# Cuttlefish: Hierarchical SDN Controllers with Adaptive Offload

Rinku Shah, Mythili Vutukuru, Purushottam Kulkarni
*Department of Computer Science and Engineering*
*Indian Institute of Technology Bombay*
{rinku, mythili, puru}@cse.iitb.ac.in

***Abstract*—**
**Offloading computation to local controllers (closer to switches) has been a popular approach to designing scalable SDN controllers. We observe that, in addition to the offload of local switch-specific state, a subset of global state can also be offloaded to, and accessed at local controllers with suitable synchronization. We present the design and implementation of Cuttlefish, an SDN controller framework that adaptively offloads a portion of the application state (and computation) to local controllers. Cuttlefish uses developer-specified input to identify control messages that can be correctly processed at local controllers, and makes offloading decisions based on the cost of synchronizing the offloaded state across controllers. SDN applications use the Cuttlefish API to access the offloaded state, and Cuttlefish transparently manages the state synchronization, and redirection of control messages to the appropriate (central or local) controller. We have implemented Cuttlefish using the Floodlight SDN controller. Our evaluation shows that Cuttlefish applications achieve ∼2X higher control plane throughput and ∼50% lower control plane latency as compared to the traditional SDN design.**

***Index Terms*—software-defined networking, scalability, controller framework**

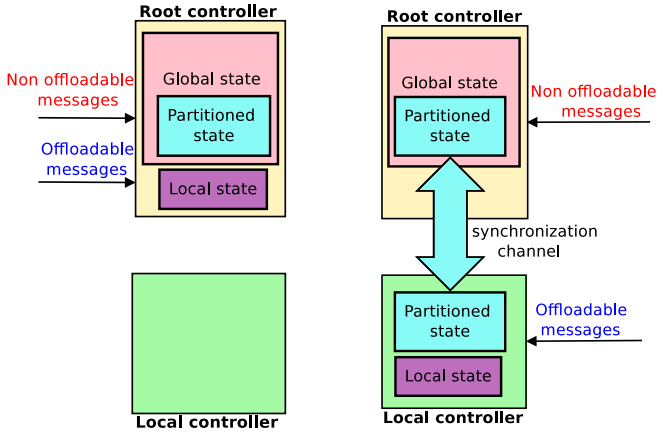
## I. Introduction

Software Defined Networking (SDN) is a design paradigm of separating the control and data planes of networking elements. A software defined network consists of a software-managed, logically centralized controller, and light-weight switches that are programmed with forwarding rules by the controller. Any data plane traffic for which rules do not exist, or signaling messages that require control plane processing, are directed to the controller by the switches. SDN "applications" running at the controller process these messages and install corresponding forwarding rules on the data plane switches.

Prior work (§V) has identified several scalability problems with centralized SDN controllers and has proposed solutions to fix the same. One set of solutions [1]–[3] develop *horizontally-scalable distributed SDN controllers* that spawn multiple controller replicas to scale controller capacity. The replicas use standard synchronization techniques to distribute application state and associated computation between themselves. Other solutions [4]–[6] propose *hierarchical SDN controllers* which offload computation that does not require network-wide view to *local* controllers running on (or close to) the switches. For example, traffic engineering applications that detect flows with large number of packets (elephant flows) before calculating optimal routes can offload the task of detecting large flows to local controllers. The local controllers maintain local state of switch flow statistics, while the central *root* controller only runs route computations that require global view. This decoupled computation setup results in a lower computation load at the root controller, and reduces the network traffic between the switches and the root controller.

Prior work [7] implicitly classifies an SDN application's state into *global network-wide state* (that pertains to, or is accessed by, multiple switches/entities in the network) and *local switch-specific state*. Local state can be easily maintained at local controllers, and control plane messages that depend on such state can be offloaded to local controllers. Global state must be maintained at the root controller (or with tight synchronization across distributed controllers), and control plane messages that access global state must necessarily be processed at the root (or its synchronized replicas). The key observation of our work is that, beyond the dichotomy of local and global state, there is a third type of state that we refer to as *partitioned state*. Partitioned state is a subset of global state that can be cached at local controllers, and can be accessed like local state during the processing of some control plane messages. These control plane messages, which we call *offloadable* messages, access this partitioned state (and local state) from a single network location and do not require concurrent access to any other non-local state. The key idea of our work is that, by synchronizing partitioned state from the root controller to specific local controllers, the messages that access this partitioned state (offloadable messages) can be offloaded to local controllers. This offload can lower the computation overhead at the central root controller, resulting in higher control plane capacity, and lower latency for the SDN application. We refer to this new mode of operation of an SDN application as the *offload mode* of operation, as shown in figure 1(b). In offload mode, partitioned state and local state resides at local controllers, and offloadable messages that access such state are handled locally. The updates to the partitioned state are synchronized between the root and local controllers. In contrast, when operating in the default *centralized mode* (Figure 1(a)), all application state resides at the central root controller, and all control plane messages (offloadable and otherwise) are processed at the root (or one of its replicas in a distributed framework).

We now describe one use case that motivates our work. Several researchers [8] [9] [10] [11] [12] [13] have proposed




(a) Centralized mode. (b) Offload mode.

Fig. 1: SDN operation modes.

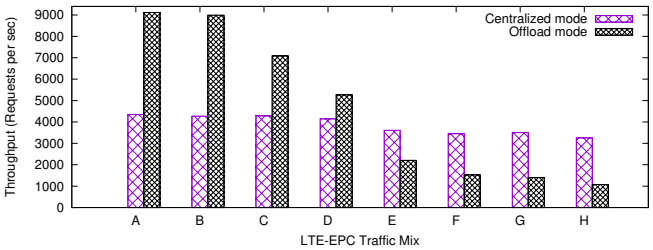



Fig. 2: Performance with different controller modes

that the 4G LTE packet core [14] can be decomposed into an SDN controller application that processes signaling messages, and simpler data plane switches that forward user data based on the state setup by the signaling messages. In this decomposed design of the LTE packet core, one important piece of application state called the user's forwarding context—the information needed to forward a user's data through the dataplane gateways—is an example of partitioned state. The user's context is created by a control message that registers the user when she turns on her data connection for the first time. Processing this registration message requires global network view and access to an authentication database, and must necessarily happen at the root controller. However, the signaling message sent when a user wants to reconnect after a small idle period requires access only to the user's context and no other global state. This is an example of an offloadable message that can be handled at a local controller closer to the ingress switch of the user, provided the user's context is synchronized with the local controller after registration. With a significant increase in signaling traffic in modern cellular networks [15], coupled with the observation that about half of this traffic consists of messages that transition the user between idle and active states [16], a framework that offloads these idle/active transition messages can significantly improve overall application scalability. We discuss other such usecases in our paper (§II) and show that several classes of SDN applications exhibit such partitioned state and offloadable messages.

