OPEN ACCESS
IOP Publishing

Journal of Neural Engineering

J. Neural Eng. **14** (2017) 045004 (16pp)

https://doi.org/10.1088/1741-2552/aa7526

Falcon: a highly flexible open-source software for closed-loop neuroscience

Davide Ciliberti[1,2,3] **and Fabian Kloosterman**[1,2,3,4,5]

[1] Neuro-Electronic Research Flanders (NERF), Leuven, Belgium
[2] Brain & Cognition Research Unit, KU Leuven, Belgium
[3] VIB, Leuven, Belgium
[4] IMEC, Kapeldreef 75, 3001 Leuven, Belgium

E-mail: fabian.kloosterman@nerf.be

Received 29 December 2016, revised 4 May 2017
Accepted for publication 26 May 2017
Published 21 June 2017

Abstract

Objective. Closed-loop experiments provide unique insights into brain dynamics and function. To facilitate a wide range of closed-loop experiments, we created an open-source software platform that enables high-performance real-time processing of streaming experimental data. *Approach.* We wrote Falcon, a C++ multi-threaded software in which the user can load and execute an arbitrary processing graph. Each node of a Falcon graph is mapped to a single thread and nodes communicate with each other through thread-safe buffers. The framework allows for easy implementation of new processing nodes and data types. Falcon was tested both on a 32-core and a 4-core workstation. Streaming data was read from either a commercial acquisition system (Neuralynx) or the open-source Open Ephys hardware, while closed-loop TTL pulses were generated with a USB module for digital output. We characterized the round-trip latency of our Falcon-based closed-loop system, as well as the specific latency contribution of the software architecture, by testing processing graphs with up to 32 parallel pipelines and eight serial stages. We finally deployed Falcon in a task of real-time detection of population bursts recorded live from the hippocampus of a freely moving rat. *Main results.* On Neuralynx hardware, round-trip latency was well below 1 ms and stable for at least 1 h, while on Open Ephys hardware latencies were below 15 ms. The latency contribution of the software was below 0.5 ms. Round-trip and software latencies were similar on both 32- and 4-core workstations. Falcon was used successfully to detect population bursts online with ~40 ms average latency. *Significance.* Falcon is a novel open-source software for closed-loop neuroscience. It has sub-millisecond intrinsic latency and gives the experimenter direct control of CPU resources. We envisage Falcon to be a useful tool to the neuroscientific community for implementing a wide variety of closed-loop experiments, including those requiring use of complex data structures and real-time execution of computationally intensive algorithms, such as population neural decoding/encoding from large cell assemblies.

Keywords: closed-loop neuroscience, multi-threaded software, real-time processing, online burst detection, open-source software

(Some figures may appear in colour only in the online journal)

[5] Author to whom any correspondence should be addressed.

Original content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.

Introduction

Neural dynamics are highly non-stationary and exhibit transient phenomena. Transient events can be observed at cellular and network levels of the nervous system, such as action potentials or short-lasting oscillatory patterns in the extracellular

field potential (e.g. hippocampal sharp-wave ripples [1]). Neither experiments adopting traditional *open-loop* stimulus-response paradigms, nor internal-state analysis of spontaneous neural activity that only identifies their correlative aspects, can assess the computational role of transient neural events. In contrast, experiments adopting a *closed-loop* paradigm, in which the stimulus is conditional on the state of neural activity [2–4], can causally determine the contribution of transient events to neural computations [5–10]. Closed-loop approaches are also relevant in clinical settings, like neuroprosthetics [11], detection of seizure in epileptic patients [12] and adaptive deep brain stimulation protocols for Parkinson's disease [13].

In its broader meaning, closed-loop neuroscience covers any interaction of the neural system with the outside world [14]. Thus, it also includes experiments in which the input is not of neural origin, e.g. a behavioral measurement (obtained with video-monitoring or specific sensors), or in which the output is not directed to nervous tissue but to the outer world, e.g. presentation of visual stimuli conditional on events in the LFP signal [15] or brain–computer interface (BCI) applications [16].

While the nature of the input and output can be diverse, all closed-loop systems need a real-time processing component that timely detects the occurrence of transient events of interest [14] and responds within a specified time ('deadline') [17, 18]. Hardware solutions based on integrated circuits, field-programmable gate-arrays (FPGAs), digital signal processors (DSPs) or other custom-built hardware [19] are well suited for real-time processing, as the computational time can be made deterministic and hard real-time requirements (in which a missed deadline is considered a system failure) can be met [18]. Nonetheless, the diversity of closed-loop experimental algorithms [4] and conditions (e.g. integration of video-monitoring with multi-channel electrophysiological recordings [5]) requires not only computational speed but also high flexibility. When flexibility is critical, software solutions are more advantageous over purely hardware solutions, as new algorithms can be quickly changed by simply recompiling or running a new version of the code. Importantly, as a closed-loop experiment is intrinsically defined by its real-time algorithm, cross-laboratory sharing of such algorithm is much easier and facilitates experimental reproducibility. Moreover, maintenance and development are more tractable than with purely hardware solutions.

In software, hard real-time requirements can be met only by using a real-time operating system (OS). However, soft real-time requirements, in which deadlines can be missed occasionally [20], can be met on a standard OS and are sufficient for many applications. To achieve these requirements when running computationally intensive algorithms, software design patterns must harness parallelization. A software design pattern that uses parallelization can rely on multi-threading, i.e. the ability of a CPU to handle multiple threads of execution [17], and leverage the presence of more than one physical core inside modern CPUs to improve throughput and latency [21].

To facilitate the implementation of customized closed-loop experiments with different computational loads, flexibility in the software design is central. Yet, most open-source software solutions have been developed around specific applications, like EEG-based BCIs [22–26] or cellular electrophysiology [27, 28], and cannot be easily extended outside the original scope. For instance, EEG- and ECoG-based BCI software platforms generally process signals acquired at low sampling rate (<2 kHz) and process data in batches of hundreds of milliseconds. These platforms, however, do not focus on support for closed-loop experiments that require millisecond-scale feedback given streaming data at higher sampling rate (>20 kHz). On the other hand, software solutions for cellular electrophysiology can process signals sampled at high rate with very low latencies (<10 μs), but offer little support for parallel processing of multi-channel signals (e.g. RTXI [28] has only a single real-time thread).

Following a different approach, the hardware-software platforms NeuroRighter [29] and Open Ephys [30] are primarily designed for data acquisition, but also support closed-loop applications (dynamic clamp; real-time detection of behavior, neural oscillations or individual spikes). However, such platforms still have important limitations in terms of flexibility as they are tailored to specific hardware, lack the ability of the user to have direct control over the CPU resources (thus potentially hampering performance) and limit the implementation of new processing elements to the use of only pre-determined data types.

To meet the needs of a highly versatile closed-loop software, we designed Falcon, a highly flexible open-source software platform for real-time analysis of streaming experimental data. Falcon is built as a client-server application and uses a graph-based definition for implementing processing graphs in which each node is mapped to a separate thread of execution (giving the user direct control over CPU resources). Falcon comes equipped with basic processing nodes (e.g. spike detection and digital filtering), basic data types (e.g. MultiChannel and Event data) and its design makes it easy to implement new processing nodes and data types for customized closed-loop experiments.

Here we give a description of the software architecture of Falcon and demonstrate its real-time processing capabilities by showing sub-millisecond round-trip latency in processing 128 channels streaming data at 32 kHz. As an example of closed-loop neuroscience application, we also show how Falcon was used to detect in real-time population bursts from the hippocampus of a freely moving rat.

Methods

Software architecture

Falcon was conceptualized as a general-purpose soft real-time processing software that operates on incoming data streams and produces outputs that can be used in closed-loop experiments (figure 1). Falcon is written in the object-oriented and compiled C++11 language and it runs on Linux-based computer systems. Its modular architecture facilitates maintenance and development of new features. The source code is made available under GPL3 license at http://bitbucket.org/kloostermannerflab.

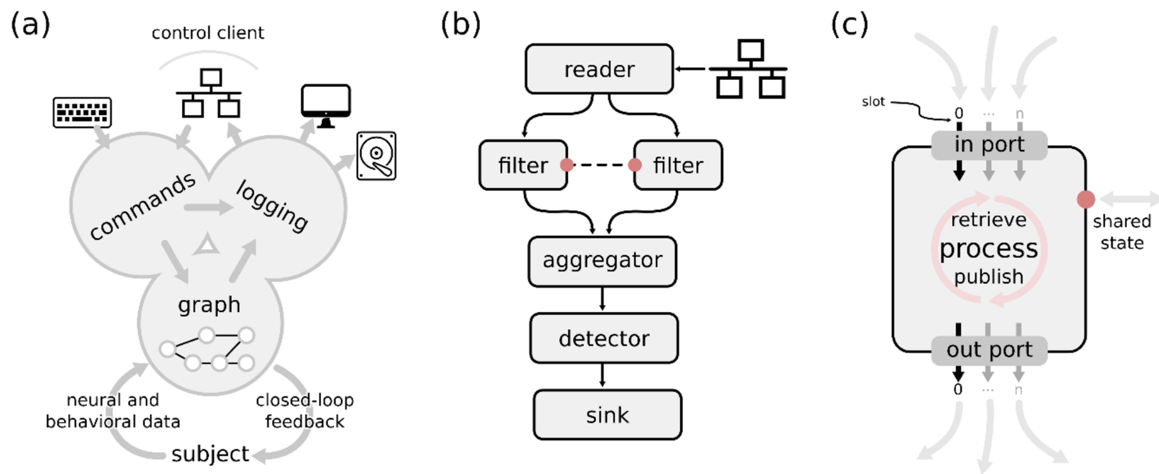

Figure 1. Overview of Falcon software. (a) Falcon is based on a client-server architecture. The Falcon server is composed of three main subsystems. The *graph* subsystem manages the construction and execution of a user-defined processing graph that analyzes incoming behavioral and neural data streams and provides closed-loop feedback. The *commands* subsystem listens for and acts on user commands originating from the keyboard or remote client. Finally, the *logging* subsystem logs all internal messages (information, debug, warnings, updates and errors) to a number of destinations, including screen, disk and network. (b) Example graph with a reader node that receives data over the network and produces two data streams. The two streams pass through filter nodes, the outputs of which are combined into an aggregator node. Finally, the aggregated data stream passes through a detector and the generated event is absorbed by a sink. The two filter nodes share a state value (red dots connected by dashed line). (c) A processing node retrieves data packets arriving on its input port(s) and publishes processed data packets on its output port(s). Each port handles data streams with identical data type through one or more slots. A processing node can also share state values with other nodes. State values may also be read and/or modified by external clients and/or by other nodes.

