Gallium: Automated Software Middlebox Offloading to Programmable Switches

Kaiyuan Zhang
University of Washington
kaiyuanz@cs.washington.edu

Danyang Zhuo
Duke University
danyang@cs.duke.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

Abstract

Researchers have shown that offloading software middleboxes (e.g., NAT, firewall, load balancer) to programmable switches can yield orders-of-magnitude performance gains. However, it requires manually selecting the middle-box components to offload and rewriting the offloaded code in P4, a domain-specific language for programmable switches. We design and implement Gallium, a compiler that transforms an input software middlebox into two parts—a P4 program that runs on a programmable switch and an x86 non-offloaded program that runs on a regular middlebox server. Gallium ensures that (1) the combined effect of the P4 program and the non-offloaded program is functionally equivalent to the input middlebox program, (2) the P4 program respects the resource constraints in the programmable switch, and (3) the run-to-completion semantics are met under concurrent execution. Our evaluations show that Gallium saves 21-79% of processing cycles and reduces latency by about 31% across various software middleboxes.

CCS Concepts

• **Networks → Programmable networks**; **In-network processing**;

Keywords

Protocol offload, Middleboxes

ACM Reference Format:
Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3387514.3405875

1 Introduction

Recent advancement in programmable switches has enabled a new approach to achieve higher middlebox performance: offloading packet processing functions into programmable switches. For example, Silkroad [20] proposes to leverage programmable switches to build high-performance load balancers that can handle substantially more concurrent connections. New use cases, such as using programmable switches for in-network DDoS detection [2], are also emerging.

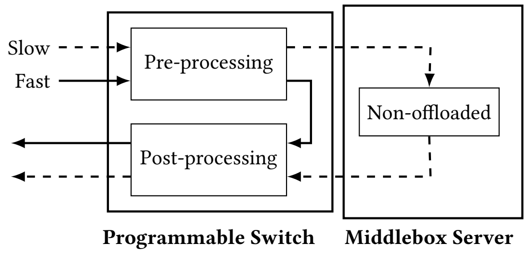


Figure 1: Packet flow in an offloaded middlebox.

To offload a software middlebox to a programmable switch, developers must consider several issues. For example, what functions to offload, how to run offloaded functions on programmable switches, how to execute the non-offloaded functions, and how to ensure that the programmable switch doesn't run out of resources? Understanding these issues requires developers to become experts on programmable switches, slowing down development.

In this paper, we ask a simple question: *given a software middlebox written in a general-purpose programming language (C++), is it possible to automatically produce a functionally equivalent middlebox that leverages a programmable switch for higher performance?* To make our question concrete, we focus on a single type of offloading scenario shown in Figure 1 that uses a programmable P4 switch (e.g., a Tofino switch [3]) and a regular middlebox server.

The goal is to automatically partition the input middlebox program into three parts, i.e., a pre-processing partition, a non-offloaded partition, and a post-processing partition. The pre-processing and post-processing partitions run on the programmable switch as a single P4 program, and the non-offloaded partition executes on the middlebox server. Packets coming to the middlebox go through the pre-processing, non-offloaded, and post-processing partitions sequentially. Because a programmable switch is efficient at packet processing, this offloading scenario can improve middlebox performance if enough instructions are offloaded to the switch. Further, if the non-offloaded partition is not involved in processing a packet, the packet can simply skip the middlebox server, reducing latency and processing cycles in the server. We call this a fast path in the middlebox.

We have designed and implemented Gallium, a compiler that automatically partitions and compiles a software middlebox written in C++ to a functionally equivalent middlebox that leverages a programmable switch to achieve higher performance. Gallium has to address three main challenges: (1) ensuring the output code is functionally equivalent to the input middlebox, (2) enforcing expressiveness constraints and resource limits on the pre- and

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7955-7/20/08...$15.00
https://doi.org/10.1145/3387514.3405875

post-processing partitions, and (3) providing the run-to-completion concurrency model even though packet processing is split across a software middlebox and a hardware switch.