*Does the offload mode of operation always improve performance?* Performing computation based on partitioned state at local controllers is beneficial only if the state at the local controller needs to be synchronized with the "master copy" at the root controller infrequently. If the traffic characteristics entail frequent updates to the partitioned state, there is frequent synchronization between the root and local controllers. This synchronization cost may outweigh the benefit of compute offload, and a traditional design that does not offload such state might work better. Figure 2 shows the throughput of the LTE packet core application, under various control plane traffic mixes, in both the centralized and offload modes of operation (details in §IV). In this experiment, the proportion of registration messages (that update the partitioned state at the root controller, resulting in synchronization between the root and local controllers) monotonically increases from traffic mix A to mix H, and the proportion of offloadable messages decreases. We can observe that offload mode performs better than centralized mode for traffic mixes A to D, because a significant fraction of the control plane messages are offloaded in offload mode, thereby improving the capacity of the SDN controller. However, for the rest of the traffic mixes, the centralized mode performs better, due to the high synchronization cost (in terms of CPU and network overhead) between the root and local controllers in offload mode. With traffic characteristics being dynamic in nature, an SDN controller framework must support offloading of partitioned state and associated computation adaptively between the centralized mode and offload mode based on the cost of synchronization, in order to optimize system performance.

This paper describes the design and implementation of Cuttlefish (§III), a hierarchical SDN controller framework that adaptively offloads computation from the root controller to local controllers at (or closer to) switches in order to optimize SDN application performance. Cuttlefish uses programmer input to identify the subset of control plane messages that are offloadable and can be correctly handled at local controllers. Beyond identifying offloadable messages at design time, the offload mechanism itself is completely transparent to the SDN applications. The application developer is required to write only a single SDN application with logic for handling and processing various control plane messages, and the same application runs at both the root and local controllers. The only change required to SDN applications running on Cuttlefish is that they must manage their partitioned state using the Cuttlefish API that provides functions to get/put/delete key-value pairs. When the system is operating in offload mode, our framework takes care of synchronizing the partitioned state between the root and local controllers, so that the correctness of the SDN application is maintained. Our framework monitors the cost of this synchronization, and periodically computes the appropriate mode (centralized vs. offload) for the application to operate in. This decision about the operational mode is enforced by pushing rules to the SDN switches to redirect

control plane messages to the suitable (root or local) controller.

We implement Cuttlefish using the Floodlight SDN controller and Open vSwitch SDN switches. We also implement three sample SDN applications— a key-value store, the LTE packet core and a stateful load balancer—to demonstrate the feasibility of our framework. We evaluate our framework using these applications under multiple traffic scenarios (§IV), and show that our framework adapts the amount of offload to traffic conditions correctly. Across applications, the performance of Cuttlefish matches that of the best operating mode (centralized or offload) for a given traffic mix. Cuttlefish improves LTE control plane throughput by ∼2X and control plane latency by ∼50% as compared to always operating in the centralized mode. For the key-value store application, Cuttlefish improves throughput by ∼3X and latency by ∼60% as compared to always operating in the offload mode. Further, we observe that the additional monitoring and metric collection of our adaptive algorithm imposes a negligible overhead, and our algorithm can identify and switch to the correct offload mode within 20-70 seconds of a shift in control traffic pattern.

The main contributions of our work can be summarized as follows: (i) We introduce a new taxonomy of state for SDN applications, beyond the existing notions of global network-wide state and local switch-specific state. The key idea of Cuttlefish is to offload computation that depends on partitioned state—a subset of global state that can be correctly cached and updated at local controllers—to improve the performance scalability of the centralized root controller; (ii) Our framework provides APIs to access partitioned state in an SDN application, and manages the synchronization of this state across the root and local controllers. Further, our framework monitors the cost of state synchronization across the root and local controllers, and automatically switches between the centralized and offload operation mode to maximize application performance, in a manner that is transparent to the application. Our open source framework [17] enables SDN application developers to easily leverage these features to improve the performance of their applications.

## II. Motivating Applications

While prior work on hierarchical SDN controllers only offloaded computation that depends on switch-local state, Cuttlefish proposes offloading computation that depends even on partitioned state—a subset of global application state that can be cached and accessed with suitable synchronization at local controllers. The usefulness of Cuttlefish therefore depends on whether enough applications exist with partitioned state, in order to benefit from the computation offload of our framework. We provide a few examples of such applications in this section.

### A. SDN-based LTE EPC

One of the network components being considered for control-data plane decomposition in telecom networks is the mobile cellular packet core, also called the LTE EPC (Long Term Evolution Evolved Packet Core). The EPC is part of

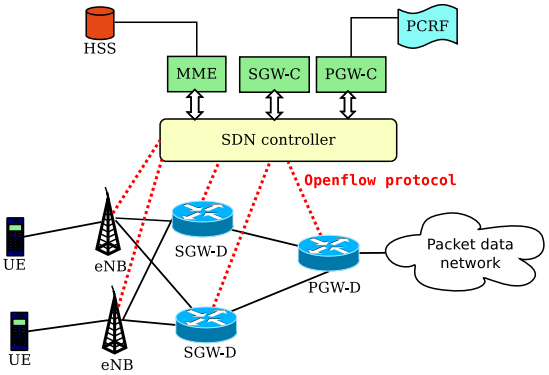



Fig. 3: Architecture of SDN-based LTE EPC.

the 4G LTE network that connects the wireless side of the network (the user and the base stations) to the rest of the Internet, as shown in Figure 3. The main network elements in the EPC are the Mobility Management Entity (MME) in the control plane, and the Serving and Packet Gateways (SGW and PGW) that forward user traffic in the data plane. Recent proposals to redesign the EPC using SDN principles (e.g., [8]) propose decomposing the control and data plane logic in the EPC gateways, and running the control logic of the packet gateways and the MME in an SDN controller. This design simplifies the EPC gateways and makes them more scalable, while making the LTE control plane in the software controller more flexible and amenable to new feature additions.

When a user equipment (UE) connects to a LTE network for the first time, the UE sends out a control plane *attach* request to the EPC to register itself. The MME processes the attach request, and sets up corresponding forwarding state for the user in the SGW and PGW. When the UE becomes active after an idle period, it generates a *service request* to restore the previous forwarding state that was released in the idle period at the dataplane gateways. The EPC also processes other control plane messages in addition to the attach request and the service request, e.g., a detach request to disconnect the UE from the network, and a handover request when the UE moves across the network.

With the traditional SDN architecture of the EPC, all signaling messages, including the attach/detach requests and service requests, are forwarded to the root controller. Now, most of the processing of an attach request must necessarily be done in the root controller because it requires access to a global database to authenticate the user, and creation of forwarding state requires access to the complete network topology. However, processing the service request requires only the forwarding state that was already created during the attach procedure, and can be entirely offloaded to a local controller, provided the forwarding state at the root is made available locally. A hierarchical SDN controller framework with local controllers at EPC gateway switches can thus efficiently offload computations like the service request processing from the root controller. However, if the traffic is predominantly composed of attach requests, the synchronization cost of propagating the forwarding state at the end of every attach request to local

controllers can outweigh the benefit of lowered computation load due to service request offload, and may ultimately lower the performance of the EPC application. Therefore, an adaptive framework like Cuttlefish offloads control plane messages such as the service request only after considering this tradeoff in realtime.

