An Open-Source Realtime Computational Platform (Short WIP Paper)

Pavan Mehrotra*
Stanford University
Department of Electrical Engineering
Stanford, CA, USA
pavmeh@stanford.edu

Sabar Dasgupta*
Stanford University
Department of Electrical Engineering
Stanford, CA, USA
sabard@stanford.edu

Samantha Robertson
Stanford University
Mathematical and Computational Science Program
Stanford, CA, USA
srobert4@stanford.edu

Paul Nuyujukian
Stanford University
Department of Bioengineering
Department of Neurosurgery
Department of Electrical Engineering
Neurosciences Program
Neurosciences Institute
Bio-X Institute
Stanford, CA, USA
18lctes@pn.stanford.edu

Abstract

Systems neuroscience studies involving in-vivo models often require realtime data processing. In these studies, many events must be monitored and processed quickly, including behavior of the subject (e.g., movement of a limb) or features of neural data (e.g., a neuron transmitting an action potential). Unfortunately, most realtime platforms are proprietary, require specific architectures, or are limited to low-level programming languages. Here we present a hardware-independent, open-source realtime computation platform that supports high-level programming. The resulting platform, LiCoRICE, can process on order 10E10 bits/sec of network data at 1 ms ticks with 18.2 μs jitter. It connects to various inputs and outputs (e.g., DIO, Ethernet, database logging, and analog line in/out) and minimizes reliance on custom device drivers by leveraging peripheral support via the Linux kernel. Its modular architecture supports model-based design for rapid prototyping with C and Python/Cython and can perform numerical operations via BLAS/LAPACK-optimized NumPy that is statically compiled via Numba's pycc. LiCoRICE is not only suitable for systems neuroscience research, but also for applications requiring closed-loop realtime data processing from robotics and control systems to interactive applications and quantitative financial trading.

CCS Concepts • **Computer systems organization** → **Real-time languages**; *Embedded software*; • **Computing methodologies** → *Real-time simulation*;

Keywords realtime, python, numerical computation

ACM Reference Format:
Pavan Mehrotra, Sabar Dasgupta, Samantha Robertson, and Paul Nuyujukian. 2018. An Open-Source Realtime Computational Platform (Short WIP Paper). In *Proceedings of 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3211332.3211344

*Contributed equally to this work.

LCTES'18, June 19–20, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5803-3/18/06…$15.00
https://doi.org/10.1145/3211332.3211344

1 Introduction

The field of systems neuroscience often conducts experiments with animal models to better understand the brain and nervous system. Many of these studies rely on precise timing and behavioral control software to prompt and reward the subjects under study. Further, in engineering applications of systems neuroscience such a brain-machine interfaces, computations based on neural data must be performed and then fed back to the user with millisecond accuracy and as minimal delay as possible. The approximate time duration of quantization in electrophysiologically-oriented systems neuroscience is one millisecond, the approximate duration of an action potential—the digital output signal of a neuron.

This duration is long enough for meaningful processing with modern computers, and thus realtime interaction with the mammalian nervous system is not only achievable, but now a standard approach in systems neuroscience.

For researchers, the choice of computational platform has largely been limited to custom commercial offerings. Some commonly used platforms are run on DOS [NIM [n. d.]; Ref [n. d.]], while others are proprietary operating systems such as VxWorks (Wind River Systems, Alameda, CA) and QNX (BlackBerry, Ottawa, ON, CA). National Instruments (Austin, TX) offers both modified kernels and a realtime Linux operating system for use with their hardware and LabVIEW suite of products. Prototyping environments such as Simulink Real-Time deliver a realtime operating system that can perform numerical operations on conventional x86 hardware. Such commercial offerings, while turnkey, are costly, may require a specific programming language, and often have limited support for modern hardware and computer architecture.

The aim of this work was to develop a realtime environment supporting linear algebra operations while transparently leveraging features of modern computer systems (e.g., multicore CPUs, hundreds of GBs of addressable memory, 10 gigabit Ethernet networking, and high-level programming languages). The resulting framework, named LiCoRICE (Linux Comodular Realtime Interactive Computation Engine), is an open-source environment running on x86 hardware that supports C and Python.