By design, Falcon is dedicated to the processing of data streams and is separated from (graphical) user interfaces in a client-server architecture. Clients communicate with Falcon over network connections to control and monitor data processing or visualize intermediate and final outputs. Clients can be written in any programming language with support for the ZeroMQ messaging library [31].

Internally, Falcon is composed of three interacting subsystems (figure 1(a)): a *commands* subsystem that receives and parses external commands; a *logging* subsystem that relays internal status and error messages to the outside; a *graph* subsystem that constructs and executes a flow graph for data processing. The flow graph is composed of processing nodes and directed data links between the nodes (figure 1(b)). To take advantage of parallel and concurrent computing and give the user maximum control of CPU resources, each Falcon node is mapped to a logical thread of execution.

Processing nodes

The building blocks of a processing graph are nodes that perform an operation on one or more incoming data streams and produce one or more outgoing data streams (figure 1(c)). Data streams flow to/from processing nodes through input/output ports. Each port manages one or more streams (connected to separate slots on the port) that all share the same data type. A port either has a predefined number of slots, or it dynamically expands the number of slots depending on how many data streams are connected to it. Processing nodes with no input ports act as data sources; for example, they read data from a network socket or a file. Nodes with no output ports act as data sinks; their processing may involve serialization to disk or network or simply logging the arrival of specific data of interest. Finally, nodes with at least one input and one output port act as filters or detectors. Whereas filters generally transform continuous input data streams into different data streams, detectors identify the occurrence of a transition in the neural/behavioral input stream(s) that is relevant for a closed-loop feedback.

Data streams consists of data packets that flow between nodes through concurrent ring buffers. The ring buffers are owned by the output slots and contain pre-allocated data packets. Importantly, the ring buffer guarantees thread-safety and prevents other synchronization errors (like dead-locks) inside the Falcon process. Multiple downstream input slots can connect to the same ring buffer to receive the same data stream (i.e. single producer, multiple consumer). A hierarchical type system is defined for data packets that facilitates the validation of connections between processor nodes based on data type. Currently, four data types and corresponding data objects have been defined: MultiChannelData (containing an array of N samples for C channels), EventData (containing an event string), SpikeData (containing spike times and peak amplitudes) and MUAData (containing the number spikes in a predefined time bin).

Processing nodes may also share state values with each other and with external clients. State values were implemented as atomic variables to prevent data races across threads. Access to shared state values is governed by read/write permissions. Shared states are equivalent to unbuffered data links and can be used for sharing slow-changing variables inside and outside the Falcon server.

We have implemented a small library of processing nodes that support basic processing of electrophysiology signals, like digital signal filtering (supporting FIR filters and IIR filters in biquad cascade), down-sampling, and detection of spikes or

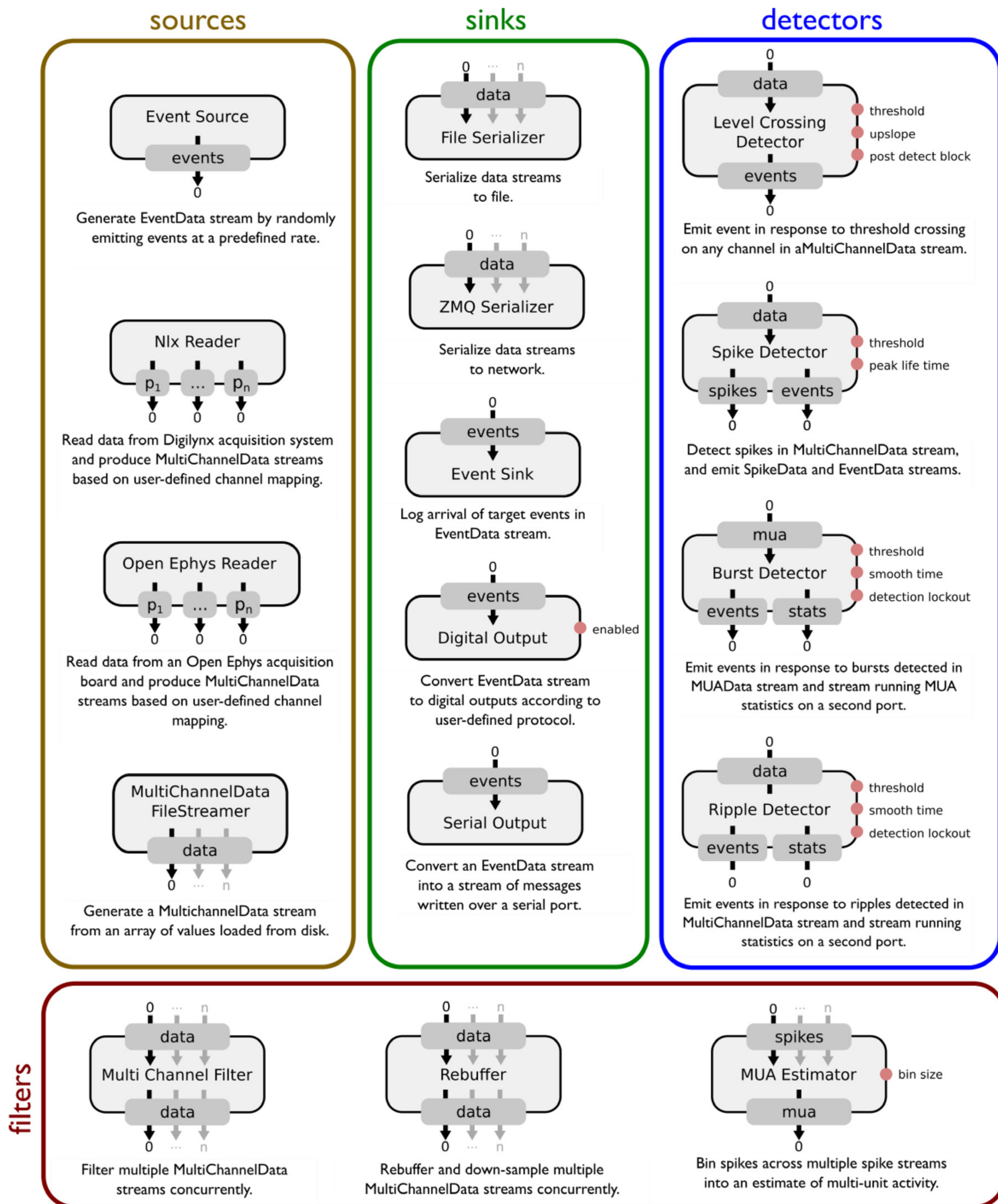

Figure 2. Library of available Falcon processors. Processors can be functionally classified as data sources (with no input ports, they read input data from network or disk), data sinks (with no output ports, they consume incoming data by producing a digital output for closed-loop feedback, serializing data to disk or network or logging information about the incoming data), filters (they transform a continuous data stream on the input into a different continuous data stream on the output) and detectors (they transform an incoming continuous data stream into an irregular stream of events of experimental interests, e.g. 'population burst'). Not shown: Dummy Sink (for testing purposes).

bursts (figure 2). For each data type, we also implemented a set of test sources that generate a stream of the specific data type (e.g. EventSource streams user-defined EventData at a fixed rate).

Real-time processing pipelines that only require the available processing nodes can be easily configured without the need for recompilation of Falcon software. New processing nodes and data types can be added by those familiar with C++11 programming. This is made very straightforward by the design pattern utilized in Falcon: in fact, a new data or processor class can be written by simply deriving the existent parent class (IData or IProcessor) and overriding the virtual methods.

Processing graph

Falcon constructs and configures the processing graph according to a user-provided description of the processing

nodes and their interconnections written in human-readable YAML language (figure 3). In the graph definition, each processing node is identified by a unique name and class and is further parameterized by a set of options and initial state values. The graph definition has a convenient and compact syntax to define connections between output and input data ports. For ports with a variable number of slots, new connections automatically create additional slots, unless a specific slot number is requested.

At any given time, the graph subsystem is in one of the following three states: 'no graph', 'ready' and 'run'. Transitions between these states are triggered by user commands (figure 4).

Initially, the graph subsystem has no processing graph defined and is in the 'no graph' state. Following a request to build a new graph, the subsystem will parse the graph definition and construct the processing nodes. The nodes then have the opportunity to configure themselves and create their input and output ports. Next, Falcon routes the data streams between output and input ports after a check for data type compatibility and links shared state values. During a subsequent negotiation phase, the parameters of a data stream (e.g. stream rate) and the data packet type (e.g. number of channels and samples) are fixed. This phase is followed by the creation of ring buffers that hold the data packets that will be streamed between connected nodes. After all nodes are constructed, configured and connected, each node is given the opportunity to do a one-time preparation (e.g. to load resources or allocate data structures). Destruction of a graph follows the reverse sequence, with first a per-node clean-up and a final destruction of connection and node objects.

Once a graph is built, a processing run can be initiated and terminated through start/stop commands. When a run is started, node threads are started and after a per-run preprocessing step they wait for a go signal to enter their main processing loop. As soon as a run is terminated, node threads are signaled to break out their processing loop, perform postprocessing and exit. The pre- and post- processing steps allow the node to execute pre-computations that are independent of the input data stream.

Parallelism and concurrency

Splitting the computation across multiple connected nodes in a graph has the benefit of modularization and flexibility that enables simple construction of new processing pipelines from a basic set of nodes. This flexibility is further enhanced by the implementation of algorithmic building blocks (e.g. FIR filter, threshold-crossing detector) that form the basis of node computations.

