuaJIT make extensive use of a file subsystem to manipulate files in the standard I/O library, load and execute script files and manage user libraries. To offer this functionality, all syscalls needed by LuaJIT were reimplemented in libc using a file allocation table (FAT) file system. This work uses the FatFS library, a generic FAT module written in C and optimized for embedded systems. Basic network socket facilities are also provided. LwIP is an open source TCP/IP stack designed for embedded systems. This third-party library relies on Xilinx PS Ethernet media access controller (EMAC) drivers. TCP and UDP protocol support has been implemented in the Lynq API with lwIP integrated as Lua modules. Sockets are available to any Lua application once they are properly initialized.

IV. Experimental Approach

To understand and evaluate the software performance benefit of using LuaJIT over existing solutions, we used the SciMark 2.0 benchmarks suite [11]. SciMark is a composite benchmark measuring the performance of numerical codes occurring in scientific applications. It provides an indication of how the underlying VM/JIT performs. SciMark reports a score in mflops. Some of the kernels exercise transcendental functions (e.g. sin, cos) as well as integer operations, so the mflops count is approximated. However, the same mflops count is used consistently, so comparisons are valid. The problems sizes are set to be small to isolate the effects of memory hierarchy and focus on internal VM/JIT and processor issues.

SciMark includes five computational kernels: Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (SOR), Monte Carlo integration (MC), Sparse matrix multiply (SPARSE) and Lower-Upper matrix factorization (LU). Each kernel was implemented in C, Python and Lua. To provide a fair comparison, we evaluated Lua and LuaJIT execution time performance with both interpreted Python and jitted PyPy. C kernels were compiled with ARM GCC 6.2.1 with and without compiler optimizations (O0 and O3). Python kernels were implemented in Python 3.4 and PyPy 5.8. Since the PyPy 5.8 interpreter does not support Python 3.4 syntax, we also implemented all the kernels in Python 2.7. The Lua kernels were implemented in Lua 5.1.5 and LuaJIT 2.0.5 without any compiler optimizations and were either executed in interpreted or jitted mode. We also used the Lua foreign function interface (FFI) library to evaluate the impact of low-level memory allocation and manipulation on execution time. The Lynq executable was compiled using GCC 6.2.1 without compiler optimizations.

All presented results were measured from single-threaded software implementations, so a single ARM CPU core was used. PYNQ and our system were freshly booted from an 8 GB class 10 SD card and were otherwise idle. For experiments with PYNQ, no hypervisor module was loaded and all power-management features were turned off. All executable code was cached in memory prior to each measurement.

Finally, we measured a Lynq booting time of 10ms when no network is required.

V. Experimental Results

The presented benchmark results in this section demonstrate the performance of executable code and VM implementations. The results give insights into how Python and Lua VM implementations perform in comparison with other approaches. A typical Lynq image size is 800 KB when LuaJIT is compiled with O3 and mthumb GCC compiler options. This number does not include the 3.85 MB ZedBoard full uncompressed bitstream configuration file.

The bar graphs in Figure 1(a) compare the computational results in mflops of SciMark benchmark kernels in PYNQ and Lynq environments running C, Python and Lua implementations. We first compare Python 3.4 and Lua 5.1.5 interpreters and note that Lua is $3.9\times$ to $41.7\times$ faster. The comparison between Python 3.4 and the LuaJIT 2.0.5 specific interpreter accentuates the differences in Python and Lua virtual machine implementations ($10.6\times$ to $108.1\times$ faster). When enabled, the LuaJIT jitted mode performs $36.8\times$ to $1,350.4\times$ faster. When using FFI, LuaJIT pushes