2 Implementation

LiCoRICE is a soft-realtime application platform that is able to accept data from multiple inputs, perform operations and computations on this acquired data, save it to disk in a structured database, and transmit data through multiple outputs. It performs this by using a modular design, where inputs are *sources*, outputs are *sinks*, and operations are performed by *modules*. LiCoRICE runs primarily on x86_64 architecture and the software is released under the GPL2. The latest version can be found at http://licorice.stanford.edu.

2.1 User-Level System

Each LiCoRICE module runs as a separate process. Given sufficient CPU cores, processes are shielded separately or share a core only with processes on a different level in the topological order defined in Figure 1. This approach also avoids any synchronization issues associated with the Python Global Interpreter Lock[Emb [n. d.]].

2.1.1 Compilation

A LiCoRICE model is defined by a YAML config file which details the signals, sources, modules, and sinks for a given model. A generator script then creates the module scaffolding and copies the user-written Python or C code that each module will execute every tick along with any necessary

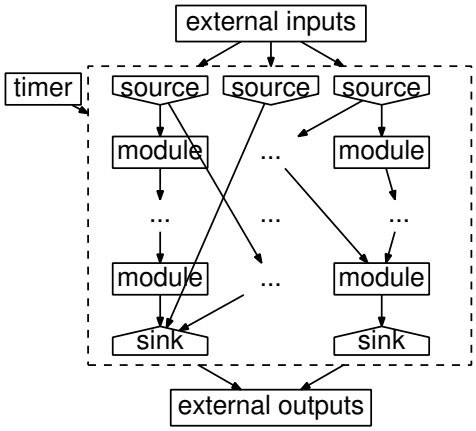

Figure 1. LiCoRICE implements model-based design, which can be conceptualized as a directed graph. Vertices of the graph are processes (i.e., sources, modules, and sinks), and edges are signals (NumPy SharedArray) permit data flow processes. Sources convert system inputs into signals for downstream processing and sinks transmit signals out of the system. The execution of processes is coordinated by a central timer process, which instructs all processes to execute once per tick in the appropriate order.

I/O handlers. Python modules performing only NumPy operations are compiled ahead-of-time into shared objects by `numba.pycc` [Num [n. d.]]. This permits efficient execution of numerical operations by a high-level language while avoiding the Python API, which would result in unpredictable memory allocation and disrupt realtime guarantees. Numba supports a significant subset of the NumPy API and can take advantage of optimization from BLAS/LAPACK if appropriately compiled. Functions not supported in Numba are compiled via Cython, at the potential risk of timing uncertainty.

2.1.2 Data Flow

Sources are executed at the start of a tick by the timer process in parallel, subject to CPU core availability. Sources will often aggregate data asynchronously and buffer it to present them to the system at the start of tick boundaries.

Signals in LiCoRICE is are stored as NumPy SharedArray objects, which store the array data in Linux shared memory. Each NumPy SharedArray may have a user-definable history that is a circular buffer. Each process that requires a specific signal attaches to the SharedArray in Python or directly opens the shared memory in C. Only one process is permitted to write data to a given signal, but signals may be read by an arbitrary number of processes.

Sinks take signals as inputs in the same way as modules. However, they also process and output the signal out of the system to peripherals. A specific filelogger sink writes signals into an SQLite database.

Table 1. System Capabilities

Worst-case jitter	18.2 μs
Average latency	1.53 ms
Maximum data throughput	119.2 MB/s
Inter-tick overhead	2.98 μs

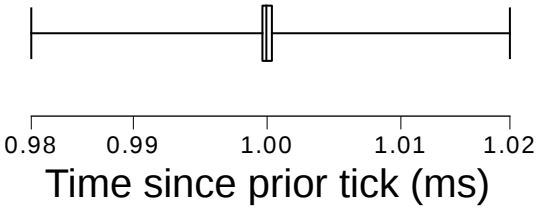

Figure 2. Box-and-whisker plot of the same distribution with a minimum of 0.982 ms and a maximum of 1.018 ms.