The modularity provided by the nodes allows Falcon to leverage the power of multi-core processors by mapping each node in the graph to a separate thread of execution. Whereas multiple threads on a single core allow multiple tasks to run concurrently, threads executed across different cores run in parallel. Thus, on multi-core CPUs multi-threading helps to speed up data processing. This allows the system to not only keep up with the incoming

```
processors: (A)
 source:
  class: NlxReader
  options:
   channelmap:
    tt1: [0,1,2,3]
    tt2: [4,5,6,7]          (B)

 filter(1-2):
  class: MultiChannelFilter
  options:
   filter:                  (C)
    file: repo://tests/filters/spike.filter

 spikes(1-2): (D)
  class: SpikeDetector
  options:
   threshold: 100.0 # in microVolts
   invert_signal: true
   buffer_size: 1 # in milliseconds

 mua:
  class: MUAEstimator
  options: {bin_size_ms: 30}

 bursts:
  class: BurstDetector
  options:
   threshold: 16
   smooth_time: 15 # seconds
   enabled: true

 eventfile:
  class: FileSerializer (E)
  options: {encoding: yaml, format: full}

connections: (F)
 - source.tt1=filter1.data
 - source.tt2=filter2.data
 - filter(1-2).data=spikes(1-2).data
 - spikes1.spikes=mua.spikes.0
 - spikes2.spikes=mua.spikes.1
 - mua.mua=bursts.mua
 - bursts.events=eventfile.data

#states: (G)
#  - [node1.state, node2.state]
```

Figure 3. Example of a configuration file defining a Falcon graph in YAML language. A graph definition has three sections that define the processing nodes, the connections among nodes and the state connections. The processors section (A) contains a map of processor node definitions, including the type of node (class) and configurable options. For example, the source node (B) is of class NlxReader: it implements a processor that reads neural data from a Neuralynx DigiLynx acquisition system. Its channelmap option determines how signal channels are mapped to the output streams of the node. Two output ports 'tt1' and 'tt2' are listed, with 'tt1' streaming data from channels 0, 1, 2 and 3. The filter nodes are of class MultiChannelFilter which implements a processor that applies a digital (FIR/IIR) filter to incoming multi-channel data streams. The filter coefficients are read from a file named 'spike.filter' (C). Note that here the file path is specified as a URI, in which repo:// has been configured to point to the local GIT repository (the loaded filter, in this case, is part of the open-source Falcon repository). In (D), two nodes of class SpikeDetector are created: in the options section, the user can set the threshold for spike detection, the buffer size and the use of inverted or non-inverted signals for detecting spikes. As for the filter nodes, two identical nodes are conveniently defined at once by appending a range of numbers between brackets. By virtue of their data type, individual data packets in streams know how to serialize their contents. A generic FileSerializer node (E) takes advantages of this capability and can be used to save data streams to disk. The connections section (F) contains a list of links between output and input ports, and their slots. Finally, the states section (G) contains a list of processing node states that should be linked (section is commented because there are no states that need to be linked in this particular graph).

data stream, but also to reduce closed-loop response latencies to values compatible with the constraints of the experiment [21].

Depending on the available hardware (i.e. number of CPUs and cores) and the application, computations in Falcon can either be split across many nodes (and threads) or grouped within a single node. For example, a graph may contain a single node that filters multiple data streams concurrently (e.g. MultiChannelFilter in figure 2), or many such nodes that filter data streams in parallel.

The average time required for processing incoming data should always be less than the total available compute time (the inverse of the data stream rate). If this basic requirement is violated, either there will be a growing backlog of data leading to increased latencies and eventually a break-down or stalling of the system, or data items will be dropped. The ring buffers between processing nodes serve to decouple the nodes and reduce the impact of fluctuations in processing time and stream rate that could lead to backlogs or missed data items. In Falcon, ring buffer sizes are configurable and high-water level indicators help to pinpoint potential weak spots in the processing graph.

Shuttling data packets between two connected processing nodes through a ring buffer may incur a significant cost (relative to the available time budget) if it is performed at high rate. When needed, this cost can be reduced at the expense of overall response latency by buffering multiple samples together and reducing the rate of communication between processors. For example, the NlxReader node receives data samples from a Neuralynx DigiLynx acquisition system at a rate of 32 kHz and collects batches of multiple samples (batch size is user configurable) before passing them on to downstream processing nodes. Dispatching of signals in smaller buffer sizes increases the probability of missing incoming data packets from the network.

On the consumer side of a ring buffer different strategies can be used to wait for new data items, the choice of which impacts performance by balancing thread contention and CPU resources [21]. The ring buffer in Falcon supports five waiting strategies that can be set separately for each data stream output port:

1. thread blocking using a condition variable and a lock (default),
2. sending a thread to the end of the scheduler's queue (yielding),
3. polling (repeated query of the input port),
4. sleeping (wait for a small amount of time),
5. a progressive combination of polling/yielding/sleeping.

These strategies can be effectively deployed to process, for instance, streaming data that arrive at different rates on separate ports inside one node.

Further fine-tuning and optimization of the real-time performance of a graph is possible by altering how a node's thread is handled by the OS scheduler. Both the thread priority and the thread affinity (i.e. bind thread to designated CPU) are configurable options in Falcon. In combination with CPU isolation, the thread affinity setting can make sure that a single CPU is dedicated to a single thread with demanding or critical processing needs (e.g. source node reading from a network socket).

Clients

A small set of keyboard commands enables basic user interaction with Falcon, including starting and stopping the execution of a graph. Full control (e.g. uploading and constructing new graphs, retrieving and setting state values, etc) is available for local and remote clients that communicate with Falcon using a request-reply messaging protocol. Clients can also subscribe to logging and update messages that Falcon broadcasts over the network. The ZeroMQ messaging library [31] that is used internally has many language bindings and thus clients can be written in the preferred language of the user. We have written a small Python module that interfaces with the Falcon server and which can be used as a foundation for custom, application specific clients. On top of this library, we have created a simple reference graphical client, that exposes many of the available commands to the user. We have also build simple visualization clients for live monitoring of a node's output data streams that are broadcast to the network using the ZMQ Serializer node (figure 2).

Testing hardware

Falcon was compiled with the GNU g++-5 compiler on a PC running 64-bit Linux Mint 18 (kernel version: 4.4.0_2.1 generic). We used both a 32-core and a quad-core machine for the latency tests, while the live animal tests were conducted using the 32-core machine only. The 32-core machine was equipped with 256 GB of RAM, 40 MB of smart cache and two 16-core CPUs (Intel Xeon(R) CPU E5-2698 V3 @ 2.30 GHz) with hardware-based simultaneous multi-threading (Hyper-Threading) [32] enabling a total of 64 virtual cores. The quad-core machine had a single CPU (Intel Core i7-4790 @ 3.60 GHz) using Hyper-Threading (for a total of 8 virtual cores), 16 GB of RAM and 8 MB of smart cache. Interference from the OS scheduler was limited by setting the priority of the Falcon executable to 'real-time' (the highest possible). We also pinned the data reader thread to an isolated virtual core to reduce the probability of missing incoming data packets.

Latency tests were carried out using both Neuralynx and Open Ephys data acquisition hardware. In tests with Neuralynx hardware, multi-channel digitized signals were acquired through a 128-channel DigiLynx data acquisition system (Neuralynx, Bozeman, MT) and sampled at 32 kHz. The first data port of DigiLynx was connected to a workstation running Cheetah (Neuralynx). Cheetah was used to configure the DigiLynx system and to acquire, visualize and save data. The second data port was used to stream UDP data packages (each containing a single sample for all 128 channels) into the machine running Falcon. The Linux machines were equipped with a dedicated Ethernet card (Intel PRO/1000PF Dual Port Server Adapter PCI-E, Fiber Optic Network Adapter). An alternative way to access data streams from the Digilynx system is by using Neuralynx'.NET-based NetCom API, which interfaces with the Cheetah software. However,

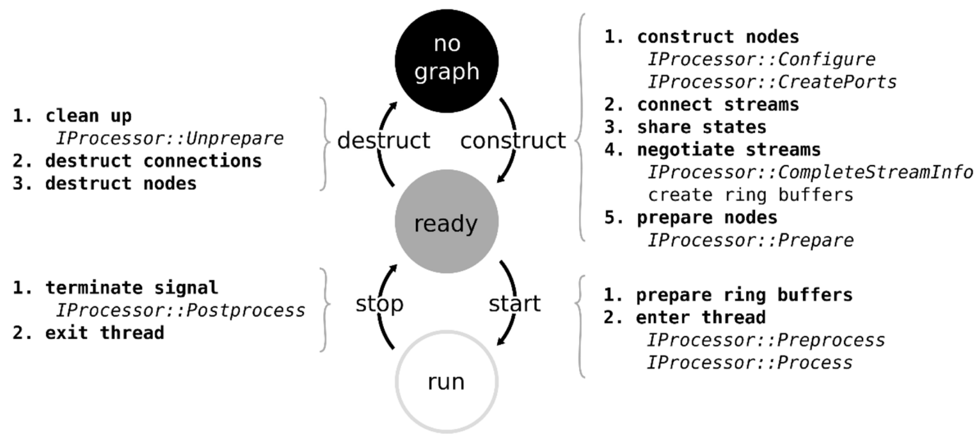

Figure 4. Diagram of graph subsystem states and transitions. For each transition, the main sequence of events is listed including the methods implemented by processing nodes (in italic).

because multi-channel data is buffered inside Cheetah in 512-sample blocks, this approach limits the minimal round-trip latency to 16 ms [33]. For this reason, we read data directly from the Digilynx Ethernet port without using NetCom.

Signals were recorded either from an analog signal generator (Square B&K Precision, Model 3003) connected to a Signal Mouse (Neuralynx, Bozeman, MT) or from an array of tetrodes implanted in the hippocampus of a freely moving rat (see Experimental Methods). In all tests on Neuralynx hardware, analog signals were buffered using a unity-gain analog pre-amplifier (HS-36, Neuralynx) before digitization in DigiLynx.

Latency tests on Open Ephys hardware used an Open Ephys acquisition board with the USB2.0-compatible Opal Kelly XEM 6010-LX45 FPGA module, a 6-foot SPI interface cable and a 32-channel Intan headstage. Implementation of the Open Ephys reader in Falcon used the available Intan libraries and was based on the implementation in the open-source Open Ephys GUI [30]. Signals were sampled at 30 kHz.

Digital output pulses were generated by an isolated Digital Input/Output (DIO) module (USB-4750, Advantech Benelux, Breda, The Netherlands) communicating with Falcon over universal serial bus (USB 2.0) using the DigitalOutput node (figure 2) that interfaced with the vendor supplied device driver. A SerialOutput node communicating with an Arduino Uno microcontroller board could be used as an alternative for generating a digital output pulse. As compared to the Advantech DIO module, round-trip latency was approximately 1 ms higher (data not shown) and hence we did not use the Arduino Uno in any of the tests.