Gallium partitions the input middlebox source code at a fine granularity (e.g., P4 operations and CPU instructions). To ensure functional equivalence, Gallium keeps track of each instruction's dependencies, i.e., what instructions need to run before a given instruction. For an instruction, I, to behave in the same way for the unpartitioned and partitioned versions, we need to make sure the state accessed by I is the same across the two versions. Therefore, we need to execute all the instructions that may affect I's observable state before we run I. Gallium provides a simple approach to extract all the instruction-level dependencies by comparing each instruction's read and write sets (i.e., the collection of variables an instruction accesses or modifies).

Because programmable switches only accept P4 programs—a declaration of a pipeline of match-action tables with limited ALU functions (e.g., addition, subtraction, comparison), Gallium statically analyzes the input middlebox to figure out the set of instructions to assign to the pre- and post-processing partitions. Gallium must also consider the resource constraints inside the programmable switch, e.g., the total amount of memory and the scratchpad state used to process a packet. We design and implement a partitioning algorithm that maximizes the number of offloaded operations while meeting the constraints on the programmable switch.

Finally, Gallium needs to generate deployable output middleboxes and execute them with the desired run-to-completion semantics. Gallium maps data structures and instructions in a general-purpose programming language (C++) to their P4 counterparts. Gallium synthesizes the packet formats to allow middlebox states to flow between the switch and the server. Gallium further employs state synchronization techniques typically found in primary-backup systems to provide the desired run-to-completion concurrency semantics.

We evaluate Gallium on five different middleboxes: MazuNAT, an L4 load balancer, a firewall, a network proxy, and a trojan detector, written using the Click [21] framework in C++ and totaling over 6K lines of code. Our evaluations show that Gallium is able to automatically offload middlebox functionality, save up to 79% processing cycles, reduce latency by up to 31%, and improve throughput by up to 46%.

2 Background

The advent of programmable switches has enabled network programmability at line-rate performance. We first describe the operational model of programmable switches and then explain the restrictions imposed on packet processing.

2.1 Programmable Switches

Programmable switches [7] are packet processing devices that allow programmability at line rate. We assume an abstract switch model, as described in [6], that provides the following functionality.

Match-action tables: The internal architecture of a programmable switch is a chain of physical match-action tables. Each match-action table can match on specific fields of the packet header and trigger a set of actions, including packet header rewriting, dropping a packet, or delivering a packet to the next table. The match operation can check for exact matches, wildcards, and longest prefix matches. As long as each packet goes through the chain only once, packet processing is at line rate.

Registers or stateful memory: A limited amount of memory can maintain state across packets, such as counters, meters, and registers. This state can be both read and updated during packet processing.

Computation primitives: The switch can perform a limited amount of processing on header fields and registers (e.g., additions, bit-shifts, and hashing).

Programmable switches support the P4 [6] programming model. A P4 program specifies the header format, the pipeline sequence, including what packet header fields each table matches on, what actions each table takes, and how the tables are interlinked. The contents of the tables are read-only for the data plane. To modify the contents of a table, the switch's CPU (x86) can issue commands to the switching silicon. These commands issued through the control plane can be significantly slower than packet processing.

2.2 Restrictions

Programmable switches support only a small set of ALU operations on limited data types. Currently, they support integers only but not floating-point numbers. The list of supported operations is limited to integer addition, subtraction, bitwise operations (i.e., AND, OR, XOR, NOT, SHIFT), and comparison. Further, a programmable switch also limits the number of sequential processing steps to the number of match-action pipeline stages (generally around 10 to 20), so the amount of computation that can be offloaded is bounded. Within a pipeline stage, rules are processed in parallel.