3 Methods

All experiments were performed with two HP EliteDesk 800 G2 SFF computers running Ubuntu Server 16.04 LTS and the PREEMPT_RT[RTW [n. d.]; McKenny 2005] patched vanilla Linux kernel (4.4.12). Each system ran on an Intel Core i7-6700 CPU with 64GB of DDR4 memory and a 512 GB Samsung 850 Pro SATA3 solid state drive. Each system was equipped with two PCIe parallel port adapter cards (Startech PEX1P) and a gigabit Intel Ethernet PCIe network card. For the network throughput test, both a gigabit card (Intel 82571EB) connected over CAT6 and a 10 gigabit card (Mellanox ConnectX-2) connected over SFP+ twinaxial direct-attach copper cable were tested. Aside from the USB jitter test, all experiments ran with USB disabled in the kernel and BIOS.

4 Performance Testing

System latency, jittyer, network throughput, and numerical processing capacity were evaluated to demonstrate LiCoRICE's capabilities on common commercial hardware.

4.1 Jitter

Jitter is a measure of inter-tick consistency. The jitter was calculated by the difference of actual tick time from the defined desired tick time. The latency from receiving a signal to outputting that same signal was also measured and confirmed for both analog (line) and digital (parallel port) signals as shown in Figures 3a and Figures 3b, respectively. Worst-case jitter was determined by toggling a parallel port output pin at each tick. The distribution of these times is shown in Figure 2 and all events fall within a maximum jitter level of 18.2 μs with mean 1.000 ms and standard deviation 595 ns. This distribution includes jitter from two realtime systems—the non-LiCoRICE computer preempting the kernel when recording the tick time in addition to the LiCoRICE computer—and is around double the jitter from just one system. Worst-case jitter was also measured on a USB-enabled model for 15 hours from a single system using an oscilloscope readout to be just over 17 μs. This is comparable to around 35 μs maximum jitter for a two-system test.

4.2 Latency

System latency was determined by sending data over parallel port from one realtime computer to a LiCoRICE source. The average latency measured using this method was 1.53 ms. This latency was at least one millisecond and no more than three milliseconds because LiCoRICE latched the parallel port pin state at the beginning of every tick and data from the current tick is not acted upon until the next tick. LiCoRICE then replied over the parallel port within one tick to guarantee timings. This deterministic behavior resulted in the asynchronous latency sweep observed in Figure 3c. There is overhead in the inter-tick time during which the timer updates buffer offsets and checks timing guarantees. On average across one hour, this setup time took 2.98 μs on the same configuration used to determine maximum jitter.

4.3 Network Throughput

To determine maximum data throughput, random data were sent into a model as UDP packets over Ethernet with one source reading in data and and one sink writing it back out. These data were sent from a non-LiCoRICE realtime computer and the volume of data was increased until either 1) system timing guarantees were broken or 2) UDP packets dropped. The default and maximum socket send and receive buffer sizes were increased to 2 GiB. IPv4 UDP buffer sizes were also increased as much as possible. The maximum amount of data processed under 1 ms ticks using a 10 gigabit card was 9.539 Mb (810 packets) in comparison to the gigabit card which was able to process 0.9539 Mb (81 packets) using packets with a UDP payload of 1472 bytes. These rates effectively maximize the respective protocols, demonstrating sustained throughput at saturated network rates.

4.4 Numerical Processing

A key feature of LiCoRICE is its ability to perform numerical operations efficiently. We conducted exhaustive computation on a test model with NumPy operations running primarily on one CPU core. Within a 1 ms tick deadline, the model sustained any one of the following operations: dot product of two random 200×200 matrices, matrix inversion of a 20×20 matrix, matrix pseudo-inverse of a 35×35 matrix. These BLAS/LAPACK optimized operations demonstrate the ease with which high-level numerical code can be executed in realtime.

