FLICK: Developing and Running Application-Specific Network Services

Abdul Alim[†], Richard G. Clegg[†], Luo Mai[†], Lukas Rupprecht[†], Eric Seckler[†],
Paolo Costa[‡†], Peter Pietzuch[†], Alexander L. Wolf[†], Nik Sultana[*],
Jon Crowcroft[*], Anil Madhavapeddy[*], Andrew W. Moore[*], Richard Mortier[*],
Masoud Koleini[♭], Luis Oviedo[♭], Derek McAuley[♭], Matteo Migliavacca[‡]

[†]Imperial College London, [‡]Microsoft Research, [*]University of Cambridge,
[♭]University of Nottingham, [‡]University of Kent

Abstract

Data centre networks are increasingly programmable, with *application-specific* network services proliferating, from custom load-balancers to middleboxes providing caching and aggregation. Developers must currently implement these services using traditional low-level APIs, which neither support natural operations on application data nor provide efficient performance isolation.

We describe FLICK, a framework for the programming and execution of application-specific network services on multi-core CPUs. Developers write network services in the FLICK *language*, which offers high-level processing constructs and application-relevant data types. FLICK programs are translated automatically to efficient, parallel *task graphs*, implemented in C++ on top of a user-space TCP stack. Task graphs have bounded resource usage at runtime, which means that the graphs of multiple services can execute concurrently without interference using cooperative scheduling. We evaluate FLICK with several services (an HTTP load-balancer, a Memcached router and a Hadoop data aggregator), showing that it achieves good performance while reducing development effort.

1 Introduction

Distributed applications in data centres increasingly want to adapt networks to their requirements. *Application-specific network services*, such as application load-balancers [23, 40], request data caches [36], and in-network data aggregators [29], therefore blur the boundary between the network fabric at the core and applications at the edge. For example, a Memcached request router can transparently scale deployments by routing requests using knowledge of the Memcached protocol [36]. *In this paper, we explore how application developers, not*

Authors are ordered alphabetically, grouped by institution and non-faculty/faculty status.

network engineers, can be supported when implementing new application-specific network services.

Existing software middlebox platforms, such as Click-OS [30], xOMB [3] and SmartSwitch [53], support only *application-independent* network services, i.e. IP routers, firewalls or transport-layer gateways. Using them to interact with payload data in network flows leads to an impedance mismatch due to their byte-oriented, per-packet APIs. Instead, application developers would prefer high-level constructs and data types when expressing processing logic. For example, when defining the dispatching logic of a Memcached request router, a developer would ideally treat key/value pairs as a first-class data type in their program.

Today's middlebox platforms also force developers to optimise their code carefully to achieve high throughput—implementing a new Click module [24, 30] in C++ that can process data at 10 Gbps line rate is challenging. As a result, many new application-specific network services [40, 29] are built from scratch rather than leveraging the above platforms.

Considerable work went into developing new high-level languages for network control within software-defined networking (SDN) [16, 33, 8, 48]. While these simplify the specification of network management policies, they typically operate on a per-packet basis and support a limited set of per-packet actions once matched, e.g. forwarding, cloning or dropping. In contrast, application-specific network services must refer to payload data, e.g. messages, key/value pairs or deserialised objects, and carry out richer computations, e.g. arbitrary payload transformations, caching or data aggregation.

Our goal is to enable developers to express application-specific network services in a natural high-level programming model, while executing such programs in an efficient and scalable manner. This is challenging for several reasons: (i) in many cases, the cost of data deserialisation and dynamic memory management reduces

achievable processing throughput. While high-level programming languages such as Java or Python can manipulate complex application objects, they struggle to provide predictable processing throughput for line-rate processing of network data; (ii) a typical data centre may host hundreds of applications, with each potentially requiring its own network service. Services must thus share resources, e.g. CPU and memory, without interference. Existing middlebox platforms use coarse-grained virtualisation [30], which carries a context-switching overhead of hundreds of microseconds. This is too high for fine-grained resource sharing between many application-specific network services; and (iii) most of the applications use TCP for transport, and an application-specific middlebox needs to terminate TCP connections to access data. Performance and scalability of such middleboxes are often bounded by the high cost of connection termination and frequent socket reads/writes.

We describe **FLICK**, a framework for developers to program and execute application-specific network services. It consists of the FLICK *language* for defining network services, and the FLICK *platform* for executing compiled programs efficiently on multi-core CPUs.

Programs in the FLICK language have *bounded resource usage* and are *guaranteed to terminate*. This is possible because most application-specific network services follow a similar pattern: they deserialise and access application data types, iterate over these data types to perform computation, and output the results as network flows. The language is therefore statically typed, and all built-in types (e.g. integer, string, and array) must have a maximum size to avoid dynamic memory allocation. Programs can refer to complex application-defined data types, such as messages or key/value pairs, for which efficient parsers are synthesised from the type definitions in the program. Since functions can only perform finite iteration over fixed-length data types, FLICK programs with finite input must terminate.

A compiler translates FLICK programs into *task graphs* implemented in C++. Tasks graphs are designed to permit the efficient and safe execution of many concurrent network services on a shared platform. A task graph consists of parallel *tasks* that define the computation of the FLICK program, and *channels* that propagate data between concurrently executing tasks. *Input/output tasks* perform the serialisation/deserialisation of data to and from application objects. Since FLICK programs explicitly specify accesses to application data fields, the compiler can generate custom parsing code, eliminating the overheads of general-purpose parsers.

The FLICK *platform* executes multiple task graphs belonging to different services. To reduce the overhead of frequent connection termination and socket operation, task graphs use a modified version of a highly-scalable user-space TCP stack (mTCP [21]) with Intel's Data Plane Development Kit (DPDK) [20]. Task graphs are also scheduled *cooperatively*, avoiding context-switching overhead. They cannot interfere with each other, both in terms of performance and resources, due to their safe construction from FLICK programs.

We evaluate a prototype implementation of FLICK using both micro-benchmarks and three application-specific network services: an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator. Our results show that FLICK can execute these services with throughput and latency that matches that of specialised middlebox implementations. In addition, it scales with a larger number of compute tasks. This paper focuses on the design, implementation and performance of a single FLICK middlebox. However, the wider vision is of a number of such boxes within a data centre [10].

2 Application-Specific Network Services

FLICK focuses on a specific context: data centres in which multiple, complex, distributed applications run concurrently. In this case, to achieve higher performance, flexibility or efficiency, it is advantageous to execute portions of these applications, e.g. related to load-balancing, caching or aggregation, as *application-specific network services* directly on network elements.

To do this, application developers must add code to network elements such as *software middleboxes*. Today this typically means that they must implement complicated features of the underlying network protocols (e.g. TCP flow construction, HTTP parsing and application data deserialisation). For performance reasons, network services must be highly parallel, which requires considerable developer expertise to achieve. Network resources are also inherently shared: even if hosts can be assigned to single applications, network elements must host many services for different applications.

The goal of FLICK is to allow developers to easily and efficiently introduce application-specific processing into network elements. Present approaches are unsatisfactory for three key reasons: (i) they provide only low-level APIs that focus on the manipulation of individual packets, or at best, individual flows; (ii) they do not permit developers to implement services in high-level languages, but typically rely on the use of low-level languages such as C; and (iii) they provide little support for the high degrees of concurrency that are required to make network service implementations perform well.