Latency measurements

To measure the round-trip latency of our Falcon-based closed-loop system, we input a square wave signal (47 Hz for tests on Neuralynx hardware and 25 Hz for tests on Open Ephys hardware; 50% duty cycle; 1.8 mV_{pp} amplitude after 1000× amplitude reduction by Signal Mouse) to all test channels of the acquisition system (128 channels for Neuralynx, 32 channels for Open Ephys). For each test a total of 20 586 measurements were collected and analyzed.

Falcon executed a graph implementing a simple threshold-crossing algorithm that detected the rising edge of the square wave. For each detected edge, Falcon sent a digital output pulse back to the DigiLynx acquisition system, the occurrence of which was timestamped and recorded together with the original square wave in Cheetah (figure 5(a)). For tests with Open Ephys hardware, the generated analog square wave was routed using a BNC connector to both a Neuralynx and an Intan headstage via two Signal Mouse devices.

All ring buffers in Falcon were set to communicate using the blocking strategy to limit CPU usage. This strategy was also shown to be efficient in other multi-threaded software for closed-loop applications [21]. We measured the used system memory as reported by the System Monitor program of Linux Mint.

Round-trip latency estimates the overall time lag between the occurrence of an event in the monitored signals and the time at which a feedback stimulus could be applied (e.g. by a stimulator or an actuator). By the same token, round-trip latency includes not only the delay introduced by the data processing and transmission within Falcon, but also any latency due to the data transmission over the hardware (e.g. delay of the UDP socket or of the transmission of the output TTL). To better characterize the specific contribution of Falcon, we also measured the internal software latency that reflects the time needed for moving data inside a Falcon graph from source to sink (figure 5(b)).

Round-trip latency was measured as the time difference between the onset time of the square wave and the Falcon-generated TTL pulse recorded in Cheetah. The rising edge of each square wave was determined offline from the recorded data as the timestamp of the first sample crossing the threshold of 100 μV on any of the test channels. For each round-trip latency measurement, we additionally timestamped in software the arrival of each UDP packet and the time before the generation of the TTL pulse on the DIO card using a monotonic nanosecond-resolution internal clock. The internal software latency was then measured as the difference between these two timestamps. By subtracting the software latencies triggered only by the last sample loaded on the buffer ('last-in' samples) from the corresponding round-trip latencies, we

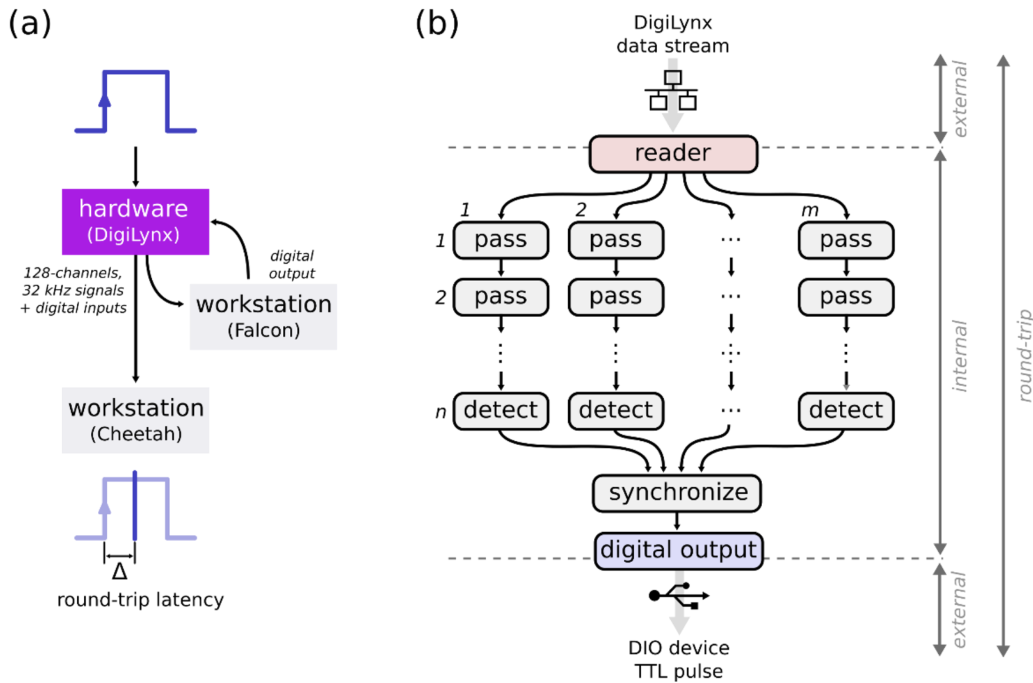

Figure 5. Hardware-software setup used for the measurement of the round-trip latency with Neuralynx hardware. (a) A square wave was input to all test channels. Signals were digitized with a 128-channel DigiLynx acquisition system and recorded with a workstation running Cheetah. Signals were routed to a workstation running Falcon and producing a TTL pulse as closed-loop feedback to the acquisition system. Feedback was triggered by the rising edge of the square wave. The time difference between the generation of the digital output and the time of the rising edge was taken as a measure of round-trip latency. (b) Round-trip latency tests were executed with graphs of variable size ($m = \{1, 32\}$ parallel and $n = \{1, 8\}$ serial stages). Each graph had a reader node, zero or more all-pass digital filters, at least one detector of rising edges, one processor synchronizing the detection events and a digital output sink controlling a USB DIO card. Latency was tested on the external, internal and round-trip paths indicated on the right. Data transfer speed on the external paths is dependent only on the hardware, while data transfer speed on the internal paths is dependent only on the software.

computed the distribution of the external hardware contribution of the round-trip latency.

For all latency tests, incoming samples were collected on a reader node in batches of 6 before being pushed through the remainder of the processing graph. This buffering was needed to reduce the probability of missing incoming UDP data packets.

To get an estimate of the latencies for graphs of varying size, we built different processing graphs that split an incoming stream of multi-channel data into n parallel pipelines each with m serial stages (figure 5(b)). All but the last serial stage was a node that passes the input stream unmodified to the output stream (all-pass FIR filter). The last serial stage was a detector that emitted events on its output when the square wave input signal crossed the threshold. The parallel pipelines fed into a single synchronization node that triggered the digital output pulse as soon as all pipelines had detected the threshold crossing. In all graphs tested, we used one additional parallel pipeline from the reader node to serialize to disk the data from a single channel using a File Serializer node (not shown in figure 5(b)).

To minimize the influence of the OS in scheduling the execution time of the threads, we set the priority of the Falcon executable to the highest level (‘real-time’). To reduce the probability of missing incoming data packets over network, besides using a 6-sample buffer, we pinned the reader node thread to an isolated virtual core: the isolation guaranteed that the reader node would always run on the same virtual core and that no other threads in the system would compete with the reader thread. The probability of missed packets was generally very low, but they did occur occasionally in our tests and more often so when the number of processing threads was high compared to the number of available cores. All tests presented in Results completed without missed packets.

Experimental methods

For recording of hippocampal multi-unit activity (MUA), a dual-bundle 3D-printed micro drive array [34] carrying 20 tetrodes (bundles of four wires) and bipolar stimulation electrodes was implanted in one male Long-Evans rat. Briefly, following sterile surgical procedures the rat was anesthetized with isoflurane and mounted in a stereotaxic frame. After exposure of the skull, anchoring screws were inserted and burr holes were drilled for access to the hippocampus. After cementing the micro-drive array in place, the rat was allowed to recover for at least 4 d before recordings started. All experimental procedures were approved by the animal ethics committee at KU Leuven.

Following recovery from surgery, most tetrodes were lowered to the cell layer of hippocampal area CA1. One tetrode was positioned in the white matter and used as reference. Recordings lasted 53 min and occurred during a resting phase in which the animal was placed in a familiar sleep box. Seventeen tetrodes with unit activity were used for recordings.

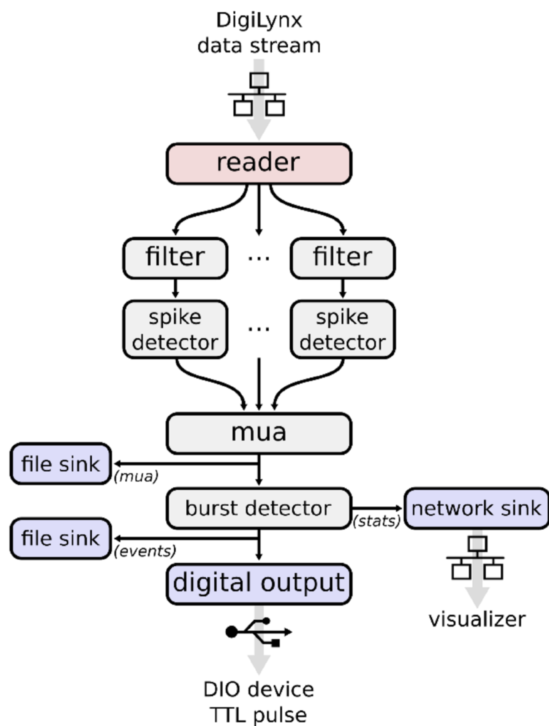

Figure 6. Falcon processing graph for real-time detection of population bursts. A reader node parses incoming streaming neural signals and dispatches them to a set of parallel pipelines for high-pass filtering and spike detection; each pipeline is processing channels coming from a single tetrode. Spike detectors provide the number of spikes detected on the tetrode in a user-defined buffer. Spiking data generated by each pipeline is synchronized into a mua estimator node which computes the multi-unit activity with a user-defined buffer size (greater than the buffer size used for spike detection). MUA data generated by the mua node is passed to a burst detector node and to a disk serializer sink. The burst detector node streams its internal statistics about the ongoing μ and *mad* to a network serializer sink to which a Python client connects. This client displays to the user the updated value of the variables. The visual feedback guides the user in adjusting the parameters that control the online algorithm of burst detection (like the factor used for threshold crossing). As soon as the online algorithm detects a burst, the burst detector node generates an EventData item marked by a 'burst_detection' string to both a disk serializer sink and a sink that controls the digital output of the closed-loop system.