### B. SDN based Stateful Load Balancer

Consider a simple stateful load balancer that balances incoming connections among its current pool of servers based on the current load on the servers (measured by, say, the utilization of the servers, or the current number of ongoing connections at the servers). If this application were to be implemented within the SDN framework, the load balancer application running at the controller would maintain server load statistics, assign servers upon start of a new connection, and install forwarding rules to direct traffic to that server for all subsequent packets of the connection. In addition to new connection requests, the load balancer application will also handle messages to add/remove servers from the pool, updates related to the load level at the servers, and so on[1].

Now, if this SDN application were to be designed in a hierarchical SDN controller framework like Cuttlefish, one possible way to offload computation could be as follows: the root controller assigns subsets of servers to local controllers, and local controllers make load balancing decisions by assigning incoming connections to servers in their local pool. The root controller maintains the global view of server load statistics and moves servers across local pools based on the incoming load distribution across local controllers. The partitioned state in this design is the set of server statistics and the assignment of servers to local controller pools based on these statistics. Note that several other middleboxes like NATs and firewalls can be decomposed into hierarchical SDN applications in this manner—a subset of application state can be partitioned across local controllers, with each local controllers handling the part of the global state pertaining to its network location or traffic.

### C. Key-value store

We demonstrate a simple key-value store as another usecase for our framework. While a key-value store by itself is not a useful application to build on an SDN controller, it is part of the state management framework of several SDN applications. For example, the LTE EPC application discussed above uses a key-value store to store per-user forwarding contexts, per-user security related state, and so on. Therefore, we build a simple key-value store on the Cuttlefish framework, and implement three key-value tables that store global, partitioned, and local states respectively. In centralized mode, all the key-value stores reside at the root controller, whereas in offload mode, global key-value store reside at the root controller, local key-value store resides at the local controller and the partitioned state store can be accessed from both the local and root controllers, with suitable synchronization performed by the framework. Our implementation supports get/put/delete operations on non-partitioned global, partitioned, and local keys, in both centralized and offload modes.

[1]Our description of the load balancer application is somewhat simplistic, but captures the essence of real implementations.

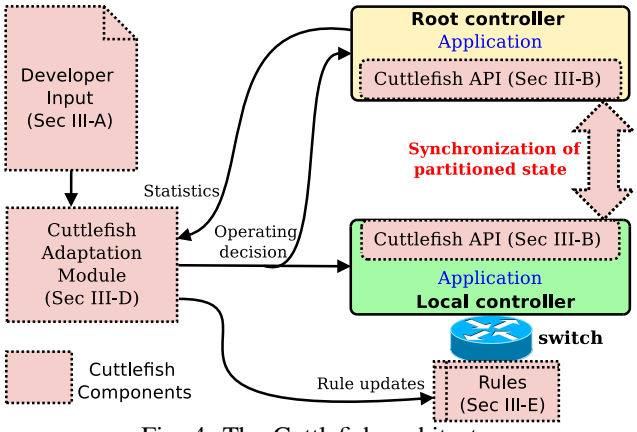


Fig. 4: The Cuttlefish architecture.

## III. Cuttlefish Design and Implementation

Figure 4 shows the architecture of Cuttlefish. Cuttlefish takes input from the application developer regarding the type of messages in the control plane and whether they are good candidates for offload (§III-A). SDN application developers write applications using Cuttlefish API (§III-B) functions to access partitioned state. The framework takes care of transparently synchronizing this state across root and local controllers based on the operating mode (§III-C). The heart of Cuttlefish is its adaptation module (§III-D) that measures the cost of synchronizing partitioned state and makes a decision on whether to operate the application in offload mode or centralized mode. The offload decision is enforced by the framework by pushing suitable rules into the data plane SDN switches (§III-E). When the adaptation module decides to switch between controller modes, Cuttlefish ensures that the migration of partitioned state and redirection of control plane traffic happen correctly without any race conditions (§III-F). Finally, we describe our implementation of sample SDN applications in the Cuttlefish framework (§III-G).

### A. Developer input

Cuttlefish requires the application developer to provide the following input: the types of messages in the control plane traffic, and whether each of these messages is offloadable or not. We assume that the control plane traffic to the application has a discrete, known number of message types, which can be identified by inspecting packets in the SDN switches[2]. The application developer provides rules to identify incoming message types as part of the input specification.

For each message type in the control plane traffic, the user specifies whether the message is offloadable or not. *How does*

[2]If the type of the control plane message cannot be identified by parsing standard L2-L4 headers alone, we assume that the switches are programmable using a language like P4 [18], in order to be able to parse application layer headers and identify the control plane message type.

*an application developer decide if a message can be offloaded to a local controller?* Given our definitions of global, local and partitioned state (§I), a control plane message is considered offloadable if processing the message requires access to only switch-local state and partitioned global state. That is, all global state accessed by the offloadable message must be amenable to caching at the local controller, and should not be accessed concurrently from any other network location during the processing of this message. We expect that application developers will have sufficient knowledge about application state semantics to be able to provide such an input. This expectation from the developers is common practice, and exists in prior work. For example Split/Merge [19] and OpenNF [20], provide APIs for moving state between distributed networking applications, and require the developer input to have a similar understanding of the semantics of application state. Table I shows an example of developer input for the LTE EPC Cuttlefish application, listing the types of messages in the control plane traffic of the EPC and whether they are offloadable.

| **Message type** | *msgID* | *offloadable* |
|---|---|---|
| Authentication Step 1 | 1 | false |
| Authentication Step 3 | 2 | false |
| NAS Step 2 | 3 | false |
| Send Access Point Name | 4 | false |
| Send UE Tunnel id (teid) | 5 | true |
| UE Context Release | 6 | true |
| UE Service Request | 7 | true |
| Context Setup Response | 8 | true |
| Detach Request | 9 | false |

TABLE I: Sample developer input for LTE EPC.

### B. The Cuttlefish API

Application developers within the Cuttlefish framework do not need to write separate applications to run at the root and local controllers. Instead, developers must use the Cuttlefish state management API to access partitioned state, and the framework takes care of transparently synchronizing this state across the controllers depending on the mode of operation. We assume all partitioned state can be stored as key-value pairs. Our API provides the following get/put/delete functions:

```
get(local_id, map_name, key)
put(msg_id, local_id, map_name, key, value)
delete(local_id, map_name, key)
```

The developer simply invokes our API functions when accessing the partitioned state in the application code, instead of invoking regular hashmap functions.