Next we elaborate on some of these challenges as encountered in our example applications (§2.1), and then contrast our approach with existing solutions (§2.2).

2.1 Use cases

We consider three sample uses for application-specific services: HTTP load balancing, Memcached request routing, and Hadoop data aggregation.

HTTP load balancer. To cope with a large number of concurrent requests, server farms employ load balancers as front ends. These are implemented by special-purpose hardware or highly-optimised software stacks and both sacrifice flexibility for performance. As a result, load balancers must often be reimplemented for each application to tailor them to specific needs. For example, this may be necessary to ensure consistency when multiple TCP connections are served by the same server; to improve the efficiency of clusters running Java code, a load balancer may avoid dispatching requests to servers that are currently performing garbage collection [27]; finally, there is increasing interest from Internet companies to monitor application-specific request statistics—a task that load balancers are ideally placed to carry out [13].

Memcached proxy. Memcached [15] is a popular distributed in-memory key/value store for reducing the number of client reads from external data sources by caching read results in memory. In production environments, a proxy such as *twemproxy* [52] or *mcrouter* [36] is situated usually between clients and servers to handle key/value mappings and instance configurations. This decouples clients and servers and allows the servers to scale out or in horizontally.

Past attempts to implement Memcached routers have involved user-space solutions [36], incurring high overheads due to expensive memory copies between kernel- and user-space. More recent proposals, such as *MemSwitch* [53], have shown that a dedicated single-purpose software switch that intercepts and processes Memcached traffic can be more efficient. To customise MemSwitch, developers, however, must write complex in-network programs that process raw packet payloads. This not only compromises the safety and performance of the network stack, but also complicates development—it requires knowledge about low-level details of networking as well as skills for writing high-performance, parallelisable packet-processing code.

Hadoop data aggregator. Hadoop [54] is a popular map/reduce framework for data analysis. In many deployments, job completion times are network-bound due to the shuffle phase [9]. This means that performance can be improved through an application-specific network service for *in-network data aggregation* [29], which executes an intermediate in-network reduction within the network topology before data reaches the reducers, thus reducing traffic crossing the network.

Providing an in-network data aggregation for Hadoop serves as a good example of an application-specific ser-

vice that must carry out complex data serialisation and deserialisation. A developer wishing to implement in-network reduce logic must therefore re-implement the logic necessary to reconstruct Hadoop key/value pairs from TCP flows—a difficult and error-prone task.

2.2 Existing solution space

There are several proposals for addressing the challenges identified in the use cases above. We observe that existing solutions typically fit into one of four classes:

(i) Specialised, hand-crafted implementations. Systems such as *netmap* [43, 44] provide for efficient user-space implementations of packet-processing applications. Unfortunately, they offer only low-level abstractions, forcing developers to process individual packets rather than high-level business logic.

(ii) Packet-oriented middleboxes. Frameworks for implementing software middleboxes, such as *ClickOS* [30] and *SmartSwitch* [53], enable high-performance processing of network data and can be used to build higher-level abstractions. However, they fail to support useful high-level language features such as strong and static typing, or simple support for data-parallel processing.

(iii) Network programmability. More recently, we see increasing deployment of *software-defined networking* techniques, usually *OpenFlow* [31]. More advanced technologies have been proposed such as *P4* [8] and *Protocol Oblivious Forwarding* [47]. These enable efficient in-network processing of traffic, selectively forwarding, rewriting and processing packets. However, they suffer from many of the same issues as (ii) due to their narrow focus on packet-level abstractions.

(iv) Flow-oriented servers. For in-network processing concerned with higher-level flow abstractions, it is common to leverage existing server implementations, such as *Nginx* [35] or *Apache* [51], and customise them either at the source level or through extensibility mechanisms such as modules. Another example is Netflix ribbon [34], which provides a number of highly configurable middlebox services along with a Java library to build custom services. While this raises the level of abstraction somewhat, the overheads of using such large, complex pieces of software to perform application-specific network services are substantial.

3 FLICK Framework

We motivate our design by outlining requirements (§3.1), and providing a high-level overview (§3.2).

3.1 Requirements

Based on the shortcomings of the approaches highlighted in §2.2, we identify the following three design requirements for our framework:

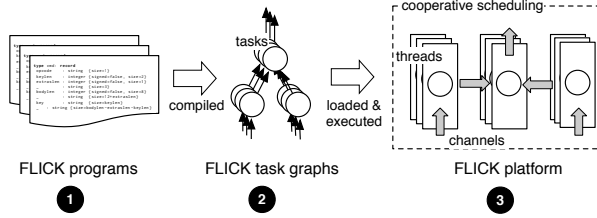**R1: Application-level abstractions:** developers should

Figure 1: Overview of the FLICK framework

be able to express their network services using familiar constructs and abstractions without worrying about the low-level details of per-packet (or per-flow) processing;

R2: High parallelism: to achieve line-rate performance, programs for application-specific network services must exploit both data and task parallelism without requiring significant effort from the developers;

R3: Safe and efficient resource sharing: middleboxes are shared by multiple applications/users, therefore, we need to ensure that programs do not interfere with one another, both in terms of CPU and memory resources.

To meet these requirements, FLICK follows the scheme shown in Figure 1. For the desired level of abstraction (**R1**), it provides a novel high-level language (❶; §4). The language allows developers to focus on the business logic of their network services ignoring low-level details (e.g. serialisation or TCP reassembly).

Compared to general-purpose languages such as C or Java, the FLICK language offers a constrained programming environment. This makes it easier to compile FLICK programs to parallel FLICK *task graphs* (❷; §5). The division of programs into tasks allows the platform to take advantage of both data and task parallelism, thus exploiting multi-core CPUs (**R2**).

Finally, the FLICK language bounds the resource usage for each invocation of a network service. This allows task graphs to be executed by the FLICK *platform* according to a cooperative scheduling discipline (❸; §5), permitting a large number of concurrent task graphs to share the same hardware resources with little interference (**R3**). A pool of worker threads execute *tasks* cooperatively, while *channels* move data between tasks.

3.2 Overview

We now give a more detailed overview of how a developer uses the FLICK framework (see Figure 1). First they write the logic of their application-specific network services in the FLICK language. After compilation by the FLICK compiler, the FLICK platform runs a program as an *instance*, consisting of a set of *task graphs*. Each task graph comprises of a directed acyclic graph of *tasks* connected by *task channels*. Depending on the program semantics, multiple instances of the task graph can be instantiated for each network request, or a single graph can be used by multiple requests.

A task is a schedulable unit of computation. Each task processes a stream of input values and generates a stream of output values. Initial input to the task graph is handled by one or more *input tasks*, which consume data from a single *input channel*, i.e. the byte stream of a TCP connection. An input task then deserialises bytes to values using deserialisation/parsing code generated by the FLICK compiler from the types specified in the FLICK program. Deserialisation splits data into the smallest units appropriate for the task being considered. For example, if the input is from a web client, the byte stream would be deserialised into individual complete HTTP requests; for Hadoop, a key/value pair is more appropriate.

Received data is then processed by one or more compute tasks and, finally, output from the task graph is emitted to the outside world via an *output task*, representing a single outgoing TCP connection. The output task also executes efficient serialisation code generated from the FLICK program, converting values into a byte stream that is placed onto an *output channel* for transmission.