A key restriction is the memory limit on the programmable switch. The total amount of memory is a few tens of MBs on today's switches. Another memory constraint that requires attention is the scratchpad memory allowed for per-packet processing. A programmable switch allows some metadata to be stored in a scratchpad memory while processing a packet. This metadata is allocated when a packet arrives and is garbage-collected when the packet leaves the switch. The total amount of scratchpad memory for storing packet-level metadata is less than a few hundred bytes for processing a single packet.

Programmable switches also restrict the control flow. P4 programs cannot have loops, because the hardware realization is a sequence of physical tables, and each packet goes through the sequence only once. Another implication is that if a packet does a lookup on a particular table, the packet can no longer access the same table in a later pipeline stage.

Programmable switches can read and write packet contents that are only at the beginning of the packet (typically, the first 200 bytes of a packet). This restriction means that developers must carefully design the packet format if they want to transfer additional data using packet headers.

3 Gallium Overview

Gallium is a compiler that transforms an input middlebox program written in a general-purpose programming language (C++) using Click APIs [21] into a functionally equivalent middlebox implementation that has two parts: (1) a P4 program that runs on

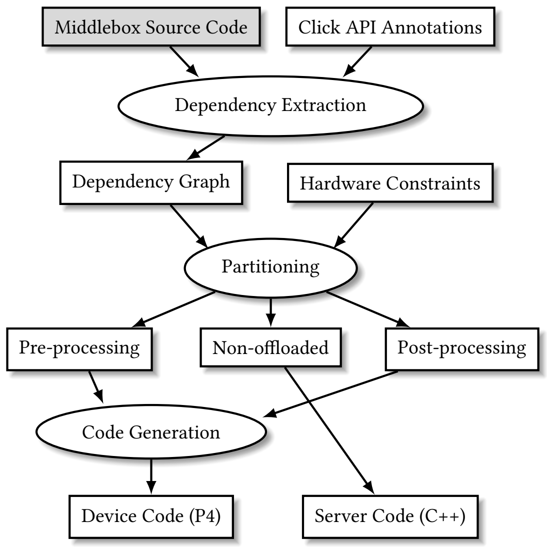


Figure 2: Workflow of Gallium. The shaded box is the input program written by middlebox programmers. Other rectangular boxes are annotation and configuration inputs given to Gallium, intermediate representations, and final outputs.

a programmable switch, and (2) a C++ program that runs on a traditional middlebox server.

3.1 Goals

Gallium has to realize three goals with the generated code.

- *Functional equivalence:* the combined effect of the two parts (i.e., the P4 program and the C++ code) should be functionally equivalent to the input middlebox program.
- *Conform to constraints:* the generated P4 program should satisfy the expressiveness and resource constraints of programmable switches.
- *Concurrency-safe execution:* the output middlebox must provide per-packet run-to-completion semantics even though the code on the switch executes concurrently with the server program.

We now elaborate on the third goal. Runtime systems for middlebox dataplanes typically guarantee the run-to-completion semantics for packets, wherein all protocol processing for a packet is done before moving on to the next packet. In our setting, where we are concurrently executing portions of the middlebox logic on the switch and the server, we run the risk of a later packet observing none or only a subset of the state updates made by a previous packet.

We, therefore, formalize the desired *run-to-completion* semantics. If a packet p_j is received by the middlebox[1] after p_i and if p_j is causally dependent on p_i's receipt (e.g., p_i is a SYN packet and p_j is the corresponding SYN-ACK packet), then we require p_j to observe all of the state updates made by p_i. If p_j is not causally dependent on p_i's receipt (i.e., there is no end-to-end constraint that the middlebox has processed p_i before p_j), then p_j should observe either all or none of the state updates made by p_i. Note that, if the later packet p_j observes none of the state updates, then the resulting execution is considered to be equivalent to the setting where p_i was delayed in the network and received after p_j.

[1]In the context of Gallium, a middlebox refers to the aggregated entity comprising of a server and a switch.