While a traditional SDN application may use a number of hashmaps to store the partitioned state, Cuttlefish stores all state in a single hashmap. Therefore, the API takes *map_name* as one of the parameters in the get/put/delete functions, and the key stored in Cuttlefish is a concatenation of this map name and the original key. Cuttlefish exposes a single hashmap to reduce the number of synchronization channels between the root and local controllers, thereby reducing synchronization overheads. To optimize the synchronization overheads further, we synchronize the partitioned state at the root controller only with the single local controller where the state is accessed. The *local_id* parameter in get/put/delete requests provides the local controller identification, and also identifies the partition of the synchronized hashmap to lookup at the root controller. To obtain the local controller identifier, the developer could use the default floodlight function to identify the ingress switch of a control plane message, which indirectly identifies the local controller. Finally, note that the parameter *msg_id* corresponding to the identifier of the message that generated the state update is part of the *put* API, in order to let our framework attribute synchronization costs to control plane messages (more details in §III-D).

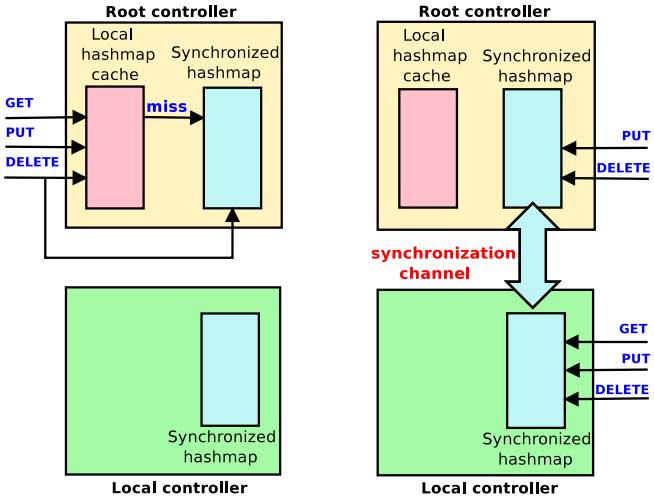
### C. Cuttlefish API Implementation

Accessing synchronized hashmaps is slightly slower than accessing local hashmaps due to additional mechanisms for consistency. Therefore, we also cache partitioned state temporarily in local hashmaps in centralized mode, because synchronization is not required in centralized mode. That is, application state in centralized mode is split between synchronized hashmaps (which would have been populated when the application was in offload mode) and the local hashmap cache (which is used when only in centralized mode). As shown in figure 5(a), all put operations in centralized mode are only applied to the local hashmap. Get operations are first performed on the local hashmap, and are applied in the synchronized hashmap only in case of a miss in the local cache. Delete operations are performed on both local and synchronized hashmaps for consistency.

When operating in offload mode, all offloadable messages are processed at local controllers, and all partitioned state accesses (get/put/delete) by the offloaded messages are performed on the synchronized hashmaps, as shown in Figure 5(b). All non-offloadable messages are handled at the root controller (e.g., because processing such messages depends on other global state), and these messages may also generate concurrent put/delete requests to the partitioned state. In order to optimize performance in offload mode, we batch updates to synchronized hashmaps at the local controller and push multiple updates at a time to the root controller. However, updates to partitioned state at the root controller are immediately pushed to the local controllers without batching, in order to ensure that the get operations at the local controller never see stale state[3].

We implement synchronized hashmaps and batching by extending the fault tolerance module of the open-source Floodlight SDN controller [21]. The Cuttlefish framework has TCP communication channels open between the root and local controllers to transport updates to the synchronized hashmaps. We batch up to 500 updates at a time at the local controller.

[3]We assume that processing non-offloadable messages does not result in any get operations on the partitioned state at the root controller, because messages that only read partitioned state would be offloadable and handled at the local controller itself. Therefore, we do not worry about stale state during get operations at the root.




(a) Centralized mode. (b) Offload mode.

Fig. 5: Cuttlefish API functions.

Note that we currently do not handle pending updates in a batch being lost due to the failure of the local controller. Our changes spanned about 350 lines of code in the Floodlight controller code base.

### D. The Adaptation Approach

The adaptation module of Cuttlefish can run as a separate application at the root controller or as a standalone application. It monitors the cost of synchronizing the partitioned state across the root and local controllers, and periodically decides the appropriate mode of operation (centralized vs. offload) of the SDN application. As discussed in §III-C, updates to partitioned state at the local controller are sent in batches, while updates at the root are propagated immediately. Therefore, updates to partitioned state at the root controller form a significant part of the synchronization cost, and form the basis for the decision algorithm in Cuttlefish.

The Cuttlefish adaptation module maintains a count of the cumulative number of put operations made to all partitioned state across all non-offloadable messages at the root controller, and computes the average rate of puts/sec every epoch. We use an epoch duration of 10 seconds in our implementation. At the end of each epoch, the put rate metric decides if Cuttlefish must switch modes. If Cuttlefish is operating in offload mode, and the average put rate crosses a threshold $\text{Thr}_{\text{Off\_Cent}}$, Cuttlefish switches the operating mode from offload to centralized, because it considers the synchronization cost to be too high to result in any performance benefits due to offload. When operating in centralized mode, Cuttlefish continues to monitor the averaged puts/sec metric, even though the put operations are performed on local hashmaps and the partitioned state is not synchronized with local controllers in the centralized mode. If the average put rate is found to be below a threshold $\text{Thr}_{\text{Cent\_Off}}$, Cuttlefish considers the synchronization overhead of partitioned state to be low enough and switches to offload mode of operation. The instrumentation to the Floodlight controller to gather the statistics of put operations, and the logic of the adaptation algorithm were implemented in about 200 lines of code.

TABLE II: Choice of thresholds for transition.

| CPU utilization | $\text{Thr}_{\text{Off\_Cent}}$ (#puts/sec) | $\text{Thr}_{\text{Cent\_Off}}$ (#puts/sec) |
|---|---|---|
| 30% | 600 | 400 |
| 55% | 1600 | 1400 |
| 75% | 2000 | 1800 |
| 80% | 2400 | 2200 |
| 90% | 2900 | 2700 |

The threshold values $\text{Thr}_{\text{Off\_Cent}}$ , and $\text{Thr}_{\text{Cent\_Off}}$ are configurable, and can be derived from the amount of CPU that the application programmer wishes to allocate towards synchronization related computation at the root controller. We have written a benchmark that executes put operations at a given rate on partitioned state in offload mode, and monitors the average CPU load at the root controller due to the synchronization overhead. This benchmark can be run by the application developer for different values of the put rate to identify the rate that corresponds to a maximum tolerable CPU burden at the root controller. This put rate can be used to determine the threshold $\text{Thr}_{\text{Off\_Cent}}$ to migrate from offload mode to centralized mode. The threshold $\text{Thr}_{\text{Cent\_Off}}$ can be chosen to be slightly lower than $\text{Thr}_{\text{Off\_Cent}}$, to ensure that the switch from centralized to offload mode happens only when we are fairly sure that the synchronization overhead is low. Table II shows the datasheet we use in our setup to pick the threshold values. This datasheet was calculated for put operations on 16 to 32 byte key value pairs, and would need to be recomputed for a different setup or key-value pair size.