4 FLICK Programming Model

FLICK provides a domain-specific language (DSL) targeting application-specific middlebox programming. While a difficult task, we decided to design a new language because we found existing general-purpose languages inappropriate for middlebox programming due to their excessive expressive power. Even safe redesigns of widely-used languages, such as *Cyclone* [22], are too powerful for our needs because, by design, they do not restrict the semantics of programs to terminate and bound the used resources. Existing specialised languages for network services, such as *PLAN* [18], are typically packet-centric. This makes it hard to implement application-specific traffic logic that is flow-centric. A new domain-specific language presents us with the opportunity to incorporate primitive abstractions that better fit the middlebox domain.

We also considered *restricting* an existing language to suit our needs (for example OCaml restricted so it performs no unbounded loops and no garbage collection). This, however, presented two difficulties: (i) exposing programmers to a familiar language but with altered semantics would be confusing; and (ii) it would prevent us from including language features for improved safety, such as static type-checking.

Numerous systems programming tasks have been simplified by providing DSLs to replace general-purpose programming languages [26, 28, 6, 39, 11]. The FLICK language is designed (i) to provide convenient, familiar high-level language abstractions targeted specifically at middlebox development, e.g. application-level types, processes and channels alongside traditional functions and primitive types; (ii) to take advantage of execution

4

Listing 1: FLICK program for Memcached proxy

```
1  type cmd: record
2    key   : string
3
4  proc Memcached: (cmd/cmd client, [cmd/cmd] backends)
5    | backends => client
6    | client => target_backend(backends)
7
8  fun target_backend: ([-/cmd] backends, req:cmd) -> ()
9
9    let target = hash(req.key) mod len(backends)
10   req => backends[target]
```

parallelism for high throughput; and (iii) to enable efficient and safe handling of multiple programs and many requests on shared hardware resources by making it impossible to express programs with undesirable behaviour, such as unbounded resource consumption.

In the FLICK language, developers describe application-specific network services as a collection of interconnected *processes*. Each process manipulates values of the application's data types, in contrast to earlier work which described network services as simple packet processors [24, 7, 30]. Application data is carried over *channels*, which interconnect processes with one another and with network flows. Processes interact with channels by consuming and processing input data read from them, and by transmitting output over them. Processes, channels and network interactions are handled by the FLICK *platform*.

The FLICK language is designed to achieve efficient parallel execution on multi-core CPUs using high-level parallel primitives. By default, the language offers parallelism across multiple requests, handling them concurrently. It supports the safe sharing of resources by *bounding* the resource use of an individual program. Processing of continuous network flows belonging to an application is subdivided into discrete units of work so that each process consumes only a bounded amount of resource. To achieve this, FLICK control structures are restricted to finite iteration only. This is not a significant limitation, however, as application-specific network services typically carry out deterministic transformations of network requests to generate responses. User-defined functions are written in FLICK itself, rather than in a general purpose language as in Click [24] or Pig [37]), which preserves the safety of network services expressed in FLICK.

After presenting the FLICK language by example (§4.1), we describe its application data types (§4.2), primitives and compilation (§4.3).

4.1 Overview

Listing 1 shows a sample FLICK program that implements a Memcached proxy. Programs are composed of three types of declarations: *data types* (lines 1–2), *processes* (lines 4–6) and *functions* (lines 8–10).

Processes have signatures that specify how they con-

nect to the outside world. In this case, a process called Memcached declares a signature containing two channels (line 4): the client channel produces and accepts values of type cmd, while backends is an array of channels, each of which produces and accepts values of type cmd.

Processes are instantiated by the FLICK platform, which binds channels to underlying network flows (§5). In this example, when a client sends a request, the FLICK platform creates a new Memcached task graph and assigns the client connection to this graph. Giving each client connection a new task graph ensures that responses are routed back to the correct client.

A process body describes how data is transformed and routed between channels connected to a process. The language design ensures that only a finite amount of input from each channel is consumed. The body of the Memcached process describes the application-specific network service: data received from *any* channel in backends is sent to the client channel (line 5); data received from the client is processed by the target_backend function (line 6), which in turn writes to a suitable channel in the backends array (line 10).

4.2 Supporting application data types

FLICK programs operate on application data types representing the exchanged messages. After an input task reads such messages from the network, they are parsed into FLICK data types. Similarly, before processed data values are transmitted by an output task, they are serialised into the appropriate wire format representation.

The transformation of messages between wire format and FLICK data types is defined as a *message grammar*. During compilation, FLICK generates the corresponding parsing and serialisation code from the grammar, which is then used in the input and output tasks of the task graph, respectively. The generated code is optimised for efficiency in three ways: (i) it does not dynamically allocate memory; (ii) it supports the incremental parsing of messages as new data arrives; and (iii) it is adapted automatically to specific use cases.

The syntax to define message grammars is based on that of the *Spicy* (formerly *Binpac++* [46]) parser generator. The language provides constructs to define messages and their serialised representation through units, fields, and variables, and their composition: *units* are used to modularise grammars; *fields* describe the structure of a unit; and *variables* can compute the value of expressions during parsing or serialisation, e.g. to determine the size of a field. FLICK grammars can express any LL(1)-parsable grammar as well as grammars with dependent fields, in a manner similar to Spicy. The FLICK framework provides reusable grammars for common protocols, such as the HTTP [14] and Memcached protocols [50]. Developers can also specify additional mes-

Listing 2: Partial grammar for Memcached protocol

```
1  type cmd = unit {
2      %byteorder = big;
3
4      magic_code        : uint8;
5      opcode            : uint8;
6      key_len           : uint16;
7      extras_len        : uint8;
8                        : uint8;  # anonymous field,
                                    reserved for future use
9      status_or_v_bucket : uint16;
10     total_len         : uint32;
11     opaque            : uint32;
12     cas               : uint64;
13
14     var value_len     : uint32
15         &parse = self.total_len -
16             (self.extras_len + self.key_len)
17         &serialize = self.total_len =
18             self.key_len + self.extras_len + $$;
19     extras   : bytes &length = self.extras_len;
20     key      : string &length = self.key_len;
21     value    : bytes &length = self.value_len;
22  };
```

Listing 3: FLICK program for Hadoop data aggregator

```
1  type kv: record
2    key : string
3    value : string
4
5  proc hadoop: ([kv/-] mappers, -/kv reducer):
6    if (all_ready(mappers)):
7      let result = foldt on mappers
8        ordering elem e1, e2 by elem.key as e_key:
9      let v = combine(e1.val, e2.val)
10     kv(e_key, v)
11   result => reducer
12
13 fun combine(v1: string, v2: string) -> (string): ...
```

sage grammars for custom formats, such as application-specific Hadoop data types.

Listing 2 shows a simplified grammar for Memcached. The cmd unit for the corresponding FLICK data type is a sequence of fixed-size fields (lines 4–12), a variable (lines 14–18), and variable-size fields (lines 19–21). Each field is declared with its wire-format data type, e.g. the opcode field is an 8-bit integer (line 5). The sizes of the extras, key, and value fields are determined by the parsed value of the extras_len and key_len fields as well as the value_len variable, which is computed during parsing according to the expression in lines 15 and 16. During serialisation, the values of extras_len, key_len, and value_len are updated according to the sizes of the values stored in the extras, key, and value fields. Subsequently, the value of total_len is updated according to the variable's serialisation expression in lines 17 and 18. The %byteorder property declaration in line 2 specifies the wire format encoding of number values—the generated code transforms such values between the specified big-endian encoding and the host byte-order. More advanced features of the grammar language include choices between alternative field sequences, field repetitions (i.e. lists), and transformations into custom FLICK field types (e.g. enumerations).