The correctness criteria described above embody both a condition on handling causal dependencies (i.e., packets observe the state updates of other packets that causally preceded them) and an atomicity requirement (i.e., packets either observe all or none of the state updates of previous packets). It is worth noting that this is similar to the correctness requirements imposed on multi-threaded middlebox programs, wherein in-flight packets are processed atomically by the middlebox in any order.

3.2 Compilation Process and Execution

We outline below how Gallium is designed to meet the desired goals. We overview the partitioning and compilation process and then describe how the runtime system ensures the correct execution of the generated code. Figure 2 shows the workflow of Gallium.

To ensure functional-equivalence, we need to keep the instruction-level dependencies equivalent between the source program and the generated program. (See §4.1.) Let us consider a code statement S in the source program, and let Gallium assign this statement to one of two partitions (i.e., server code or switch program). We need to ensure that S has the same effect in the generated middlebox as in the input middlebox. This requirement means that, before S runs, the program state that S can access should be the same in the input and the generated program. To capture this notion, we define a "depends on" relation that combines both data and control dependencies that exist in the program. A statement S_2 depends on S_1, if and only if the observable variables in S_2 are affected by S_1's execution. Any time two statements access the same state, with at least one of them being a write, or if a statement determines whether or not some other statement will be executed, we will tag a dependency between the two statements. In the generated program, we ensure that, for any statement S, all statements that S depends on have been executed before S.

The second step is to partition the input program's source code into the desired three partitions (i.e., a pre-processing partition, a non-offloaded partition, and a post-processing partition) while maintaining the dependencies. (See §4.2.) At the same time, Gallium must ensure that the generated P4 program, which contains the pre- and post-processing partitions, conforms to the expressiveness and the hardware resource constraints.

We develop a label-removing algorithm that partitions the source program and preserves dependencies while meeting the switch constraints. Our algorithm first finds a partitioning that maximizes the number of statements offloaded to the programmable switch by only considering the restrictions imposed by the program dependencies and P4's expressiveness constraints. Our algorithm then refines the partitioning result by gradually moving statements from the pre- and post-processing partition to the non-offloaded partition until it satisfies all the resource constraints.

The final step is to generate code that provides the desired concurrency semantics. (See §4.3.) The partitioning process assumes that the middlebox state is synchronously replicated across the switch and the server and that the packets are processed one at a time. The code generation step, along with the runtime system, ensures that the generated program provides a per-packet run-to-completion semantics even as the switch and the server concurrently consume packets. Crucially, Gallium identifies what program state has to be replicated and includes distinct mechanisms for handling replicated and non-replicated state in order to provide run-to-completion semantics.

For state replicated across the programmable switch and the middlebox server, the concurrent processing of packets should exhibit behaviors that would have been obtained if the packets had been processed sequentially. As a concrete example, consider a NAT that maintains a bidirectional address mapping using two connection tables: one which maps an internal address and port to an externally visible port, and the other which maps the externally visible port to an internal address and port. When the NAT receives a SYN packet from a new TCP connection, it updates both of these connection tables to handle subsequent packets from both sides properly. If this computation is performed on the server, then the updates would have to be consistently and atomically replicated on the switch. Therefore, Gallium provides efficient mechanisms for state synchronization.

Gallium also provides mechanisms to communicate non-replicated state between the server and the switch. Because the non-offloaded partition may require additional information from the pre-processing partition (e.g., a temporary variable computed by the pre-processing partition), Gallium has to synthesize a packet format where the additional information can be delivered using the packet header. A similar mechanism also has to exist for delivering information from the non-offloaded partition to the post-processing partition. Gallium uses static analysis to identify the set of variables that need to be transferred and allocates packet header fields to store these variables. Finally, Gallium maps all the data structures and instructions to their P4 counterparts, e.g., from a HashMap lookup to a P4 table lookup. Gallium requires a middlebox developer to annotate the maximum size for each data structure stored in the programmable switch.