### E. Enforcing the Offload Mode

When the Cuttlefish adaptation algorithm makes a decision to switch from the offload mode of operation to a centralized mode, or vice versa, the SDN switches in the data plane must be configured in real time to redirect messages to the suitable controller. We now describe how this redirection happens in our system.

Our framework has been implemented over the Open-vSwitch (OVS) [22] SDN switches managed by the Floodlight controller. The OVS switches are configured with rules to identify the various message types specified in the user input. When the system switches modes, the controller and switches must redirect specific offloadable message types to the appropriate controller (root/local) based on the mode of operation. The controller in our implementation did not come with this support to direct packets to a specified controller; all switches forwarded traffic to all configured SDN controllers by default. Therefore, we developed an extension to the Floodlight controller by implementing the `NiciraSetControllerId` feature in the Loxigen library [21], which allows the Floodlight controller to identify and communicate with specific switches. In order to adaptively switch between modes, we also added logic to the controller to automatically generate Openflow commands that

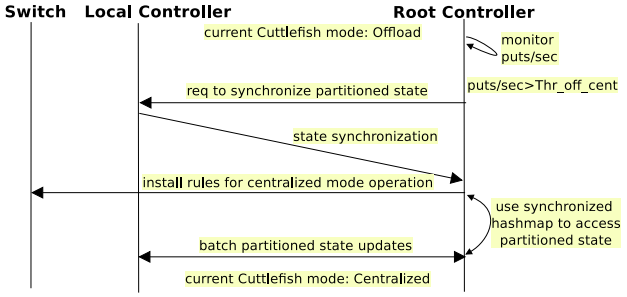



Fig. 6: Switch from Offload mode to Centralized mode.

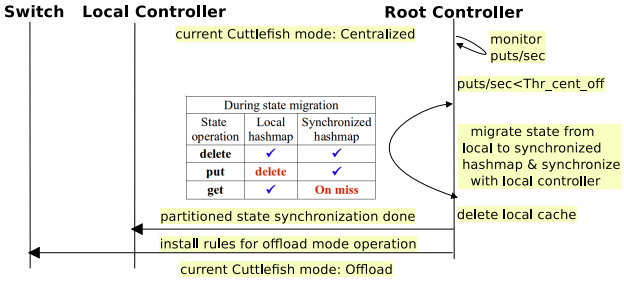



Fig. 7: Switch from Centralized mode to Offload mode.

add/delete/modify rules to direct specific message types to specific controllers at the OVS switches. Finally, we added a new Openflow action type `of_action_nicira` to Floodlight that allows adding routes at switches to direct packets to a specific controller (instead of forwarding to all controllers, as in the default implementation). These changes required modifying ∼150 lines of code in the controller (Java), and Loxigen library (C++) code base and required no changes to the OVS switch implementation.

### *F. Transition between Controller Modes*

When transitioning between modes, the Cuttlefish framework takes care to avoid race conditions between the installation of switch rules to divert traffic, and the process of synchronizing state across the root and local controllers. When the framework switches from a centralized mode of operation to an offload mode (Figure 7), the state from the local hashmaps at the root is migrated to the synchronized hashmaps first, and switch rules to divert the offloadable messages to the local controllers are installed only after the root and local controllers have completed the synchronization of partitioned state. Similarly, when migrating from the offload mode to the centralized mode (Figure 6), the switch rules are installed only after all the batched updates from the local controllers have been flushed to the root. We now describe this mechanism in more detail.

**Offload to Centralized mode:** Recall that the partitioned state is synchronized in batches at local controllers in offload mode. Cuttlefish uses a flag *push_update* at the local controller to indicate that updates must be pushed immediately to the root; this flag is set to false in offload mode. When we want to switch from offload to centralized mode, we must immediately synchronize the partitioned state, so the root controller sets the *push_update* flag at the local controller, causing the synchronized hashmaps to flush all pending updates immediately. After waiting for a grace period for the synchronization to complete, the root controller is ready to switch to centralized mode. The root controller first pushes the rules onto the OVS at the local controller to forward all the messages (offloadable and otherwise) to the root controller. However, there could still be packets in the pipeline at the switch of the local controllers, which could continue to update the partitioned state for a short duration after the switch rules have been installed. In order to correctly handle such packets, the root controller accesses partitioned state from synchronized hashmaps for a brief waiting period. Further, new packets arriving at the root are buffered until the packets in the local switch's pipeline have been processed, in order to avoid reordering. Once this grace period for flushing the switch pipeline has expired, the root stops state synchronization from the local controller by turning the *push_update* flag to false. The root controller can now switch to centralized mode without any state inconsistency issues, and store newly created partitioned state in the local hashmap cache for better application performance. The values of the grace periods are a few milliseconds in our implementation, and will have to be configured based on the network latency between the root and local controllers for other deployments.

**Centralized to Offload mode:** When Cuttlefish is operating in centralized mode, some of the partitioned state is stored in the local hashmap cache at the root controller, and some in the synchronized hashmaps. When the framework decides to switch from centralized to offload mode, we must migrate the partitioned state from the local hashmap cache to the synchronized hashmap at the root controller. At the root controller, the boolean variable *migrating* (if true) indicates that the state is being migrated from local hashmap cache to synchronized hashmap. When we set the *migrating* flag to true, all delete operations at the root are performed on both the local and synchronized hashmaps, all put operations are performed only on the synchronized hashmaps, whereas all get operations are handled normally (get from the local cache, and on a miss get from the synchronized hashmap). Also, for all put operations during state migration, we first perform delete on local hashmap to avoid state inconsistency. After the local hashmap has been transferred to the synchronized hashmaps at the root, the local cache is cleared to avoid stale state, and the *migrating* flag is set to false. We then wait for a grace period for the synchronized hashmap updates to propagate to the switches, after which we push rules on to the switches to forward all offloadable messages to the local controller. Finally, we also enable batching of updates to partitioned state at the local controller in offload mode.

### *G. Implementation of Use cases*

We implement the three sample applications discussed in §II— a key-value store, an SDN-based LTE packet core, and a stateful load balancer—over the Cuttlefish framework, to demonstrate and evaluate the benefits of our framework.

**Key-value store.** We implemented a centralized key value store application that performs put operations to partitioned state at the root controller, and get operations to the same state at the local controller, using the Cuttlefish API. We have also implemented a load generator to generate traffic with varying ratios of put/get requests, in order to test the adaptive offload component of our system. We use the IP ToS field in packet headers to identify put and get requests at the switches. The application and load generator were implemented in about 1400 lines of Java/C++ code.

**SDN based LTE EPC.** We implement the SDN-based LTE EPC application by extending an existing version of the code [23] built atop the Floodlight controller and OVS SDN switches, and adapting it to use the Cuttlefish API. We extended the load generator in the existing code to tag packets with message types in the IP ToS field, in order to enable easy identification of the various control plane messages. We also modified the load generator to generate traffic of varying mixes, e.g., vary the ratio of the attach requests to the service requests. Our changes modified 1800 lines of Java/C++ code in the original application code base.