FLICK grammars aim to be reusable and thus include all fields of a given message format, even though application-specific network services often only require a subset of the information encoded in a message. To avoid generated parsers and serialisers handling unnecessary data, FLICK programs make accesses to message fields explicit by declaring a FLICK data type corresponding to the message (Listing 1, lines 1–2). This enables the FLICK compiler to generate input and output tasks that only parse and serialise the required fields for these data types and their dependencies. Other fields are aggre-

gated into either simplified or composite fields, and then skipped or simply copied in their wire format representation. Developers can thus reuse complete message grammars to generate parsers and serialisers, while benefiting from efficient execution for their application-specific network service.

The current FLICK implementation does not support exceptions, but data type grammars could provide a default behaviour when a message is incomplete or not in an expected form.

4.3 Primitives and compilation

The FLICK language is strongly-typed for safety. To facilitate middlebox programming, it includes channels, processes, explicit parallelism, and exception handling as native features. For example, events such as broken connections can be caught and handled by FLICK functions, which can notify a backend or record to a log. State handling is essential for describing many middleboxes, and the language supports both session-level and long-term state, whose scope extends across sessions. The latter is provided through a key/value abstraction to task graph instances by the FLICK platform. To access it, the programmer declares a dictionary and labels it with a global qualifier. Multiple instances of the service share the key/value store.

The language is restricted to allow only computations that are guaranteed to terminate, thus avoiding expensive isolation mechanisms while supporting multiple processes competing for shared resources. This restriction allows static allocation of memory and cooperative task scheduling (see §5).

The FLICK language offers primitives to support common datatypes such as bytes, lists and records. Iteration may only be carried out on finite structures (e.g. lists). It also provides primitives such as fold, map and filter but it does not offer higher-order functions: functions such as fold are translated into finite for-loops. Datatypes may be annotated with cardinalities to determine statically the required memory. Loops and branching are compiled to their native counterparts in C++. Channel- and process-related code is translated to API calls exposed by the platform (see §5). The language re-

lies on the C++ compiler to optimise the target code.

Channels are typed, and at compile time the platform determines that FLICK programs only send valid data into channels. Due to the language's static memory restrictions, additional channels cannot be declared at runtime, though channels may be rebound, e.g. to connect to a different backend server.

The language also provides foldt, a parallel version of fold that operates over a *set* of channels. This allows the efficient expression of typical data processing operations, such as a *k*-way merge sort in which sorted streams of keys from *k* channels are combined by selecting elements with the smallest key. The expression foldt f o cs aggregates elements from an array of channels cs, selecting elements according to a function o and aggregating according to a function f. As f must be commutative and associative, the aggregation can be performed in parallel, combining elements in a pair-wise manner until only the result remains.

As shown in Listing 3, the foldt primitive can be used to implement an application-level network service for parallel data aggregation in Hadoop. Whenever key/value pairs become available from the mappers (lines 5–6), foldt is invoked (lines 7–10). Elements elem are ordered based on elem.key (line 8), and values of elements with the same key (e_key) are merged using a combine function (line 9) to create a new key/value pair (line 10).While foldt could be expressed using core language primitives, the FLICK platform has a custom implementation for performance reasons.

While designed to achieve higher safety and performance, the constraints introduced in the design of the FLICK language, e.g. the lack of support for unbounded computation or dynamic memory allocation, imply that not all possible computations can be expressed in FLICK. For instance, algorithms requiring loops with unbounded iterations (e.g. while-like loops) cannot be encoded. In a general purpose programming language, this would be a severe constraint but for the middlebox functionality that FLICK targets we have not found this to cause major limitations.

5 FLICK Platform

The FLICK platform is designed around a *task graph* abstraction, composed of tasks that deserialise input data to typed values, compute over those values, and serialise results for onward transmission. The FLICK compiler translates an input FLICK program to C++, which is in turn compiled and linked against the platform runtime for execution. Figure 2 shows an overview of the FLICK platform, which handles network connections, the task graph life-cycle, the communication between tasks and the assignment of tasks to worker threads. Task graphs exploit task and data parallelism at runtime as tasks are
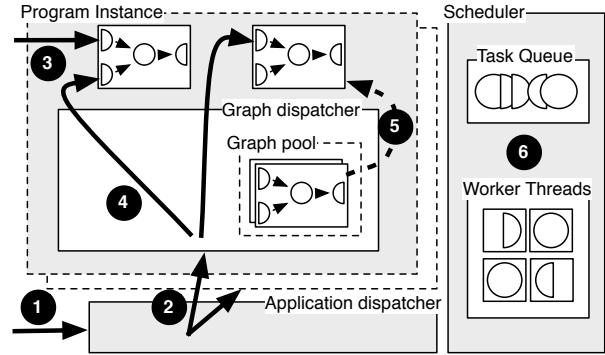

Figure 2: Main components of the FLICK platform

assigned to worker threads. Even with only one large network flow, serialisation, processing and deserialisation tasks can be scheduled to run on different CPU cores.

(i) The *application dispatcher* manages the life-cycle of TCP connections: first it maps new incoming connections ❶ to a specific program instance ❷, typically based on the destination port number of the incoming connection. The application dispatcher manages the listening sockets that handle incoming connections, creating a new input channel for each incoming connection and handing off data from that connection to the correct instance. When a client closes an input TCP connection, the application dispatcher indicates this to the instance; when a task graph has no more active input channels, it is shut down. New connections are directly connected to existing task graphs ❸.

(ii) The *graph dispatcher* assigns incoming connections to task graphs ❹, instantiating a new one if none suitable exists. The platform maintains a pre-allocated pool of task graphs to avoid the overhead of construction ❺. The graph dispatcher also creates new output channel connections to forward processed traffic.

(iii) Tasks are cooperatively scheduled by the *scheduler*, which allocates work among a fixed number of worker threads ❻. The number of worker threads is determined by the number of CPU cores available, and worker threads are pinned to CPU cores.

Tasks in a task graph become runnable after receiving data in their input queues (either from the network or from another task). A task that is not currently executing or scheduled is added to a worker queue when it becomes runnable. All buffers are drawn from a pre-allocated pool to avoid dynamic memory allocation. Input tasks use non-blocking sockets and epoll event handlers to process socket events. When a socket becomes readable, the input task attached to the relevant socket is scheduled to handle the event.

For scheduling, each worker thread is associated with its own FIFO task queue. Each task within a task graph has a unique identifier, and a hash over this identifier de-
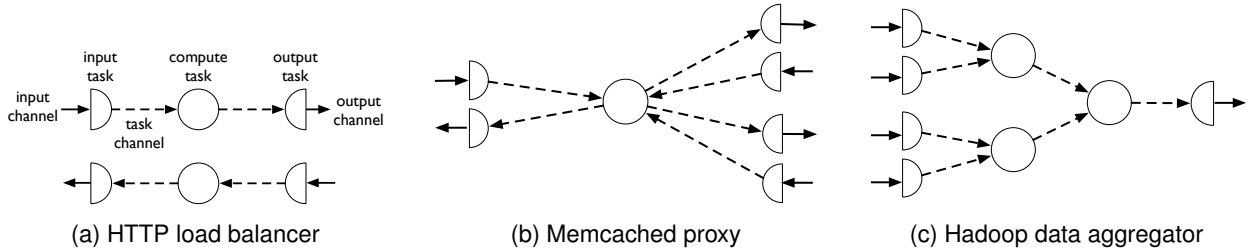
Figure 3: Task graphs for different application-specific network services

termines which worker's task queue the task should be assigned to. When a task is to be scheduled, it is always added to the same queue to reduce cache misses.