Online detection of multi-unit activity bursts

To detect bursts in real-time, we loaded into Falcon a graph (figure 6) composed of the following elements:

- a reader node of class NlxReader dispatching multi-channel data on 17 output ports that each served data from one tetrode sampled at 32 kHz;
- a pre-processing block of spike detection composed of 17 parallel pipelines, each pipeline composed of a filter node of class MultiChannelFilter (Bessel 4th order, 600–6000 Hz in biquad cascade) and a spike detection node of class SpikeDetector operating with a 1 ms buffer;
- a MUA node of class MUAEstimator computing the aggregated MUA signal with a buffer of 10 ms;
- a burst detector node of class BurstDetector implementing a threshold-based algorithm for burst detection suitable for real-time processing;
- a digital node of class DigitalOutput that controlled the USB DIO card for closed-loop feedback.

In the online burst detection algorithm, an individual burst was detected as a threshold crossing in the MUA signal according to $\text{signal} - \mu > \text{factor} * mad$, where μ and *mad* correspond to the running estimates of the mean and mean absolute deviation respectively, and factor is a user-defined multiplier. In our test, factor was set to 4 and adjusted online to 8 (for ~150 s) and back to 4 during the course of the experiment. Running estimates of μ and mad were computed using an exponentially weighted moving average filter [5]. The span of the filter was set to 22.5 s (2250 samples), corresponding to a half-life of 7.8 s; this value was selected manually to fit the slow changes in the MUA signal. Detection rate was limited to 10 Hz to simulate a maximum feedback stimulation frequency (as it would be the case in an actual disruption experiment).

Offline detection of multi-unit activity bursts

Spiking activity and digital burst detection pulses from Falcon were recorded simultaneously using Cheetah data acquisition software (Neuralynx, Bozeman, MT). A smoothed histogram (1 ms bins, Gaussian kernel with 15 ms standard deviation) of MUA was constructed using all recorded spikes. Slow non-burst fluctuations were removed from the MUA signal by detrending with an exponentially weighted moving average filter applied forward and backwards (span = 7.5 s (750 samples); corresponding to half-life of 2.6 s). Mean and standard deviation were calculated on the detrended MUA signal. MUA bursts were defined as periods in which MUA was higher than 1.5 times the standard deviation. Burst onset and offset times were defined as the times in which MUA was higher than its mean at the closest time before and after the time of threshold crossing. If the time difference between two bursts was less than 30 ms, the two events were merged to form a single burst event.

Results

Round-trip latency

The real-time processing component of closed-loop systems must be able to reliably produce a response within a maximum predefined time (deadline) when an event of interest is detected [20]. Response latencies should therefore be guaranteed to be below a certain critical value, which depends on the specific application. For instance, while a value of 30–40 ms might be acceptable for detecting transient oscillatory patterns that last 70–100 ms, like hippocampal sharp-wave ripples [6, 35], the same value will be too high for a feedback based on the occurrence of an action potential.

Falcon was conceived as a general-purpose real-time software framework. It is therefore not ideal to characterize its response time performance on the basis of a specific application. For a given closed-loop application, the response latency is affected by the algorithm (algorithmic latency) used for event detection and by the computational latency introduced by the closed-loop framework. To characterize Falcon's

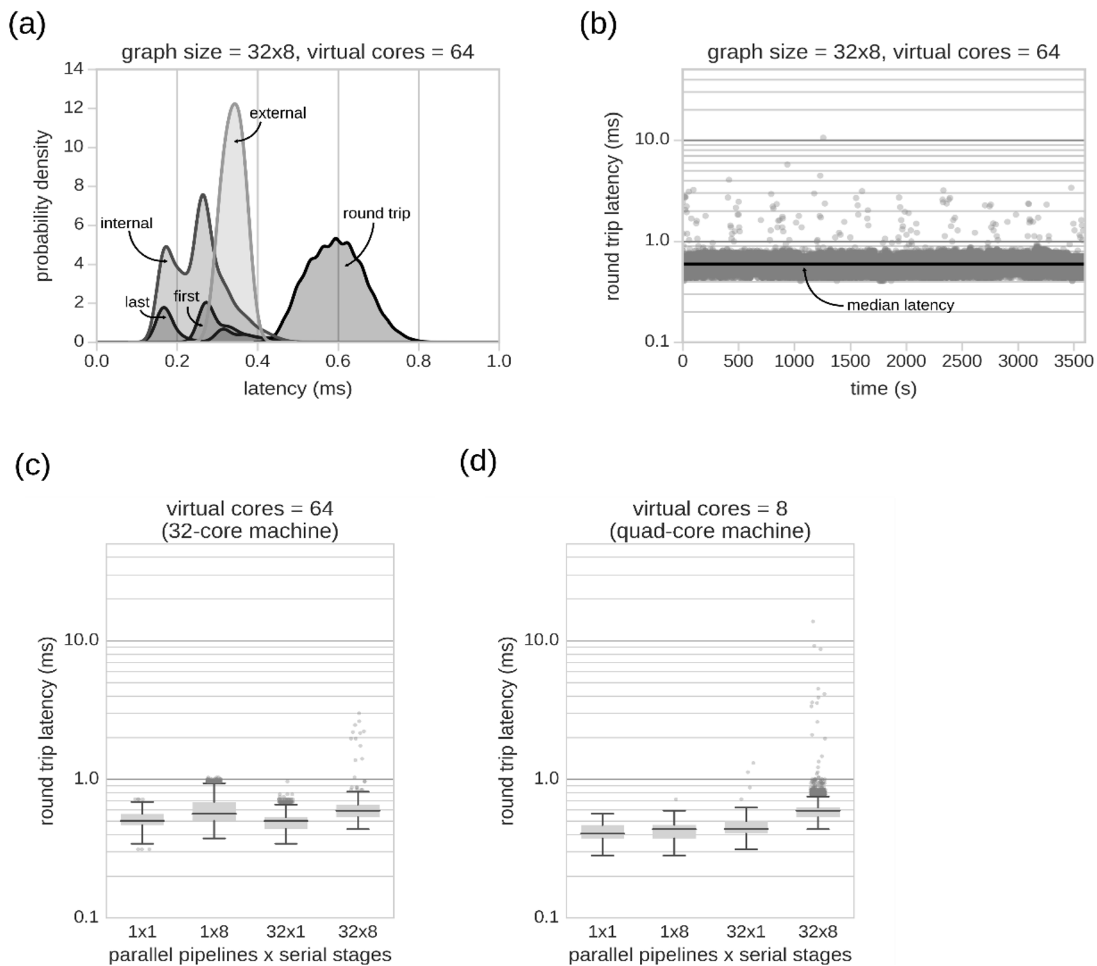

Figure 7. Sub-millisecond round-trip latencies of a Falcon-based closed-loop system. (a) Distribution of round-trip, internal and external latencies obtained with a graph with 32 parallel pipelines and eight serial stages running on a 32-core machine with hardware-based simultaneous multithreading (64 virtual cores). Internal and external latencies represent, respectively, the specific software- and hardware-related time lags that contribute to the round-trip latency (see table 1 for median latencies and 99% intervals). Contributions of detections triggered by samples that arrived first and last in the 6-sample buffer used in the reader node are also shown as partial densities of the distribution of internal latencies. Median values (with 99% intervals) were 0.19 ms (0.14–0.48) for last-in samples and 0.28 ms (0.24–0.49) for first-in samples. (b) The lack of drift of the median of the distribution (black line) of the latency values (scattered gray dots) indicates the temporal stability of the round-trip latency. Test lasted one hour and a 32×8 graph was used with 64 virtual cores. (c) and (d) sub-millisecond round-trip latencies can be achieved also on a quad-core machine. In log-scale, the graphs show the box plots of the round-trip latencies obtained using 64 or eight virtual cores (corresponding to 32 and 4 physical cores) for four different combinations of parallel pipelines and serial stages of the Falcon graph. Outliers were determined as values lying 1.5 times the inter-quartile range below the first or above third quartile.

real-time performance, we used the round-trip latency as a measure of the minimal added computational latency. The round-trip latency was defined as the time difference between the occurrence of a trivially detectable event (with minimal algorithmic latency) and the time of closed-loop feedback (a digital pulse). We determined the round-trip latency of our hardware-software system using a basic edge-detection graph (with minimal algorithmic components) in which the latency is dominated by data transmission within Falcon and communication with the hardware. Latency tests were executed using both Neuralynx and Open Ephys hardware for reading multichannel neural signals (see Methods).

Tests on Neuralynx hardware

In the highest complexity graph tested, Falcon processed 128 channels at 32 kHz using 32 parallel pipelines and 8 serial stages. With all 64 CPU cores enabled (32 physical cores with two hardware threads per core), round-trip latencies were well below 1 ms (figure 7(a)), with a median of 0.59 ms (99% interval: (0.44–0.78)). The worst-case latency over the 7 min test period was 3.0 ms and less than one in a thousand detections occurred with more than 1 ms lag (table 1). Round-trip latency remained stable during the time course of one hour (figure 7(b)). Considering a deadline of 1 ms as a constraint for a general-purpose tool for closed-loop neuroscience, these results demonstrate the (soft) real-time capabilities of a Falcon-based closed-loop system.

We next explored the specific contribution of Falcon to the overall closed-loop response latency. Round-trip latency has both external and internal contributions (figure 5(b)): external contributions relate to the hardware and include network transfer of the digitized input signals and communication delays to the output module, whereas internal contributions

Table 1. Summary of latencies for different graph sizes obtained on two workstations, a 32-machine with either 64 or eight virtual cores enabled and a quad-core machine with all eight virtual cores enabled, using either a Neuralynx or Open Ephys data acquisition system. Latencies obtained with Open Ephys are about one order of magnitude higher than with Neuralynx mostly because of the ms-order USB buffering.