**Stateful load balancer.** We built a stateful load balancer as an SDN application on top of the Floodlight controller in about 600 lines of Java code. We also wrote a load generator that varies the distribution of load to servers, in order to force updates to the partitioned state of server load statistics.

## IV. Evaluation

We now describe our evaluation of the Cuttlefish framework. Our evaluation aims to answer two important questions:

- What are the performance gains of adaptively offloading computation across local controllers? (§IV-B)
- How efficiently does Cuttlefish accomplish the process of adaptively switching modes? (§IV-C)

### A. Experimental Setup

**Testbed.** We deployed the Cuttlefish applications over our testbed consisting of a Floodlight v1.2 controller as the root, and six OVS v2.3.2 switches as the dataplane switches. A Floodlight local controller was also colocated with the switches. All components (controller and switches) used Ubuntu 14.04, and were hosted over separate LXC containers to ensure isolation. The containers are distributed amongst two 16-core Intel Xeon E312xx @2.6Ghz servers with 64GB RAM. The root and local controllers, and all gateway switches, were allocated 1 CPU core and 4GB RAM each.

**Parameters and metrics.** We generate different experiment scenarios by varying the mix of offloadable and non-offloadable messages in the control plane traffic processed by the SDN controllers. All experiments ran for 300 seconds unless mentioned otherwise. The performance metrics measured in our experiments were the average control plane throughput (number of control plane messages processed/sec) and average response latency of control plane requests. We compare these metrics across three modes of operation of the application: (a) centralized mode, where all control plane messages are handled at the root controller, (b) offload mode, where all offloadable messages are always offloaded to local controllers, and (c) the Cuttlefish adaptive offload mode, where offloadable messages are processed at the local controller only if the Cuttlefish adaptation algorithm decides that the synchronization overhead is low enough.

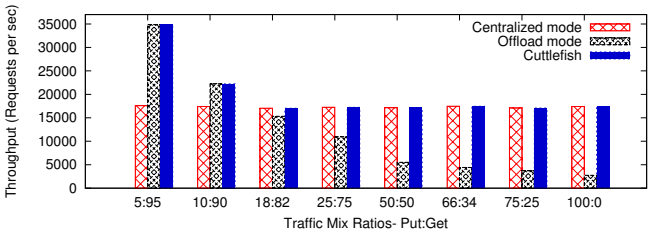



Fig. 8: Key-value store: control plane throughput.

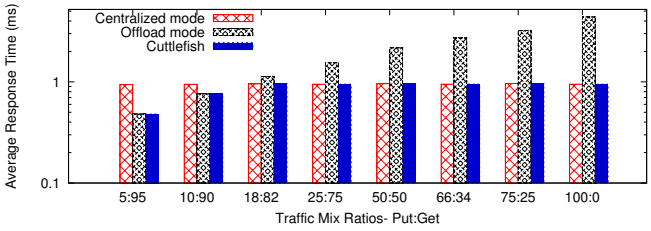



Fig. 9: Key-value store: control plane latency.

### B. Efficacy of Adaptive Offload

We first quantify the performance gains due to the adaptive offload mechanism of Cuttlefish. We vary the mix of get and put requests in the incoming traffic (mix $x : y$ denotes $x\%$ non-offloadable puts, i.e., puts to the partitioned state at the root controller, and $y\%$ offloadable gets, i.e., gets from the partitioned state at the local controller), and measure the performance of the Cuttlefish key-value store application. Figure 8 shows the average throughput of all the controller modes, and Figure 9 shows the average request processing latency. As expected, the performance of the offload mode degrades as compared to the centralized mode, as the proportion of non-offloadable traffic increases. However, across all traffic mixes, we see that the performance of the Cuttlefish adaptive offload mode matches that of the best non-adaptive mode for that traffic mix. We observe that the Cuttlefish throughput is up to 2X higher than that of the traditional centralized mode, and its latency is up to 50% lower. Also, Cuttlefish throughput is up to 6.4X higher than that of the offload mode, and its latency is up to 80% lower. Further, the throughput and latency of Cuttlefish are almost equal to that of the optimal mode (whether centralized or offload) for a given traffic mix, because the cost of running the adaptation module is almost negligible.

Figure 10 and Figure 11 show the control plane throughput and latency respectively of the LTE EPC application, as we vary the traffic mix ($x : y$ denotes $x\%$ non-offloadable attach and detach requests and $y\%$ offloadable service requests). Our observations remain the same for this application as well.

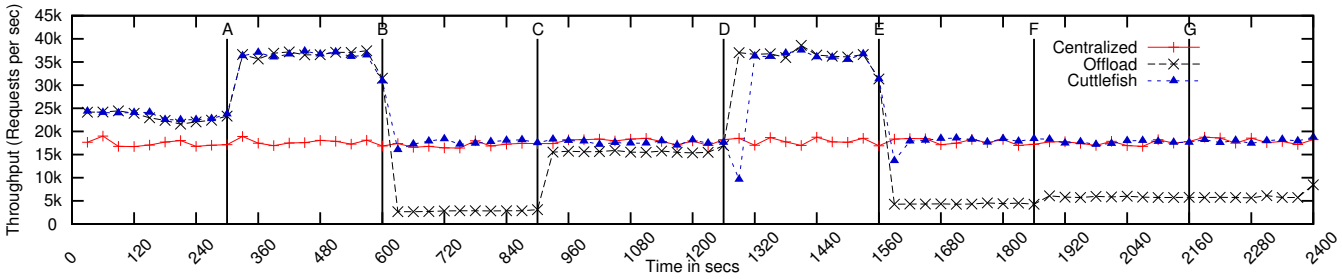



Fig. 12: Throughput with varying traffic mix for the key-value store application.

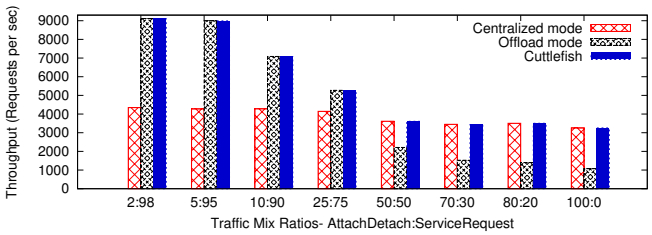



Fig. 10: LTE EPC: control plane throughput.

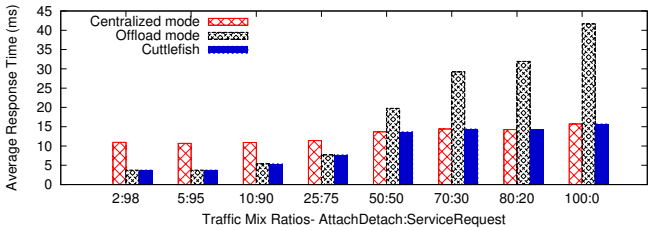



Fig. 11: LTE EPC: control plane latency.