Each worker thread picks a task from its own queue. If its queue is empty, the worker attempts to scavenge work from other queues and, if none is found, it sleeps until new work arrives. A worker thread runs a task until either all its input data is consumed, or it exceeds a system-defined time quantum, the *timeslice threshold* (typically, 10–100 μs; see §6). If the timeslice threshold is exceeded, the code generated by the FLICK compiler guarantees that the task re-enters the scheduler, placing itself at the back of the queue if it has remaining work to do. A task with no work is not added to the task queue, but when new items arrive in its input channels, it is scheduled again.

A disadvantage of allocating tasks belonging to the same task graphs onto different CPU cores is that this would incur several cache invalidations as data move from one core to another. On the other hand, our design enables higher parallelism as different tasks can execute concurrently in a pipelined fashion, leading to higher throughput.

Some middlebox services must handle many concurrent connections, and they frequently write and read small amounts of data. The kernel TCP stack has a high overhead for creating and destroying sockets to support the Linux Virtual File System (VFS) interface [17]. Socket APIs also require switching between user- and kernel-mode, which adds further overhead. As a result, the FLICK platform uses mTCP [21], a highly scalable user-space TCP stack, combined with Intel's DPDK [20] to reduce these overheads. The original mTCP implementation did not support multi-threaded applications, and we modified mTCP so that Flick I/O tasks can access sockets independently. To take utilise the efficient DPDK runtime environment, mTCP executes as a DPDK task. All of these optimisations, significantly improve performance for network-bound services (see §6.3).

6 Evaluation

The goals of our evaluation are to investigate whether the high-level programming abstraction that FLICK carries a performance and scalability cost and whether DPDK and mTCP improve performance. We implement FLICK programs for the use cases introduced in §2.1, i.e. an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator, and compare their performance against baselines from existing implementations.

After describing the implementation of our use cases (§6.1) and the experimental set-up (§6.2), we explore the performance and scalability of FLICK (§6.3). After that, we examine how well the FLICK platform isolates resource consumption of multiple FLICK programs using cooperative scheduling (§6.4).

6.1 Use case implementation

For our three use cases, Figure 3 shows the task graph obtained from the corresponding FLICK program.

HTTP load balancer. This FLICK program implements an HTTP load balancer that forwards each incoming HTTP request to one of a number of backend web servers. Forwarding is based on a naive hash of the source IP and port and destination IP and port. Figure 3a shows the corresponding task graph. The application dispatcher forwards each new TCP connection received on port 80 to the graph dispatcher. The graph dispatcher creates a new task graph, which is later destroyed when the connection closes. The input task deserialises the incoming data into HTTP requests. For the first request, the compute task calculates a hash value selecting a backend server for the request. Subsequent requests on the same connection are forwarded to the same backend server. On their return path no computation or parsing is needed, and the data is forwarded without change. We also implement a variant of the HTTP load balancer that does not use backend servers but which returns a fixed response to a given request. This is effectively a static web server, which we use to test the system without backends.

Memcached proxy. In this use case, the FLICK program (Listing 1) receives Memcached look-up requests for keys. Requests are forwarded based on hash partitioning to a set of Memcached servers, each storing a disjoint section of the key space. Responses received from the Memcached servers are returned to clients.

Figure 3b shows the corresponding task graph. As before, a new task graph is created for each new TCP connection. Unlike the HTTP load balancer, requests

from the same client can be dispatched to different Memcached servers, which means that the compute task must have a fan-out greater than one.

When a request is received on the input channel, it is deserialised by the input task. The deserialisation code is automatically generated from the type specification in Listing 2. The deserialiser task outputs the Memcached request object, containing the request keys and body, which are passed on to the compute task. The compute task implements the dispatching logic. It identifies the Memcached server responsible for that key and forwards the request to it through the serialiser task. When the response is received from the Memcached server, the deserialiser task deserialises it and passes the response object to the compute task, which returns it to the client through the serialiser task.

Hadoop data aggregator. The Hadoop data aggregator implements the combiner function of a map/reduce job to perform early data aggregation in the network, as described in §2.1. It is implemented in FLICK according to Listing 3. We focus on a wordcount job in which the combiner function aggregates word counters produced by mappers over a set of documents.

For each Hadoop job, the platform creates a separate task graph per reducer (Figure 3c). The input tasks deserialise the stream of intermediate results (i.e. key/value pairs) from the mappers. Compute tasks combine the data with each compute task taking two input streams and producing one output. The output task converts the data to the byte stream, as per the Hadoop wire format.

6.2 Experimental set-up

We deploy the prototype implementation of the FLICK platform on servers with two 8-core Xeon E5-2690 CPUs running at 2.9 Ghz with 32 GB of memory. Clients and back-end machines are deployed on a cluster of 16 machines with 4-core Xeon E3-1240 CPUs running at 3.3 Ghz. All machines use Ubuntu Linux version 12.04. The clients and backend machines have 1 Gbps NICs, and the servers executing the FLICK platform have 10 Gbps NICs. The client and backend machines connect to a 1 Gbps switch, and the FLICK platform connects to a 10 Gbps switch. The switches have a 20 Gbps connection between them. We examine the performance of FLICK with and without mTCP/DPDK.

To evaluate the performance of the HTTP load balancer, we use multiple instances of *ApacheBench* (ab) [4], a standard tool for measuring web server performance, together with 10 backend servers that run the *Apache* web server [51]. Throughput is measured in terms of connections per second as well as requests per second for HTTP keep-alive connections. We compare against the standard Apache (*mod_proxy_balancer*) and the Nginx [35] load balancers.

For the Memcached proxy, we deploy 128 clients running *libmemcached* [1], a standard client library for interacting with Memcached servers. We use 10 Memcached servers as backends and compare the performance against a production Memcached proxy, *Moxi* [32]. We measure performance in terms of throughput (i.e. requests served per second) and request latency. Clients send a single request and wait for a response before sending the next request.

For the Hadoop data aggregator, the workload is a *wordcount* job. It uses a sum as the aggregation computation and an input dataset with a high data reduction ratio. The datasets used in experiments are 8 GB, 12 GB and 16 GB (larger data sets were also used for validation). Here we measure performance in terms of the absolute network throughput.

In all graphs, the plotted points are the mean of five runs with identical parameters. Error bars correspond to a 95% confidence interval.

6.3 Performance

HTTP load balancer. We begin by measuring the performance of the static web server with an increasing load. This exercises the following components of the FLICK platform: HTTP parsing, internal and external channel operation and task scheduling. The results are for 100 to 1,600 concurrent connections (above these loads, Apache and Nginx begin to suffer timeouts). Across the entire workload, FLICK achieves superior performance. It achieves a peak throughput of 306,000 requests/sec for the kernel version and 380,000 requests/sec with mTCP. The maximum throughput achieved by Apache is 159,000 requests/sec and by Nginx is 217,000 requests/sec. FLICK also shows lower latency, particularly at high concurrency when Apache and Nginx use large numbers of threads. This confirms that, while FLICK provides a general-purpose platform for creating application-specific network functions, it can outperform purpose-written services.