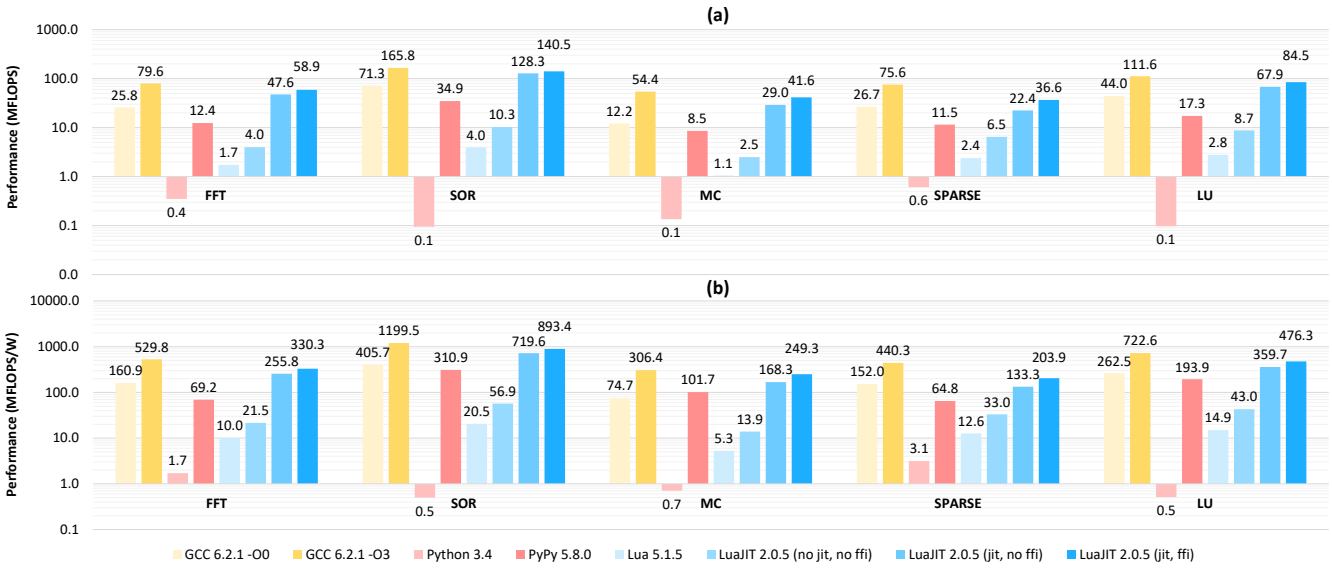

Figure 1: Comparison of computation performance in mflops (a) and energy efficiency in mflops/Watt (b) when running SciMark benchmark kernels for various implementations. Bar graphs are presented using logarithmic scales

the Python interpreter to its performance limit. The computational raw performance is $60\times$ to $1,478.9\times$ faster than Python. In these cases, the execution time results from Lua jitted implementations are compared with Python interpreted implementations increasing the performance spread. To make a fairer comparison, PyPy 5.8.0 performance is compared with LuaJIT with FFI resulting in performance $3.2\times$ to $4.9\times$ better in favor of LuaJIT. LuaJIT with FFI is also $1.4\times$ to $3.4\times$ faster than C implementations created with the O0 compiler optimization. However, as expected when using GCC with the O3 compiler optimization, C outperforms LuaJIT by a factor of $1.2\times$ to $2.1\times$. For the sake of fairness, we compare results in terms of energy efficiency for every implementations studied in this paper. As shown in Figure 1(b), LuaJIT performs $2.5\times$ to $4.8\times$ better than PyPy for the entire benchmark. The choice of LuaJIT for Lynq implementation exhibits significant performance and energy benefits over the PYNQ Python interpreter and improvements over PyPy.

VI. Conclusions

A lightweight software layer for rapid SoC FPGA prototyping, Lynq, has been presented in this paper. It can operate in standalone mode and takes advantage of a Lua-jitted implementation, LuaJIT, to provide dramatically improved boot time and higher computational performance and better performance per Watt versus existing approaches, such as PYNQ. It features a high-level API that exposes the ARM processor and FPGA programmable logic to the user. Lynq is open-source under an MIT license and is available at [12].

Acknowledgment

Research reported in this publication is part of the APOGEES project supported by BPI France, region Occitanie and region Nouvelle Aquitaine. The APOGEES project has been labeled from pole Aerospace Valley, pole Images et Reseaux and pole Elopsys in the framework of the French FUI22 research program.

References

[1] Intel, "Intel completes acquisition of Altera," 2015. [Online]. Available: https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[2] Amazon, "Amazon elastic compute cloud," 2006. [Online]. Available: https://aws.amazon.com/fr/ec2

[3] M. Pall. (2008) The LuaJIT project. [Online]. Available: http://luajit.org

[4] G. Van Rossum *et al.*, "Python programming language," in *USENIX ATC*, 2007.

[5] A. Rigo *et al.* (2003) PyPy: A fast, compliant alternative implementation of the python language. [Online]. Available: https://pypy.org/

[6] Xilinx. (2016) PYNQ: Python productivity for Zynq. [Online]. Available: http://www.pynq.io

[7] Intel. Bios implementation test suite. [Online]. Available: http://biosbits.org

[8] R. Ierusalimschy *et al.*, "Lua-an extensible extension language," *Softw., Pract. Exper.*, 1996.

[9] A. G. Schmidt *et al.*, "Evaluating rapid application development with python for heterogeneous processor-based FPGAs," in *FCCM*, 2017.

[10] C. F. Bolz and L. Tratt, "The impact of meta-tracing on VM design and implementation," *Sc. of Computer Programming*, 2015.

[11] R. Pozo and B. Miller. (2000) Scimark 2.0. [Online]. Available: http://math.nist.gov/scimark2

[12] Lynq [online] available https://github.com/contactlynq/lynq.