	Round-trip latency			Internal (software) latency		External (hardware) latency	
	Median (ms) (99% interval)	Worst-case (ms)	‰ larger than 1 ms	Median (ms) (99% interval)	Worst-case (ms)	Median (ms) (99% interval)	Worst-case (ms)
Graph size	Neuralynx + 64 enabled virtual cores (32-core machine)						
1 × 1	0.50 (0.37–0.66)	0.72	0.0	0.19 (0.07–0.34)	0.42	0.33 (0.28–0.39)	0.43
1 × 8	0.56 (0.41–0.94)	1.03	0.7	0.25 (0.09–0.63)	0.70	0.33 (0.28–0.39)	0.45
32 × 1	0.50 (0.34–0.66)	0.97	0.0	0.17 (0.06–0.34)	0.64	0.32 (0.27–0.39)	0.41
32 × 8	0.59 (0.44–0.78)	3.00	0.6	0.26 (0.15–0.45)	2.64	0.33 (0.28–0.39)	0.41
	Neuralynx + 8 enabled virtual cores (quad-core machine)						
1 × 1	0.41 (0.28–0.53)	0.56	0.0	0.11 (0.01–0.23)	0.26	0.31 (0.27–0.38)	0.40
1 × 8	0.44 (0.31–0.56)	0.72	0.0	0.13 (0.03–0.24)	0.36	0.32 (0.27–0.38)	0.39
32 × 1	0.44 (0.31–0.56)	1.31	0.1	0.13 (0.03–0.28)	1.03	0.31 (0.26–0.38)	0.39
32 × 8	0.59 (0.44–0.84)	13.81	0.9	0.26 (0.15–0.50)	13.47	0.32 (0.27–0.39)	0.44
	Neuralynx + 8 virtual cores enabled (32-core machine)						
1 × 1	0.50 (0.37–0.63)	2.47	0.0	0.17 (0.05–0.31)	2.13	0.33 (0.28–0.39)	0.41
1 × 8	0.50 (0.37–0.63)	0.75	0.0	0.18 (0.07–0.32)	0.41	0.33 (0.28–0.39)	0.41
32 × 1	0.50 (0.37–0.66)	12.09	0.6	0.17 (0.08–0.34)	11.66	0.32 (0.27–0.38)	0.40
32 × 8	0.66 (0.50–1.03)	29.41	6.6	0.30 (0.19–0.70)	29.05	0.32 (0.27–0.38)	0.39
	Open Ephys + 64 enabled virtual cores (32-core machine)						
1 × 1	9.22 (4.06–14.53)	14.84	n/a	n/a	n/a	n/a	n/a

only include software-related delays like the data streaming between nodes and synchronization lags in aggregator nodes.

External data transfer contributed to the overall round-trip latency slightly more than internal data transfer, with a median value of 0.33 ms (99% interval: (0.28–0.39)) for the external latency and a median value of 0.26 ms (99% interval: (0.15–0.45)) for the internal latency (table 1, figure 7(a)). The distribution of the internal latencies computed with only 'last-in' samples, which were not affected by buffering-induced latency on the reader node (see Methods), had a lower median value of 0.19 ms (99% interval: (0.14–0.48)). These results demonstrate the limited computational overhead added by Falcon on our CPU hardware and highlight that Falcon can be versatilely deployed in closed-loop experiments with stringent latency requirements.

We next asked how the size of the graph impacted the round-trip latency. In general, smaller graph sizes resulted in lower median round-trip latencies and fewer measurements that exceeded the 1-ms deadline (figure 7(c), table 1). By reducing the number of parallel pipelines from 32 to 1, the reduction in median latency was limited to ~5%, from 0.59 ms (99% interval: (0.44–0.78)) to 0.56 ms (99% interval: (0.41–0.94))). The graphs with a single serial stage had the lowest median latency (0.50 ms) and sub-millisecond worst-case latency. Overall, these results show that the parallelization control offered by Falcon was effective in reducing the overall response latency.

We next asked whether CPU hardware was critical for ensuring low round-trip and software latencies in simple and larger graphs. For this, we measured round-trip latency on a machine with a lower number of available CPU cores either the 32-core machine with only four cores enabled or a separate quad-core machine (in both cases a total of 8 virtual cores was available). Again, for all graph sizes the median round-trip latency was well below 1 ms (table 1). For the larger graph sizes (32 × 1 and 32 × 8), the worst-case latency and the fraction of missed deadlines were increased when compared to the test with a fully enabled 32-core machine. When comparing the quad-core machine to the 32-core machine, we found a reduced median round-trip latency for the smaller graph sizes. This reduction is most likely explained by the higher CPU clock rate of the quad-core machine (3.60 GHz versus 2.30 GHz).

To test whether memory availability (rather than CPU) could be a bottle-neck during latency tests, we checked the system memory usage. Falcon's memory usage was respectively 120 MB, 27.5 MB and 12.8 MB during the executing of a 32 × 8, 32 × 1 and 1 × 8 graph. These values are at least one order of magnitude below the typical amount of memory available on modern PCs. Although memory is not a concern for running Falcon, the user must ensure that the memory required by custom-made processors can be accommodated by the available system memory.

Overall, these results highlight that the number of available physical cores and system memory is not critical for ensuring

low median round-trip latencies. Real-time behavior is, however, slightly affected (more missed deadlines) when the graph size greatly exceeds the number of available cores and graphs that incorporate heavy algorithms of neural processing are going to be highly dependent on the number of physical cores (see [21]). Nevertheless, Falcon's well-controlled parallelization can help reach the limits of the available CPU hardware to achieve desired real-time capabilities.

Tests on Open Ephys hardware

To demonstrate the flexibility of Falcon in adapting to different data acquisition systems, we implemented a node for reading multi-channel neural data recorded with Open Ephys hardware. For this system, the round-trip latency cannot meet the 1 ms deadline as it is limited by the use of USB for data transfer and the internal 10 ms buffer size needed to prevent data loss [30]. For the smallest 1×1 graph (processing 4 channels), we measured a median round-trip latency of 9.22 ms (99% interval: (4.06–14.53)) and a worst-case latency of 14.84 ms.

Closed-loop neuroscience applications

With the available detector nodes (figure 2), Falcon can be used to trigger an output upon the occurrence of a single spike (recorded on a single- or multi-electrode sensor), a transient oscillation in the extracellular field potential (e.g. hippocampal ripples) and a population burst.

These detections can be accomplished with acceptably low latencies and sufficient accuracy for closed-loop manipulations. Using a SpikeDector node, single spikes on a tetrode can be detected within 1 ms. With a RippleDetector node we have been able to detect 150–250 Hz hippocampal ripple oscillations with 30 ms average latency from the event start (corresponding to less than 50% average latency relative to total duration of the event) and 80% sensitivity. We have successfully coupled the detection with electrical feedback stimulation to disrupt ripple events and test their role in spatial memory processing, replicating the experimental protocols of previous closed-loop studies [5–7]. The RippleDetector node implementation is generic and when configured with an appropriate filter, it could also be used to detect other transient oscillatory patterns relevant in cognition, like spindles (8–16 Hz), high-gamma bouts (50–125 Hz) and fast ripples (250–500 Hz) [36, 37].

Since hippocampal ripple events are associated with increased population activity, an alternative detection method could rely on detection of bursts in multi-unit activity. To demonstrate this approach, a graph for detection of bursts from a population of neurons was constructed (figure 6). During a test on live recordings from a rat implanted with an array of tetrodes located in the CA1 hippocampal area (see experimental methods), Falcon was able to detect population bursts with low latency (figure 8(a)). The median burst detection latency relative to the start of offline identified burst events was 40.51 ms (figure 8(b)), while the median relative detection latency compared to offline identified burst duration was 42.63% (figure 8(c)).

To better understand the origin of the response latency, we also measured the contribution of the added latency due to the software latency, the external latency due to outgoing transmission of the digital pulse and the computational time involved in executing the algorithms. These added latencies were computed as the difference between the timestamp of the last sample needed to detect threshold crossing and the timestamp of the digital event received in Cheetah. Added latencies had a median value of 0.37 ms (99% interval: (0.31–7.73); see figure 8(d)), demonstrating that the largest contributing factor to the burst detection latency was the algorithmic (e.g. time to reach threshold) and not the computational latency.

The online burst detection algorithm was adapted for real-time use and thus differed from the offline algorithm. Nevertheless, there was a large agreement between the online and offline detected bursts. A total of 1954 events were detected both online and offline with 70.62% (1954/2767) of offline bursts that were detected online and 72.42% (1954/2698) of online bursts that were also detected offline. Most of the disagreement stems from differences in the time-varying thresholds computed by the online and offline algorithms, which results in mismatched detection of mainly low amplitude bursts. A low fraction of bursts was detected twice online (6% of all online detections, corresponding to 0.05 detections s^{-1}). These results illustrate how Falcon can be used to implement common closed-loop experiments.

Online encoding–decoding framework

Falcon's features make it well-suited for real-time decoding of behavioral information (e.g. position of a freely moving animal) at tens of milliseconds time scales from population activity recorded on multi-electrode arrays [38, 39], for example with the intent to manipulate specific hippocampal reactivation patterns [5–7]. In the future Falcon will be extended with nodes that implement decoding algorithms to support this application (figure 9). In the online decoding scenario, spike detection and likelihood computations are performed independently for each electrode (or tetrode) and parallelization can be used to reduce overall latency. For instance, a recently developed decoding approach for unsorted spikes [40, 41], takes less than 1 ms decoding time per spike and could leverage Falcon's parallelization ability for achieving millisecond-scale latency when applied to the activity of a large neuronal population recorded on tens of electrodes. In this application, the flexibility of Falcon can be further exploited to online update the encoding models based on new incoming spike and behavioral data for a real-time adaptive encoding–decoding solution (encoder node in figure 9) [40].

Discussion

Falcon is an open-source development platform for soft real-time analysis of streaming neural and behavioral data during closed-loop neuroscience experiments. The creation of Falcon was driven by the need of a software tool that easily and flexibly implements different closed-loop experiments and provides direct control over the hardware resources of a multi-core computer.