At the end of the compilation, Gallium outputs: (1) a deployable P4 program that contains both the pre- and post-processing partitions; and (2) a C++ server program that corresponds to the non-offloaded partition.

4 Design

This section describes the details of dependency extraction, partitioning, code generation, and runtime execution. We use a simple load balancer, `MiniLB`, as a running example. `MiniLB` uses consistent hashing over the source and destination IP addresses to assign incoming TCP connections to a list of server backends. `MiniLB` steers packets by rewriting the destination IP address of the packet. To ensure that packets in a given connection are sent to the same backend server even when the list of backends changes, `MiniLB` stores the mapping from existing connections to backends and steers packets using this mapping. For simplicity, `MiniLB` does not garbage collect completed connections. `MiniLB` contains a single Click element, and its C++ source code is shown below.

```
class MiniLB {
  HashMap<uint16_t, uint32_t> map;
  Vector<uint32_t> backends;
  void process(Packet *pkt) {
    iphdr *ip = pkt->network_header();
    uint32_t hash32 = ip->saddr ^ ip->daddr;
    uint16_t key = (uint16_t)(hash32 & 0xFFFF);
    uint32_t *bk_addr = map.find(&key);
    if (bk_addr != NULL) {
      ip_hdr->daddr = *bk_addr;
      pkt->send();
    } else {
      uint32_t idx = hash32 % backends.size();
      uint32_t bk_addr = backends[idx];
      ip_hdr->daddr = bk_addr;
      map.insert(&key, &bk_addr);
      pkt->send();
    }
  }
};
```

4.1 Dependency Extraction

The first step is to extract the statement-level dependencies in the source program. When we create the partitions (pre-processing, non-offloaded code, post-processing), we want to move as many statements as possible to the pre-processing and post-processing partitions to maximize offloading. The statement-level dependencies determine whether it is possible to move a particular statement to other locations in the source program that are conducive to offloading.

We define the dependency relation "S_2 depends on S_1" to represent the constraint that "S_2" must run after "S_1". This dependency could exist due to one of many reasons, e.g., both statements write to the same memory location. To formally define the dependency condition, we first define a "can happen after" relation. "S_2 can happen after S_1" means that, for all possible program executions of the input program, there is at least one execution trace where S_2 is performed after S_1. This "can happen after" relation denotes a possible dependency of S_2 on S_1. On the contrary, if S_2 cannot happen after S_1, it is impossible for S_2 to depend on S_1.

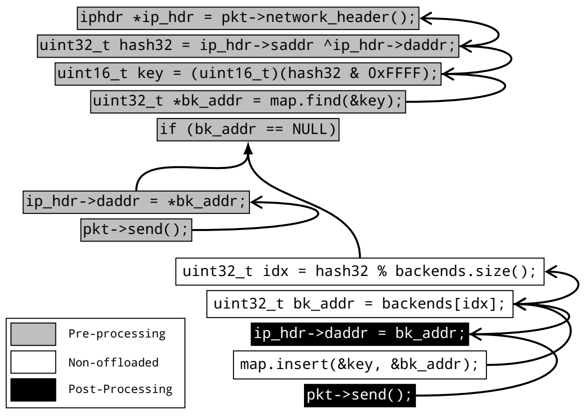
Extracting "can happen after" relations is straightforward. Gallium builds a control-flow graph of the source program. Whether S_2 can happen after S_1 is simply whether S_2 is reachable from S_1 in the control-flow graph.

After we have extracted all the "can happen after" relations, we need to pick the real dependencies inside this set. There are three types of dependencies that we consider, as in a program dependence graph [10].

- Data Dependency: S_1 modifies the state that S_2 reads from or writes to (i.e., read after write and write after write).
- Reverse Data Dependency: S_1 reads some variable or state modified by S_2 (i.e., write after read).
- Control Dependency: S_1 modifies a condition variable used to determine whether S_2 should be executed.