To investigate the per-flow overhead due to TCP setup/tear-down, we also repeat the same experiment but with each web request establishing a separate TCP connection (i.e. non-persistent HTTP). This reduces the throughput in all deployments: 35,000 requests/sec for Apache; 44,000 requests/sec for Nginx; and 45,000 requests/sec for FLICK, which maintains the lowest latency. Here the kernel TCP performance for connection set-up and tear-down is a bottleneck: the mTCP version of FLICK handles up to 193,000 requests/sec.

Next, we repeat the experiment using our HTTP load balancer implementation to explore the impact of both receiving and forwarding requests. The set-up is as described in §6.2. We use small HTTP payloads (137 bytes each) to ensure that the network and the backends are
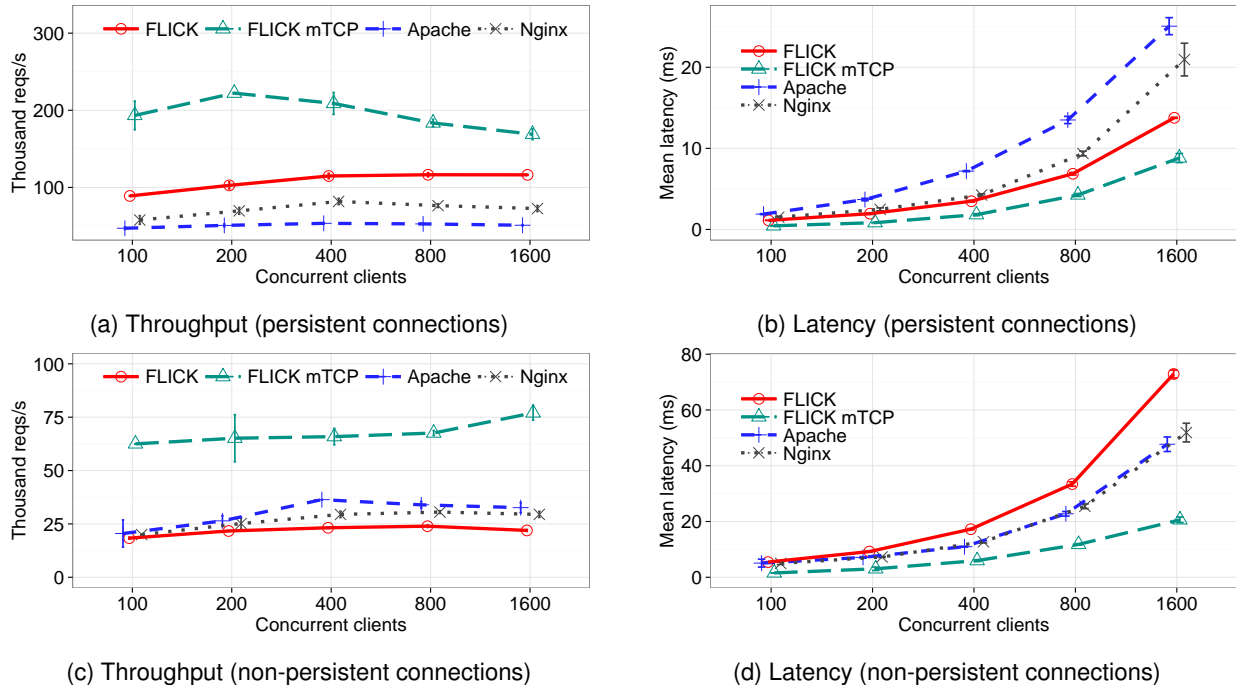
(a) Throughput (persistent connections)

(b) Latency (persistent connections)

(c) Throughput (non-persistent connections)

(d) Latency (non-persistent connections)

Figure 4: HTTP load balancer throughput and latency for an increasing number of concurrent connections

never the bottleneck. As for the web server experiment, we first consider persistent connections. Figures 4a and 4b confirm the previous results: FLICK achieves up to 1.4× higher throughput than Nginx and 2.2× higher than Apache. Using mTCP, the performance is even better with higher throughput and lower latency: FLICK achieves a maximum throughput 2.7× higher than Nginx and 4.2× higher than Apache. In all cases, FLICK has lower latency.

With non-persistent connections, the kernel version of FLICK exhibits a lower throughput than Apache and Nginx (see Figure 4c). Both Apache and Nginx keep persistent TCP connections to the backends, but FLICK does not, which increases its connection set-up/tear-down cost. When mTCP is used with its lower per connection cost, FLICK shows better performance than either with a maximum throughput 2.5× higher than that of Nginx and 2.1× higher than that of Apache. In addition, both the kernel and mTCP versions of FLICK maintain the lowest latency of the systems, as shown in Figure 4d.

Memcached proxy. For the Memcached proxy use case, we compare the performance of FLICK against *Moxi* [32], as we increase the number of CPU cores. We chose *Moxi* because it supports the binary Memcached protocol and is multi-threaded. In our set-up, 128 clients make concurrent requests using the Memcached binary protocol over persistent connections, which are then multiplexed to the backends.

Figures 5a and 5b show the throughput, in terms of the number of requests/sec, and the latency, respectively.

With more CPU cores, the throughout initially increases for both systems. The kernel version achieves a maximum throughput of 126,000 requests/sec with 8 CPU cores and the mTCP version achieves 198,000 requests/sec with 16 CPU cores. Moxi peaks at 82,000 requests/sec with 4 CPU cores. FLICK's latency decreases with more CPU cores due to the larger processing capacity available in the system. The latency of Moxi beyond 4 CPU cores and FLICK's beyond 8 CPU cores increases as threads compete over common data structures.

Hadoop data aggregator. The previous use cases had relatively simple task graphs (see Figure 3) and considerable overhead comes from the connection set-up and tear-down, with many network requests processed in parallel. In contrast, the Hadoop data aggregator use case has a more complex task graph, and we use it to assess the overhead of FLICK's communication channels and intra-graph scheduling. Here the tasks are compute bound, and the impact of the network overhead is limited. We only present the kernel results because the mTCP results are similar.

We deploy 8 mappers clients, each with 1 Gbps connections, to connect to the FLICK server. The task graph therefore has 16 tasks (8 input, 7 processing and 1 output). The FLICK Hadoop data aggregator runs on a server with 16 CPU cores without hyper-threading.