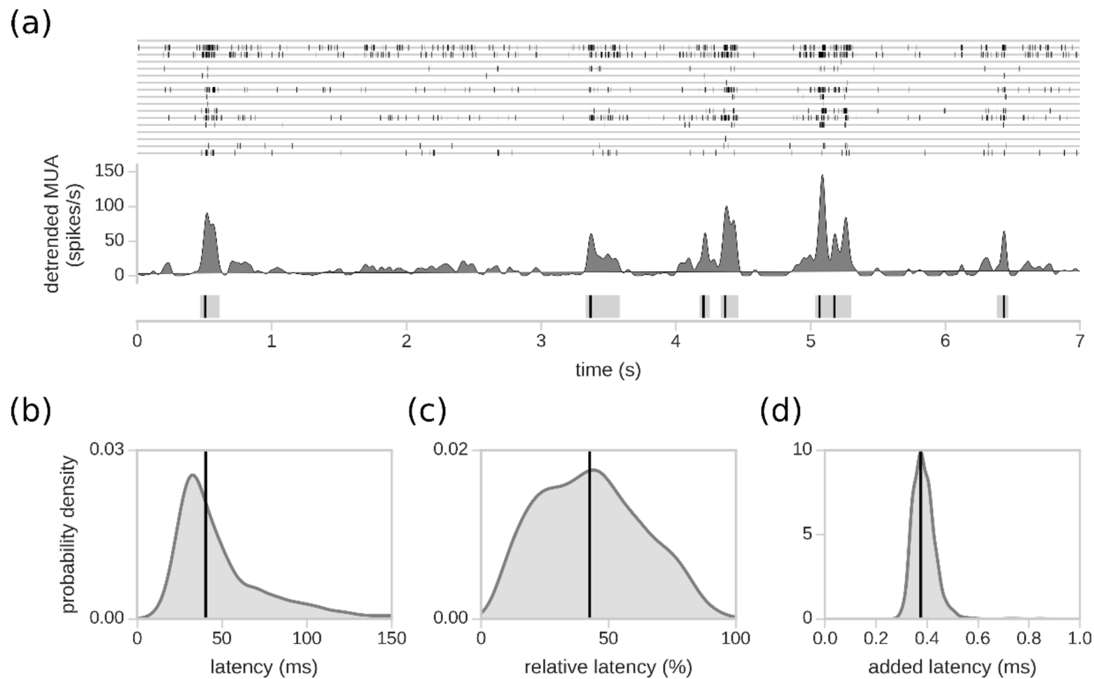

Figure 8. Online detection of hippocampal population bursts in a freely moving rat with Falcon. (a) Top: Raster plot of 7 s of spiking activity recorded by the multi-tetrode array implanted in CA1 during a rest session; each row represents all spikes on a tetrode. Note the transient increases in firing across tetrodes. Bottom: Offline MUA activity (dark gray) of the CA1 population (computed using all recorded spikes and detrended with a double exponential average filter) is displayed together with the online detections (black vertical lines) generated by Falcon. Light-gray boxes indicate the offline-defined burst onset and offset times. Note the early detection of most of the bursts. (b) Distribution of the online detection latencies from the time of burst onset. The black vertical line at 40.51 ms represents the median of the distribution. (c) Distribution of the latencies of the online detections relative to the total duration of each burst. The black line at 42.63% represents the median of the distribution. (d) Distribution of the added latencies of the online detections, following the last sample needed to detect the burst. The black vertical line at 0.37 ms represents the median of the distribution.

At its core, Falcon relies on a graph subsystem that manages the execution of a fully configurable and highly customizable processing graph. Communication between graph nodes is supported through shared state values and streaming data ports that handle arbitrary data types. Each processing node (source, sink, filter, detector) is mapped to a single thread of execution. The general problem of optimally mapping computations to available resources on multi-core CPUs and scheduling when computations are best executed remains a hard challenge. In Falcon, online feedback on fill levels of ring buffers and timing of computations can assist the user in manually tuning the graph topology and parameters to optimize performance.

In addition to its flexibility, a major strength of Falcon is the very low overhead in the execution of a processing graph. For a variety of graph sizes tested on both 32- and 4-core CPU machines, the latency added by the software was less than 0.5 ms for the large majority of measurements with only occasional missing of deadlines (as expected for any soft real-time system). In fact, Falcon rarely missed deadlines (less than 1 in 1000 measurements) when the number of nodes in the graph did not greatly exceed the number of available cores. As such, Falcon is a capable software platform for running sub-millisecond latency closed-loop neuroscience experiments.

Besides the software framework overhead, the overall round-trip latency of a closed-loop system based on Falcon also includes contributions from external data transmission (both on the input and output side), data buffering, and the computations performed in each processing node. By adjusting the level of parallelization and concurrency through the number of parallel pipelines and number of serial stages of the graph, the experimenter can achieve lower real-time response latencies for a given algorithm of interest. Reductions in response latency are primarily obtained by dividing the computational load over a number of parallel processing lines ('independent substream parallelization', see [21]). Limitations due to input/output (transmission latencies) and processing hardware capabilities (number of CPU cores) must, however, be considered and the user must also be careful not to over-parallelize the processing graph. In fact, the CPUs cannot run in parallel more threads than the number of virtual cores and an excessive level of parallelization can even compromise performance over a purely serial implementation.

Falcon round-trip latencies are comparable to if not lower than other closed-loop systems. The sub-millisecond round-trip latency that we measured on Neuralynx hardware is lower than the ~4 ms lower bound of NeuroRighter [29]. Unfortunately, detailed round-trip and software latencies for other open-source BCI tools in similar test conditions (i.e. processing of >100 channels at high sampling rate) are not available. When comparing our results to Open Ephys software, Falcon seems to perform slightly better, with a ~2 ms lower median latency and ~5 ms lower maximum latency [42].

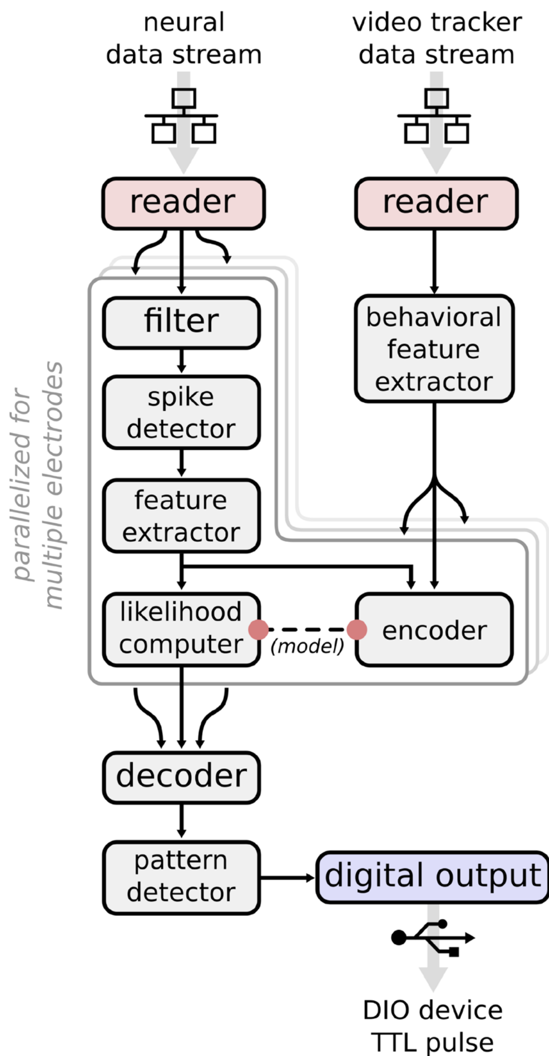

Figure 9. Processing graph design for a hypothetical online encoding–decoding scenario with the goal to detect target spike patterns. The nodes inside the enclosure are duplicated for parallel processing of multiple electrodes.

By design, Falcon is decoupled from specific hardware for data acquisition and closed-loop feedback. We have successfully interfaced Falcon with Digilynx (Neuralynx) and Open Ephys data acquisition systems. For Digilynx, the sample-by-sample signal transmission over Gigabit Ethernet resulted in a sub-millisecond round-trip latency. For Open Ephys, round-trip latencies varied from 4 to 15 ms due to data transmission over USB and internal buffering. Support for other signal sources may be added by implementation of a new reader node, provided the availability of Linux drivers and libraries that interface to the hardware and/or documentation that describes how to parse the incoming data stream. This includes generic input modules for analog and digital signals, and digital video/audio streams. For some commercial neural data acquisition systems it may be difficult to obtain access to the raw data stream and support from the manufacturer would be required. In the future, Falcon could also be integrated with preprocessing hardware like an FPGA or a DSP. For instance, instead of reading the raw data stream, Falcon may directly receive detected spike waveforms and provide closed-loop feedback after processing the spike data.

On the output side, we interfaced Falcon with a USB digital output module and an Arduino microcontroller. Digital output pulses can be used as triggers for external hardware, such as stimulus generators. Closed-loop feedback through microcontrollers can be used for controlling actuators and other devices, for example servo motors that move doors or robotic arms. In addition, through digital triggers and network communication a Falcon-based system can interact with other neuroscientific software applications that provide specialized hardware interfaces, for example to present visual stimuli on a screen, produce audio output or dispense food rewards (e.g. Bonsai [43] and Psychopy [44]).

Because Falcon was primarily designed for flexible real-time processing of streaming experimental data and not as a general data acquisition and visualization software, it lacks an integrated graphical user interface. To perform simultaneous real-time processing of experimental data streams in Falcon and data logging and/or visualization in a graphical user interface, there are two options. As a first option, Falcon's client-server architecture can be harnessed to achieve customized control and implement visualization applications for dedicated processing graphs. Advantageously, such client applications can be written in a programming language most convenient for the user. We have written a simple reference client in Python for control of Falcon and processor-specific visualization clients for online adjustment of critical parameters (e.g. threshold for burst detection). Alternatively, Falcon could tap into a duplicate data stream generated by the acquisition hardware (as is the case for Neuralynx' Digilynx) or by an acquisition server (see [25]). In the latter case, the server is the single point of interaction with the acquisition hardware and serves data to multiple clients, including Falcon and a visualization software.

To solve common closed-loop neuroscience tasks, at present Falcon includes three types of detectors: a spike detector, a ripple-like oscillation detector and a burst detector. In particular, we demonstrated how Falcon can be deployed for online detection of bursts in neural population activity recorded *in vivo*. We detected bursts with approximately 40 ms latency relative to offline determined onsets, with the largest contribution to the latency stemming from the algorithm (time to reach threshold) and not from computational and data transfer time inside Falcon. However, the potential of Falcon goes far beyond the simple experimental case of population burst detection. We described a real-time encoding–decoding scenario that leverages Falcon's power and flexibility. Future work will focus implementing the necessary processing nodes to make this scenario reality. Given its versatility and low round-trip latency against high-sampling high-count input streams, Falcon can also be deployed for controlling EEG-based BCIs and invasive brain–machine interfaces (BMIs) [45] with latency demands in the order of tens to hundreds of milliseconds.