Note that, for programs with loops, S "can happen after" itself and, thus, can also "depend" on itself.

When performing the dependency analysis, Gallium needs to understand what variables and data structures a program statement might access. Specifically, Gallium has to identify a read set, including all the locations a statement may read, and a write set, including all the locations a statement may modify. For simple operations, the source code itself contains the information. For operations invoked through abstract data structure APIs, we need to know the state accessed for each API invocation. This knowledge comes from annotations on the APIs.



iphdr *ip_hdr = pkt->network_header();
uint32_t hash32 = ip_hdr->saddr ^ip_hdr->daddr;
uint16_t key = (uint16_t)(hash32 & 0xFFFF);
uint32_t *bk_addr = map.find(&key);
if (bk_addr == NULL)
ip_hdr->daddr = *bk_addr;
pkt->send();
uint32_t idx = hash32 % backends.size();
uint32_t bk_addr = backends[idx];
ip_hdr->daddr = bk_addr;
map.insert(&key, &bk_addr);
pkt->send();
Pre-processing
Non-offloaded
Post-Processing

Figure 3: The dependency graph of `MiniLB` with partitions.

In Gallium, we require annotations for both data structure APIs (such as `HashMap` and `Vector`) and APIs used to access packet headers. In particular, we need two types of annotations for the Click APIs: (a) the data read and modified when calling into the API and (b) if the API returns a pointer, the data referred to by the pointer.

In `MiniLB`, we have the following annotations:

- The methods `network_header()` and `transport_header()` return pointers to the IP and TCP headers of the packet, respectively. Further, a read/write using the returned pointers is also a read-/write of the corresponding headers.
- The method `HashMap::find()` performs reads on both the input parameter (e.g., `key` pointer) and the `HashMap` data structure (e.g., `map`).
- The `[]` operator of `Vector` class reads the parameter (e.g., `idx`) and the `Vector` (e.g., `backends`).
- The method `HashMap::insert()` reads the two input parameters (e.g., `key` and `bk_addr`) and modifies the HashMap (e.g., `map`).

With these annotations, Gallium constructs the read and write sets for each statement. For statements without method calls, Gallium directly constructs the read and write sets because the variables are explicit. For a Click API invocation, Gallium uses the corresponding annotation to build the read and write set. When the source program uses a pointer dereference, Gallium performs pointer analysis to determine the variable referred to by the pointer. For example, in `MiniLB`, when dereferencing the pointer `ip_hdr`, Gallium traces the origin of the pointer and uses the annotation of `network_header()` API to determine that this is an access to the packet's IP header. Gallium inlines all other function calls before constructing the read and write sets.

Finally, Gallium builds a directed dependency graph from the per-instruction read and write sets. Vertices in the graph are statements in the program, and edges denote the dependencies between statements. For each pair S_1 and S_2, Gallium creates an edge from S_1 to S_2 if S_2 depends on S_1 by checking whether one of the three dependency conditions hold. Figure 3 is the extracted dependency graph for `MiniLB`.

In our examples, we represent the dependency graph using statements in C++. Our implementation, however, creates the dependency graph on LLVM Intermediate Representation (IR) of the source code because LLVM's syntax is simpler than C++. We also ensure that a statement in the LLVM IR can be mapped to a corresponding switch pipeline statement if the programmable switch supports the execution of the operations performed in the statement. This step is relatively straightforward, given that our switch target supports only a limited number of operations, all of which are available as primitives in the LLVM IR. As we expand Gallium to target other, more expressive execution platforms, we might need more flexible intermediate representations.

4.2 Partitioning

Given the dependency information, Gallium then partitions the middlebox program into three pieces, pre-processing, non-offloaded, and post-processing segments, each corresponding to a packet processing step.