Figure 6 shows that FLICK scales well with the number of CPU cores, achieving a maximum throughput of 7,513 Mbps with 16 CPU cores. This is the maximum capacity of the 8 network links (once accounted for TCP
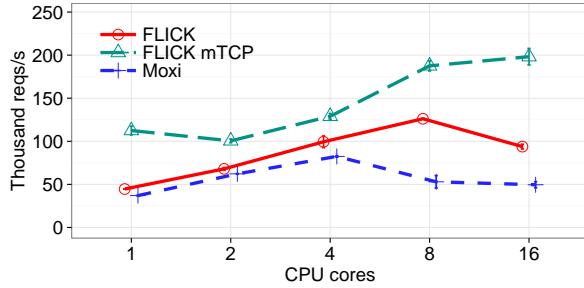
(a) Throughput
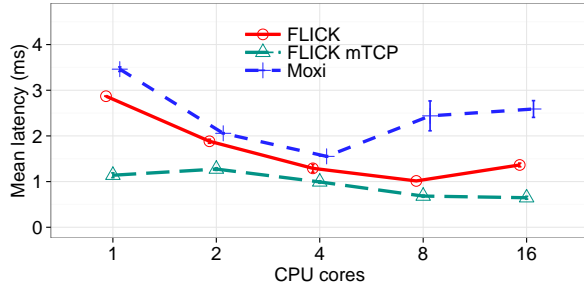
(b) Latency

Figure 5: Memcached proxy throughput and latency versus number of CPU cores used
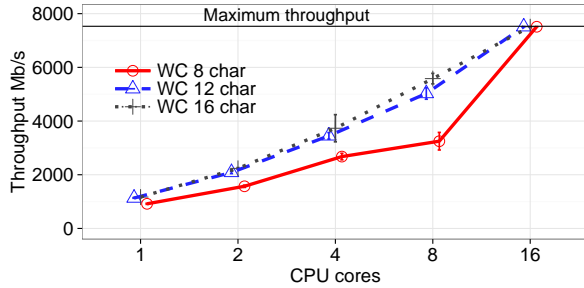

Figure 6: Median throughput for Hadoop data aggregator versus number of CPU cores

overhead), and matches measurements from `iperf`. We conclude that the FLICK platform can exploit the high level of parallelism of multi-core servers and efficiently schedule multiple tasks concurrently to maximise network throughput.

The results in Figure 6 represent three data sets of 8 GB, 12 GB and 16 GB mentioned in §6.2, consisting of words of 8, 12 and 16 characters, respectively. The FLICK platform can more efficiently process the longer words because they comprise fewer key value pairs.

6.4 Resource sharing

We finish our experiments by examining the ability of the FLICK platform to ensure efficient resource sharing, as described in §3.1. For this, we use a micro-benchmark running 200 tasks. Each task consumes a finite number of data items, computing a simple addition for each input byte. The tasks are equally split between two classes:
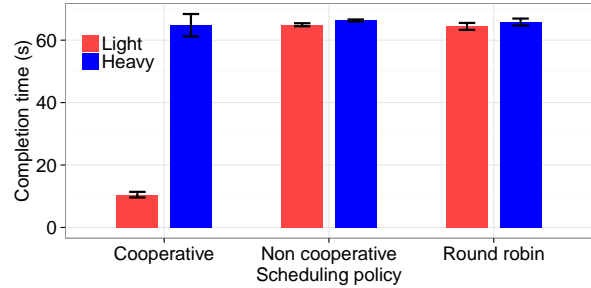

Figure 7: Completion time for "light" and "heavy" tasks with three scheduling policies

light tasks operate on 1 KB data items; and *heavy* tasks operate on 16 KB data items. We consider three scheduling policies: (i) *cooperative* is the policy used by FLICK, in which each task is given a fixed amount of CPU time before it yields control to another task; (ii) *non-cooperative* runs a scheduled task to completion, potentially letting the OS scheduler preempt it; and (iii) *round robin* schedules each task for one data item only.

Figure 7 shows the total completion time for light and heavy tasks. Since the light tasks handle less data, they should, given a fair share of resources, finish before the heavy tasks. With the *round robin* policy, this does not happen: the heavy tasks take longer to process one data item. Each time they are scheduled, they occupy the worker thread for longer than a light task. Conversely, with the *non-cooperative* policy, each task runs to completion. The total completion time for the light and heavy tasks is determined by their scheduling order. However, with FLICK's *cooperative* policy, the light tasks are allowed to complete ahead of the heavy tasks without increasing the overall runtime—each task is given a fair share of the CPU time.

7 Related Work

Network programming languages are essential to the usability and scalability of *software-defined networking* (SDN), allowing high-level configuration logic to be translated to low-level network operations. Inspired by *Frenetic* [16], *NetKAT* [2] is a high-level network programming language based on Kleene algebra [25], in which network policies are compiled into a low-level programming abstraction such as *OpenFlow* [31] flow tables. Similarly, the *Policy Graph Abstraction* (PGA) [41] expresses network policies as a coherent, conflict-free policy set and supports automated, correct and independent composition of middlebox policies. These systems focus on network management and configuration and not on the more expressive programs for application-specific network services that are targeted by FLICK.

FLICK ensures that programs execute in a timely manner by restricting the expressiveness of the programming language. Another possible approach is to explicitly ver-

ify that a specific program meets requirements. This verification apparoach has been used to check simple, stateless Click pipelines [12] but might be harder for more complex middlebox programs.

There are proposed extensions to the packet processing done by OpenFlow. *P4* [8] is a platform- and protocol-independent language for packet processors, which allows the definition of new header fields and protocols for use in match/action tables. *Protocol Oblivious Forwarding* (POF) [47] also provides a flexible means to match against and rewrite packet header fields. *Packet Language for Active Networks* (PLAN) [18] is a stateless and strongly-typed functional language for active networking in which packets carry programs to network nodes for execution. In general, these approaches are limited to expressing control-plane processing of packets in contrast to FLICK, which deals with application layer data.

Software middlebox platforms. Recently network services have been deployed on commodity hardware to reduce costs and increase flexibility. *Click* [24] processes packets through a chain of installed elements, and it supports a wide variety of predefined elements. Programmers, however, must write new elements in C++, which can be error-prone. *ClickOS* [30] combines Click with MiniOS and focuses on the consolidation of multiple software middlebox VMs onto a single server. It overcomes current hypervisor limitations through a redesigned I/O system and by replacing Open vSwitch [38] with a new software switch based on VALE [45]. ClickOS targets packet level processing, e.g. manipulating header fields or filtering packets; FLICK, by contrast, operates at the application level, and the approaches can be seen as orthogonal. It would be challenging for ClickOS to parse and process HTTP data when a single data item may span multiple packets or Memcached data when a packet may contain multiple data items.

Merlin [48] is a language that safely translates policies, expressed as regular expressions for encoding paths, into Click scripts. Similarly, *IN-NET* [49] is an architecture for the deployment of custom in-network processing on ClickOS with an emphasis on static checking for policy safety. In a similar vein, *xOMB* [3] provides a modular processing pipeline with user-defined logic for flow-oriented packet processing. *FlowOS* [7] is a flow-oriented programmable platform for middleboxes using a C API similar to the traditional socket interface. It uses kernel threads to execute flow-processing modules without terminating TCP connections. Similar to ClickOS, these platforms focus on packet processing rather than the application level. *SmartSwitch* [53] is a platform for high-performance middlebox applications built on top of NetVM [19], but it only supports UDP applications, and it does not offer a high-level programming model.

Eden is a platform to execute application-aware network services at the end hosts [5]. It uses a domain-specific language, similar to F#, and enables users to implement different services ranging from load balancing to flow prioritisation. By operating at the end hosts, it limits the set of network services that can be supported. For example, it would be impossible to implement in-network aggregation or in-network caching.

Split/Merge [42] is a hypervisor-level mechanism that allows balanced, stateful elasticity and migration of flow state for virtual middleboxes. Per-flow migration is accomplished by identifying the external state of network flows, which has to be split among replicas. Similar elasticity support could be integrated with FLICK.