Finally, we foresee several additional lines of future development. The first one relates to the use of graphical processing unit (GPU) calls inside processor nodes [40], for further boosting real-time capabilities [38, 39], especially on limited CPU hardware. Secondly, Falcon could achieve better control over latencies, especially over the worst-case latencies, by using a real-time OS rather than a standard OS (this could of interest for applications of cellular electrophysiology, like dynamic clamping). Finally, since all libraries used in Falcon are cross-platform, with some effort Falcon could be ported to Windows and thus operate on PCs in which other Windows-specific software must be executed or installation of a different OS is not possible (e.g. clinical settings).

Conclusions

We presented Falcon, a novel tool for implementing a wide variety of closed-loop neuroscientific experiments. Falcon is a highly versatile open-source software capable of sub-millisecond response latency over high-rate (e.g. 32 kHz) streaming data acquired from high-count multi-electrode arrays (e.g. 128 channels). Falcon offers a unique combination of features that distinguish it from existing open-source software. In fact, it can not only be used to build arbitrary processing graphs with existing nodes, but it can also use new highly customizable data types and nodes; moreover, by mapping each node directly onto one thread, it empowers the user with direct control over CPU resources. Falcon can be exploited for implementing closed-loop experiments requiring computationally intensive neural algorithms and complex data structures, like population encoding and decoding of firing patterns from neuronal ensembles.

Acknowledgments

We are grateful to Cristian Inel for his important contribution in the development of a predecessor software framework, Frédéric Michon for his help with online burst detection *in vivo*, Joana Serpa Santos for providing the datasets used for testing online ripple detection and Nicolas Sist for his feedback on the installation and use of the software.

Conflict of interest

The authors declare no competing financial interests.

References

[1] Buzsáki G 2015 Hippocampal sharp wave-ripple: a cognitive biomarker for episodic memory and planning *Hippocampus* **25** 1073–188
[2] Potter S M, El Hady A and Fetz E E 2014 Closed-loop neuroscience and neuroengineering *Front. Neural Circuits* **8** 115
[3] Zrenner C, Belardinelli P, Müller-Dahlhaus F and Ziemann U 2016 Closed-loop neuroscience and non-invasive brain stimulation: a tale of two loops *Front. Cell. Neurosci.* **10** 92
[4] El Hady A 2016 *Closed Loop Neuroscience* (London: Academic)
[5] Jadhav S P, Kemere C, German P W and Frank L M 2012 Awake hippocampal sharp-wave ripples support spatial memory *Science* **336** 1454–8
[6] Girardeau G, Benchenane K, Wiener S I, Buzsáki G and Zugaro M B 2009 Selective suppression of hippocampal ripples impairs spatial memory *Nat. Neurosci.* **12** 1222–3
[7] Ego-Stengel V and Wilson M A 2010 Disruption of ripple-associated hippocampal activity during rest impairs spatial learning in the rat *Hippocampus* **20** 1–10
[8] de Lavilléon G, Lacroix M M, Rondi-Reig L and Benchenane K 2015 Explicit memory creation during sleep demonstrates a causal role of place cells in navigation *Nat. Neurosci.* **18** 493–5
[9] Siegle J H and Wilson M A 2014 Enhancement of encoding and retrieval functions through theta phase-specific manipulation of hippocampus *Elife* **3** e03061
[10] O'Connor D H, Hires S A, Guo Z V, Li N, Yu J, Sun Q-Q, Huber D and Svoboda K 2013 Neural coding during active somatosensation revealed using illusory touch *Nat. Neurosci.* **16** 958–65
[11] Wright J, Macefield V G, van Schaik A and Tapson J C 2016 A Review of control strategies in closed-loop neuroprosthetic systems *Front. Neurosci.* **10** 312
[12] Heck C N *et al* 2014 Two-year seizure reduction in adults with medically intractable partial onset epilepsy treated with responsive neurostimulation: final results of the RNS System Pivotal trial *Epilepsia* **55** 432–41
[13] Little S *et al* 2013 Adaptive deep brain stimulation in advanced Parkinson disease *Ann. Neurol.* **74** 449–57
[14] El Hady A, Varona P, Arroyo D, Rodríguez F B and Nowotny T 2016 Online event detection requirements in closed-loop neuroscience *Closed Loop Neuroscience* (London: Academic) ch 6, pp 81–91
[15] Rutishauser U, Kotowicz A and Laurent G 2013 A method for closed-loop presentation of sensory stimuli conditional on the internal brain-state of awake animals *J. Neurosci. Methods* **215** 139–55
[16] van Gerven M *et al* 2009 The brain–computer interface cycle *J. Neural Eng.* **6** 041001
[17] Ben-Ari M 2006 *Principles of Concurrent and Distributed Programming* (Reading, MA: Addison-Wesley)
[18] Kopetz H 2011 *Real-Time Systems* (Boston, MA: Springer)
[19] Nguyen T K T, Navratilova Z, Cabral H, Wang L, Gielen G, Battaglia F P and Bartic C 2014 Closed-loop optical neural stimulation based on a 32-channel low-noise recording system with online spike sorting *J. Neural Eng.* **11** 046005
[20] Liu J W S 2000 *Real-Time Systems* (Englewood Cliffs, NJ: Prentice Hall)
[21] Fischer J, Milekovic T, Schneider G and Mehring C 2014 Low-latency multi-threaded processing of neuronal signals for brain-computer interfaces *Front. Neuroeng.* **7** 1
[22] Delorme A, Mullen T, Kothe C, Akalin Acar Z, Bigdely-Shamlo N, Vankov A and Makeig S 2011 EEGLAB, SIFT, NFT, BCILAB, and ERICA: new tools for advanced EEG processing *Comput. Intell. Neurosci.* **2011** 130714
[23] Kothe C A and Makeig S 2013 BCILAB: a platform for brain–computer interface development *J. Neural Eng.* **10** 056014
[24] Brunner C *et al* 2012 BCI software platforms *Towards Practical Brain-Computer Interfaces* (Berlin: Springer) pp 303–31
[25] Renard Y, Lotte F, Gibert G, Congedo M, Maby E, Delannoy V, Bertrand O and Lécuyer A 2010 OpenViBE: an open-source software platform to design, test, and use brain–computer interfaces in real and virtual environments *Presence Teleoperators Virtual Environ.* **19** 35–53
[26] Degenhart A D, Kelly J W, Ashmore R C, Collinger J L, Tyler-Kabara E C, Weber D J and Wang W 2011 Craniux: A LabVIEW-based modular software framework for

brain-machine interface research *Comput. Intell. Neurosci.* **2011** 363565
[27] Biró I and Giugliano M 2015 A reconfigurable visual-programming library for real-time closed-loop cellular electrophysiology *Front. Neuroinform.* **9** 17
[28] Ortega F A, Butera R J, Christini D J, White J A and Dorval A D 2014 Dynamic clamp in cardiac and neuronal systems using RTXI *Methods Mol. Biol.* **1183** 327–54
[29] Newman J P, Zeller-Townson R, Fong M-F, Arcot Desai S, Gross R E and Potter S M 2013 Closed-loop, multichannel experimentation using the open-source neurorighter electrophysiology platform *Front. Neural Circuits* **6** 98
[30] Siegle J H, Cuevas López A, Patel Y, Abramov K, Ohayon S and Voigts J 2017 Open Ephys: an open-source, plugin-based platform for multichannel electrophysiology *J. Neural Eng.* **14** 045003
[31] Hintjens P 2013 *ZeroMQ: Messaging for Many Applications* (Sebastopol, CA: O'Reilly Media)
[32] Marr D T, Binns F, Hill D L, Hinton G, Koufaty D A, Miller J A and Upton M 2002 Hyper-threading technology architecture and microarchitecture *Intel Technol. J.* **6** 1–12
[33] Fried I, Cerf M and Kreiman G 2014 Data analysis techniques for human microwire recordings: spike detection and sorting, decoding, relation between neurons and local field potentials *Single Neuron Studies of the Human Brain* (Cambridge, MA: MIT Press) pp 59–98
[34] Kloosterman F, Davidson T J, Gomperts S N, Layton S P, Hale G, Nguyen D P and Wilson M A 2009 Micro-drive array for chronic *in vivo* recording: drive fabrication *J. Vis. Exp.* **26** e1094
[35] Talakoub O, Gomez Palacio Schjetnan A, Valiante T A, Popovic M R and Hoffman K L Closed-loop interruption of hippocampal ripples through fornix stimulation in the non-human primate *Brain Stimul.* **9** 911–8
[36] Kucewicz M T *et al* 2014 High frequency oscillations are associated with cognitive processing in human recognition memory *Brain* **137** 2231–44
[37] Averkin R G, Szemenyei V, Bordé S and Tamás G 2016 Identified cellular correlates of neocortical ripple and high-gamma oscillations during spindles of natural sleep *Neuron* **92** 916–28
[38] Davidson T J, Kloosterman F and Wilson M A 2009 Hippocampal replay of extended experience *Neuron* **63** 497–507
[39] Kloosterman F 2012 Analysis of hippocampal memory replay using neural population decoding *Neuronal Network Analysis* ed T Fellin and M Halassa (New York: Human Press) pp 259–82
[40] Kloosterman F, Layton S P, Chen Z and Wilson M A 2014 Bayesian decoding using unsorted spikes in the rat hippocampus *J. Neurophysiol.* **111** 217–27
[41] Sodkomkham D, Ciliberti D, Wilson M A, Fukui K, Moriyama K, Numao M and Kloosterman F 2016 Kernel density compression for real-time Bayesian encoding/decoding of unsorted hippocampal spikes *Knowl.-Based Syst.* **94** 1–12
[42] Voigts J *et al* 2016 A open-source system and interface standards for very high data rate neurophysiology and low latency closed-loop experiments *Program No. 848.13. 2016 Neuroscience Meeting Planner* (San Diego, CA: Society for Neuroscience) (online)
[43] Lopes G *et al* 2015 Bonsai: an event-based framework for processing and controlling data streams *Front. Neuroinform.* **9** 7
[44] Peirce J W 2007 PsychoPy—psychophysics software in python *J. Neurosci. Methods* **162** 8–13
[45] Lebedev M A and Nicolelis M A L 2006 Brain–machine interfaces: past, present and future *Trends Neurosci.* **29** 536–46