The throughput of Cuttlefish is up to 2X higher than that of the traditional centralized mode, and its latency is up to 66% lower. Cuttlefish throughput is also up to 3X higher than that of the offload mode, and its latency is up to 62% lower. As before, the performance of Cuttlefish matches the best performing mode for a given traffic mix. For the load balancer, we observe that the amount of synchronization traffic is very low, and hence the offload mode is always suitable. But the application still requires to use the Cuttlefish API to synchronize the partitioned state (server load) in order to scale, as well as take flow route decisions faster. We omit presenting those results here.

### C. Convergence of Adaptive Offload

In our next set of experiments, we demonstrate effectiveness of the adaptation mechanism and measure the amount of time taken by Cuttlefish to compute the correct mode of operation and switch to it when the traffic mix changes. We only present results for the key-value store application; the results were qualitatively similar for the other applications.

In this experiment, we generate get/put traffic to the key-value store application for a duration of 2400 seconds, while varying the traffic mix during the experiment as follows. The ratio of put to get requests changes from 10:90 during the first 300s of the experiment, to 5:95 in the next 300s, to 100:0 in the next 300s, to 18:82 in the next 300s, to 5:95 in the next 300s, to 66:34 in the next 300s, and finally to 50:50 in the final 600s. We use the threshold values $Thr_{Off\_Cent}$=2400 puts/sec, and $Thr_{Cent\_Off}$=2200 puts/sec on the average rate of puts to partitioned state for the offload-to-centralized and centralized-to-offload transitions respectively. These values correspond to a synchronization overhead of 80% CPU utilization at the root controller, as seen from Table II. Figure 12 shows the throughput of the key-value store application, sampled every 30 seconds for the duration of the experiment. The corresponding put rate metric that was used to make the offload decision is shown in Figure 13 (samples shown every 40 seconds for visual clarity).

From the graphs, we see that when the traffic consists of predominantly get requests in the first 600s (upto point B in the graphs), Cuttlefish operates in offload mode. After point B, the put rate crosses the threshold, the adaptation algorithm switches to centralized mode, and stays in this mode upto point D. After point D, the non-offloadable traffic reduces, the Cuttlefish adaptation algorithm switches to offload mode, and stays there upto point E. After point E, the traffic mix incurs a high synchronization cost, and Cuttlefish switches to centralized mode, and remains in this mode for the rest of the experiment. Throughout the experiment, we observe that the Cuttlefish adaptation algorithm always correctly identifies the best performing controller mode and correctly switches to it. We observe transient drop in performance after points B,D, and E, due to the mechanisms of migrating between modes in Cuttlefish. We find that the Cuttlefish framework takes around 20–30 seconds to switch to a new mode of operation after a change in traffic mix. This switching duration is obviously a function of the frequency at which we invoke our decision algorithm (every 10 seconds), and on size of the application-centric state requiring synchronization (700 key value pairs in this experiment).

Given that Cuttlefish takes a few tens of seconds to identify and switch between modes, it is expected that Cuttlefish will not perform well if the traffic mix changes very frequently. Of course, Cuttlefish can perform better if it reduces its monitoring interval from 10s to something smaller.

## V. Related Work

Several approaches have been proposed to ameliorate the scalability concern in a logically centralized SDN controller

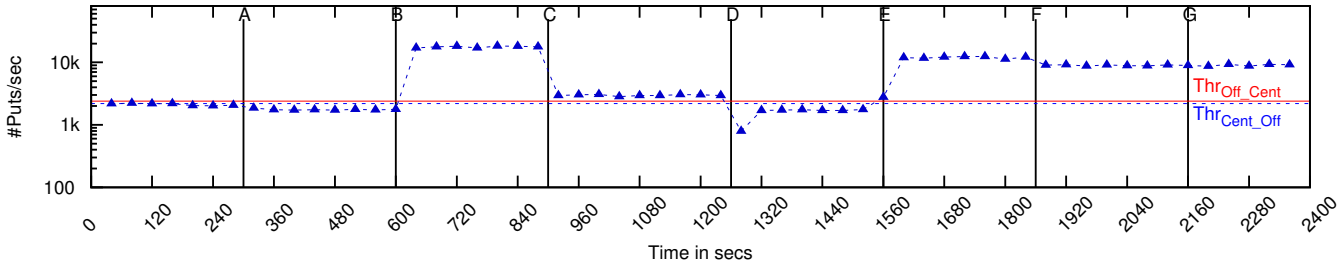


Fig. 13: Synchronization metric for the key-value store application.

design. Horizontally-scalable distributed controllers [1]–[3] scale the centralized SDN controller by instantiating multiple homogeneous instances of the centralized controller, and distributing the control load with techniques like network topology partitioning or state partitioning. On the other hand, prior hierarchical scaling techniques [4]–[6] and our Cuttlefish framework scale controllers by offloading computation from the central root controller. The two design options—distributed controllers and hierarchical controllers—are complementary ideas, with their own strengths and drawbacks. While the former design can offload any control plane computation to any replica (after suitable state synchronization), the latter can offload only a subset of control plane computation that can be correctly performed at local controllers. However, distributed controller frameworks incur a performance overhead due to the synchronization of network-wide state across replicas, while hierarchical controller designs have no such associated costs because local switch-specific state does not require synchronization.

**Horizontally-scalable distributed controllers.** Onix [1] provides a control plane API for programmers to implement distributed network applications, without worrying about state distribution, element discovery, and failure recovery mechanisms. Hyperflow [2] provides SDN control application scalability by use of multiple physical controllers, but with a logically centralized view. Hyperflow passively synchronizes network-wide view of controllers using the publish/subscribe event based system, without any changes to the SDN application. The work that comes closest to Cuttlefish is the distributed controller design of Beehive [3]. Application state in Beehive is stored as key-value pairs in a shared distributed data store, much like how Cuttlefish stores partitioned state in synchronized hashmaps. These key-value pairs of an application can be placed at any of the distributed controllers, and the Beehive controllers must run an expensive synchronization protocol to agree on the location of every piece of distributed state. While this approach has good fault tolerance, it also incurs a high synchronization overhead. In contrast, Cuttlefish distributes only a subset of application state (partitioned state) to local controllers. Therefore, Cuttlefish can offload only a subset of control plane messages that depend on such state to local controllers, while Beehive requires locating application data and offloading of computation. Further, Cuttlefish does not explicitly handle the failure of local controllers and the resulting loss of state, while Beehive ensures fault tolerance via replication of state at the distributed controllers.

However, as compared to Beehive and other distributed controller frameworks, our approach trades off generality and robustness in favor of performance. We compare the reported performance of a key-value store application in Beehive with our own key-value store application. While Beehive can take up to 20ms to identify a remote node and perform a put operation, Cuttlefish takes under 5ms for a similar request. The difference in performance comes from the fact that Beehive applications must query a globally synchronized index to determine the location of state required for a certain computation, while Cuttlefish routes messages to controllers by identifying message types from packet headers at the dataplane switches itself.