8 Conclusions

Existing platforms for in-network processing typically provide a low-level, packet-based API. This makes it hard to implement application-specific network services. In addition, they lack support for low-overhead performance isolation, thus preventing efficient consolidation.

To address these challenges, we have developed FLICK, a domain-specific language and supporting platform that provides developers with high-level primitives to write generic application-specific network services. We described FLICK's programming model and runtime platform. FLICK realises processing logic as restricted cooperatively schedulable tasks, allowing it to exploit the available parallelism of multi-core CPUs. We evaluated FLICK through three representative use cases, an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator. Our results showed that FLICK greatly reduces the development effort, while achieving better performance than specialised middlebox implementations.

References

[1] AKER, B. libmemcached. http://libmemcached.org/libMemcached.html.

[2] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *Proc. 41th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2014).

[3] ANDERSON, J. W., BRAUD, R., KAPOOR, R., PORTER, G., AND VAHDAT, A. xOMB: Extensible Open Middleboxes with Commodity Servers . In *Proc. 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2012).

[4] APACHE FOUNDATION. Apache HTTP Server Benchmarking Tool, 2015. https://httpd.apache.org/docs/2.2/programs/ab.html.

[5] BALLANI, H., COSTA, P., GKANTSIDIS, C., GROSVENOR, M. P., KARAGIANNIS, T., KOROMILAS, L., AND O'SHEA, G. Enabling End-host Network Functions. In *Proc. of ACM SIGCOMM* (2015).

[6] BANGERT, J., AND ZELDOVICH, N. Nail: A Practical Tool for Parsing and Generating Data Formats. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[7] BEZAHAF, M., ALIM, A., AND MATHY, L. FlowOS: A Flow-based Platform for Middleboxes. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2013).

[8] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[9] CHOWDHURY, M., KANDULA, S., AND STOICA, I. Leveraging Endpoint Flexibility in Data-intensive Clusters. In *Proc. of ACM SIGCOMM* (2013).

[10] COSTA, P., MIGLIAVACCA, M., PIETZUCH, P., AND WOLF, A. L. NaaS: Network-as-a-Service in the Cloud. In *Proc. 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)* (2012).

[11] DAGAND, P.-E., BAUMANN, A., AND ROSCOE, T. Filet-o-fish: Practical and Dependable Domain-specific Languages for OS Development. *SIGOPS Oper. Syst. Rev. 43*, 4 (2010), 35–39.

[12] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *Proc. USENIX Networked Systems Design and Implementation (NSDI)* (2014), pp. 101–114.

[13] DONOVAN, S., AND FEAMSTER, N. Intentional Network Monitoring: Finding the Needle without Capturing the Haystack. In *Proc. 13th ACM Workshop on Hot Topics in Networks (HotNets)* (2014).

[14] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1, June 1999. `http://ietf.org/rfc/rfc2616.txt`.

[15] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal 2004*, 124 (2004).

[16] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A Network Programming Language. In *Proc. 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (2011).

[17] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

[18] HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. PLAN: A Packet Language for Active Networks. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP)* (1998).

[19] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[20] INTEL. Intel Data Plane Development Kit (DPDK). http://www.intel.com/go/dpdk, 2014.

[21] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[22] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A Safe Dialect of C. In *Proc. USENIX Annual Technical Conference (ATC)* (2002).

[23] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proc. European Conference on Computer Systems (EuroSys)* (2013).

[24] KOHLER, E., MORRIS, R., , CHEN, B., JANNOTTI, J., AND KAASHOEKBENJIE, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (2000), 263–297.

[25] KOZEN, D., AND SMITH, F. Kleene Algebra with Tests: Completeness and Decidability. In *Proc. 10th International Workshop on Computer Science Logic (CSL)* (1996).

[26] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative Networking. *Commun. ACM 52*, 11 (2009), 87–95.

[27] MAAS, M., ASANOVIĆ, K., HARRIS, T., AND KUBIATOWICZ, J. The Case for the Holistic Language Runtime System. In *Proc. 1st International Workshop on Rack-scale Computing (WRSC)* (2014).

[28] MADHAVAPEDDY, A., HO, A., DEEGAN, T., SCOTT, D., AND SOHAN, R. Melange: Creating a "Functional" Internet. *SIGOPS Oper. Syst. Rev. 41*, 3 (2007), 101–114.

[29] MAI, L., RUPPRECHT, L., ALIM, A., COSTA, P., MIGLIAVACCA, M., PIETZUCH, P., AND WOLF, A. L. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proc. 10th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2014).

[30] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the Art of Network Function Virtualization. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[31] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (2008), 69–74.

[32] MEMBASE. Moxi – Memcached Router/Proxy, 2015. `https://github.com/membase/moxi/wiki`.

[33] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-defined Networks. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[34] NETFLIX. Ribbon Wiki. `https://github.com/Netflix/ribbon/wiki`.

[35] NGINX Website. `http://nginx.org/`.

[36] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[37] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2008).

[38] Open vSwitch. `http://openvswitch.org/`.

[39] PANG, R., PAXSON, V., SOMMER, R., AND PETERSON, L. Binpac: A Yacc for Writing Application Protocol Parsers. In *Proc. 6th ACM SIGCOMM Conference on Internet Measurement (IMC)* (2006).

[40] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREEN-BERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proc. of ACM SIGCOMM* (2013).

[41] PRAKASH, C., LEE, J., TURNER, Y., KANG, J.-M., AKELLA, A., BANERJEE, S., CLARK, C., MA, Y., SHARMA, P., AND ZHANG, Y. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proc. of ACM SIGCOMM* (2015).

[42] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[43] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX Annual Technical Conference (ATC)* (2012).

[44] RIZZO, L., CARBONE, M., AND CATALLI, G. Transparent Acceleration of Software Packet Forwarding Using Netmap. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)* (2012).

[45] RIZZO, L., AND LETTIERI, G. VALE, a Switched Ethernet for Virtual Machines. In *Proc. 8th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2012).

[46] SOMMER, R., WEAVER, N., AND PAXSON, V. HILTI: An Abstract Execution Environment for High-Performance Network Traffic Analysis. In *Proc. 14th ACM SIGCOMM Conference on Internet Measurement (IMC)* (2014).

[47] SONG, H. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)* (2013).

[48] SOULÉ, R., BASU, S., MARANDI, P. J., PEDONE, F., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Merlin: A Language for Provisioning Network Resources. In *Proc. 10th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2014).

[49] STOENESCU, R., OLTEANU, V., POPOVICI, M., AHMED, M., MARTINS, J., BIFULCO, R., MANCO, F., HUICI, F., SMARAGDAKIS, G., HANDLEY, M., AND RAICIU, C. In-Net: In-Network Processing for the Masses. In *Proc. European Conference on Computer Systems (EuroSys)* (2015).

[50] STONE, E. Memcache Binary Protocol, 2009. `https://code.google.com/p/memcached/wiki/BinaryProtocolRevamped`.

[51] THE APACHE SOFTWARE FOUNDATION. The Apache HTTP Server Project. `http://httpd.apache.org/`.

[52] TWITTER. Twemproxy (nutcracker). `https://github.com/twitter/twemproxy`.

[53] ZHANG, W., WOOD, T., RAMAKRISHNAN, K., AND HWANG, J. SmartSwitch: Blurring the Line Between Network Infrastructure & Cloud Applications. In *Proc. 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2014).

[54] Apache Hadoop. `http://hadoop.apache.org`.