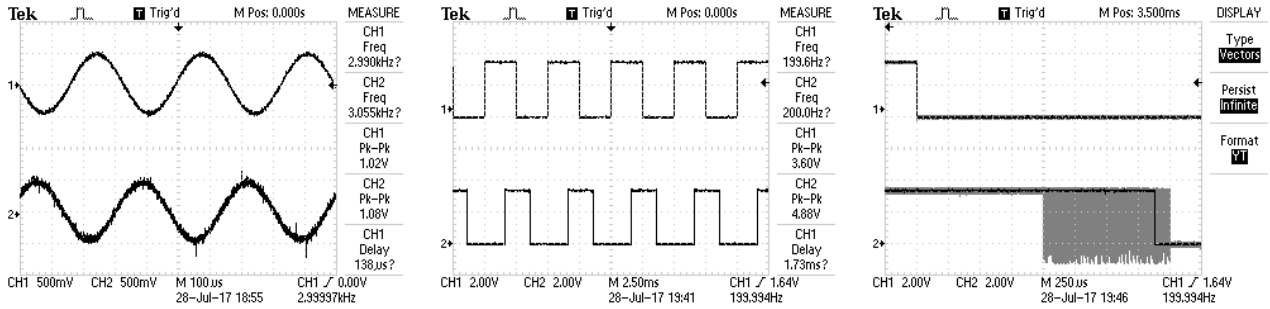

Figure 3. Response (bottom) to analog and digital input (top) from a signal generator. (a) shows a line level input fed into a model replicating the input on a line level output with 138 μs of latency—75 μs of which were manually introduced to buffer samples from the audio card. (b) shows a parallel input fed into a model outputting over parallel as quickly as possible with a switch in output whenever a switch in input is detected. (c) shows the exact same output as (b) over time. Latency sweeps over time due to clock skew between the LiCoRICE model and the signal generator.

5 Discussion

5.1 Assumptions

Some assumptions about the numerical operations performed and the structure of the signals are necessary to meet timing guarantees. All numerical computations must be fixed operations, and dynamic functions are not advised. Despite being NumPy arrays, signals should only comprise Python primitive data types–arbitrary Python objects not supported due to non-deterministic memory needs. Similarly, all signals must have static ndarray dimensions, variable-sized arrays are not supported.

5.2 Applications

Although motivated by and built for the needs of systems neuroscience, LiCoRICE was designed to be as generic and flexible as possible. It has additional potential applications in several fields, spanning control systems, robotics, computer vision, aerospace, automotive, data acquisition, and interactive systems.

LiCoRICE is an open-source alternative to commercial offerings that support realtime operations and model-based design. It doesn't have a GUI to aid in visual wiring of model inputs and outputs like Mathworks' Simulink, but instead currently relies on YAML config files. To our knowledge, LiCoRICE is the first platform to deliver soft realtime guarantees with Python and the only open-source realtime platform supporting complex numerical operations with a high-level programming language.

5.3 Future Work

Modules are currently single-tick in that they perform operations on input signals and must set their output signals within a single clock tick, but it may be useful to support multi-tick modules that are only required respond every N ticks.

LiCoRICE currently runs on conventional multicore x86 architectures, but it may be desirable to support embedded architectures like ARM. With fewer available cores, a realtime-optimized scheduler may be more important to prevent timing violations and guarantee efficient use of resources. In theory, LiCoRICE should support any architecture compatible with the PREEMPT_RT kernel patch.

6 Author Contributions

PM was responsible for writing the LiCoRICE source code and writing the paper. SD was responsible for writing the LiCoRICE source code, writing the paper, conducting testing experiments, and collecting data. SR was responsible for USB support and participated in data collection. PN conceived of the work, participated in writing and reviewing the paper, and provided guidance at all stages of the effort.

7 Acknowledgments

The authors would like to thank Beverly Davis and Margaret Truong for administrative support, Axel Sly and Alexander Sosa for their work on an early monolithic Cython prototype, Philip Levis for helpful discussions on project presentation and communication, and Krishna V Shenoy for supporting the effort in several capacities since its inception.

References

[n. d.]. Embedding Cython. ([n. d.]). https://github.com/cython/cython/wiki/EmbeddingCython

[n. d.]. Linux Foundation Wiki: Real-Time Linux. ([n. d.]). https://wiki.linuxfoundation.org/realtime/start

[n. d.]. NIMH Software Projects. ([n. d.]). https://www.nimh.nih.gov/labs-at-nimh/research-areas/clinics-and-labs/ln/shn/software-projects.shtml

[n. d.]. Numba - Compiling code ahead of time. ([n. d.]). http://numba.pydata.org/numba-doc/dev/user/pycc.html

[n. d.]. Reflective Computing. ([n. d.]). http://reflectivecomputing.com

Paul McKenny. 2005. A realtime preemption overview. (Aug 2005). https://lwn.net/Articles/146861