In summary, we see Cuttlefish and distributed controller frameworks like Beehive as complementary techniques that represent two very different design points in the space of scalable SDN controller frameworks. We can also envision both techniques being applied together—SDN applications can offload whatever state and computation they can easily offload to local controllers, and use a fault-tolerant distributed controller framework to scale the non-offloadable computation at the root controller.

**Hierarchical controller frameworks.** Prior work (e.g., [4]–[6]) has considered offloading computations that rely on local state to local controllers. Kandoo [5] offloads tasks like gathering flow statistics and detecting elephant flows, while FOCUS [6] offloads node discovery via ARP flooding. Most of these optimizations are related to computation tasks that do not affect global controller state. In contrast, Cuttlefish not only offloads local computation, but also computation that depends on partitioned global state that need to be only occasionally synchronized with its copy at the root controller.

Our prior work Devolve-redeem [24] proposed a design of a hierarchical SDN framework that offloads partial global state and its related computation to local controllers, and is a precursor to our present work. However, this earlier work did not contain a mechanism for adaptively switching between the centralized and offload modes, and required a more complex input from the user. Our present work significantly improves upon this earlier work.

**Offload of rules.** Difane [25] caches pre-computed forwarding rules across a subset of local switches, to avoid expensive

communication with the controller when new flows arrive. Eden [26] provides a framework for implementing the network functions that do not require high network support at the end hosts. Eden tags packets with the message type in an application library, and processes them at end hosts via a set of match-action tables and a runtime. Cuttlefish also works by identifying application message types. However, Cuttlefish offloads the entire application logic (not just the match-action rules) to local controllers on switches, and hence is more powerful than these approaches.

**State distribution frameworks.** The techniques used in Cuttlefish to manage distributed state across root and local controllers are similar to ideas used in frameworks to manage distributed state in networking applications [19], [20], [27], [28]. Split/merge [19] provides a state management API to applications for managing scale-up and scale-down operations. State is transparently split between middlebox replicas for scale-up, and merged to one replica for scale-down. OpenNF [20] improves split/merge by providing options for loss-free, and ordered state updates between middlebox replicas. On the other hand, the goal of Pico replication [27] is to provide a low overhead, high availability framework for middleboxes. In order to dynamically grow or shrink the number of SDN controllers, Elasticon [28] proposes a switch migration protocol, and enables load shifting between controllers. Some ideas of Cuttlefish, including the state management API and the protocol to guarantee ordered message delivery when migrating between controller modes, have been inspired by this body of literature.

## VI. Conclusion

This paper presented the design and implementation of Cuttlefish, a hierarchical SDN controller that offloads a subset of SDN application state and computation to local controllers on switches, in order to scale SDN control plane capacity. In addition to switch-local state, we identified a subset of global application state, called partitioned state, that can be correctly cached and updated at local controllers on switches. Cuttlefish incorporated an adaptive state offload capability to balance the tradeoff between performance gains due to offload of partitioned state, and the cost of synchronizing this state across the root and local controllers. We developed three sample applications—the SDN-based LTE packet core, a key-value store, and a simple load balancer—and demonstrated efficacy of the Cuttlefish framework. Our evaluation of the sample SDN applications demonstrated that Cuttlefish improved control plane throughput by $\sim$2X and control plane latency by $\sim$50% as compared to the traditional SDN design, and correctly chooses the amount of offload to optimize application performance. Our framework, based on the popular Floodlight SDN controller, is available for use by SDN application developers [17].

## References


[1] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proc. of OSDI 2010*.

[2] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proc. of the INM/WREN 2010*.

[3] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple distributed programming in software-defined networks," in *Proc. of SoSR 2016*.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proc. of ACM SIGCOMM 2011*.

[5] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. of HotSDN 2012*.

[6] J. Yang, Z. Zhou, T. Benson, X. Yang, X. Wu, and C. Hu, "Focus: Function offloading from a controller to utilize switch power," in *Proc. of IEEE NFV-SDN 2016*.

[7] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, 2013.

[8] A. Basta *et al.*, "A virtual sdn-enabled lte epc architecture: A case study for s-/p-gateways functions," in *Proc. of IEEE SDN4FNS 2013*.

[9] M. R. Sama, L. M. Contreras, J. Kaippallimalil, I. Akiyoshi, H. Qian, and H. Ni, "Software-defined control of the virtualized mobile packet core," *IEEE Communications Magazine*, vol. 53, 2015.

[10] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann, "Applying nfv and sdn to lte mobile core gateways, the functions placement problem," in *Proc. of All Things Cellular 2014*.

[11] L. Fang, F. Chiussi, D. Bansal, V. Gill, T. Lin, J. Cox, and G. Ratterree, "Hierarchical sdn for the hyper-scale, hyper-elastic data center and cloud," in *Proc. of SoSR 2015*.

[12] Z. A. Qazi, P. K. Penumarthi, V. Sekar, V. Gopalakrishnan, K. Joshi, and S. R. Das, "Klein: A minimally disruptive design for an elastic cellular core," in *Proc. of SoSR 2016*.

[13] M. Moradi, W. Wu, L. E. Li, and Z. M. Mao, "Softmow: Recursive and reconfigurable cellular wan architecture," in *Proc. of CoNEXT 2014*.

[14] "The evolved packet core," http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core, 2017.

[15] D. Wonak, "A storm is brewing an lte signaling storm," http://www.diametriq.com/wp-content/uploads/downloads/2013/01/A-Storm-is-Brewing.pdf, 2012.

[16] w. p. Nokia Siemens Networks, "Signaling is growing 50% faster than data traffic."

[17] R. Shah, M. Vutukuru, and P. Kulkarni, "Cuttlefish project," https://github.com/networkedsystemsIITB/cuttlefish, 2018.

[18] "Programming Protocol-independent Packet Processors," https://p4.org/, 2014.

[19] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes." in *Proc. of NSDI 2013*.

[20] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, 2014.

[21] T. Ribeiro, "Floodlight," https://github.com/floodlight, 2016.

[22] OVS, "OpenvSwitch," https://github.com/openvswitch, 2011.

[23] A. Jain, S. Lohani, and M. Vutukuru, "Opensource SDN LTE EPC," https://github.com/networkedsystemsIITB/SDN_LTE_EPC, 2016.

[24] R. Shah, M. Vutukuru, and P. Kulkarni, "Devolve-redeem: Hierarchical sdn controllers with adaptive offloading," in *Proc. of ACM APNet 2017.*

[25] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *Proc. of ACM SIGCOMM 2010.*

[26] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling end-host network functions," in *Proc. of ACM SIGCOMM 2015*.

[27] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. of SoCC 2013*.

[28] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "Elasticon; an elastic distributed sdn controller," in *Proc. of ANCS 2014*.