There are two problems to solve in partitioning the source program: (1) we need to consider the expressiveness of P4, and (2) we need to consider resource constraints. The latter includes issues such as how much memory is consumed by the pre- and post-processing partitions, how much packet header space we need to transfer information between programmable switches and servers, and can pre- and post-processing partitions fit onto the limited number of processing pipeline stages in the programmable switch.

We choose to deal with these two problems separately. The first step is to determine how to partition the program if we have an unbounded amount of resources (e.g., switch memory, switch processing pipelines, and packet header space). In this step, we only consider P4's expressiveness and the source program's dependencies. The result of the first step is three partitions where we put as many statements as possible in the pre- and post-processing partitions, while maintaining functional equivalence to the source program. For the next step, we refine the partition by carefully moving statements from the pre- and post-processing partitions to the non-offloaded partition so that the final partitions meet the resource constraints.

4.2.1 Assigning Execution Labels to Statements

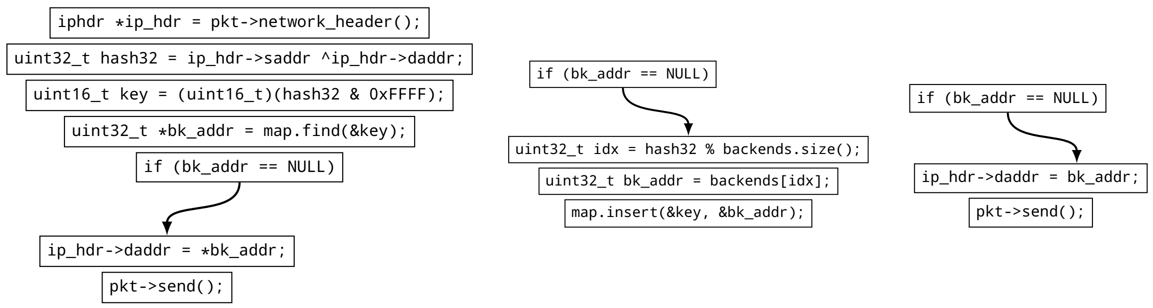
We use a label-removing algorithm to solve the first problem. The basic idea is to assign a set of labels (*pre*, *non_off*, *post*) to each statement to denote whether a statement can belong to a specific partition. We begin with each statement having all possible labels and then gradually remove labels based on a set of label-removing rules that use the computed dependencies.

We define $L(S)$ to be the set of labels for S. We initialize labels in the following way:

$$L(S) = \begin{cases} \{pre, post, non\_off\} & \text{if } S \text{ is supported by P4} \\ \{non\_off\} & \text{otherwise} \end{cases}$$

This initialization means that any statement not supported by P4 should be in the non-offloaded partition. A statement S is supported by P4 if and only if these conditions hold:

1. S involves only those operations that P4 supports, e.g., integer addition, subtraction, and comparison.
2. S's access of the packet, if any, is only to the packet header fields and not packet payloads.



```
iphdr *ip_hdr = pkt->network_header();
uint32_t hash32 = ip_hdr->saddr ^ip_hdr->daddr;
uint16_t key = (uint16_t)(hash32 & 0xFFFF);
uint32_t *bk_addr = map.find(&key);
if (bk_addr == NULL)
ip_hdr->daddr = *bk_addr;
pkt->send();
```

(a) pre-proccessing

```
if (bk_addr == NULL)
uint32_t idx = hash32 % backends.size();
uint32_t bk_addr = backends[idx];
map.insert(&key, &bk_addr);
```

(b) non-offloaded

```
if (bk_addr == NULL)
ip_hdr->daddr = bk_addr;
pkt->send();
```

(c) post-processing

Figure 4: Control-flow graphs for pre-processing, non-offloaded, and post-processing partitions of `MiniLB`.

(3) S is a Click API invocation with a P4 implementation, e.g., a hash table lookup that can be replaced with a P4 table lookup.

Gallium applies a set of label-removing rules over all the statements repetitively. The updating rules are given as constraints between the labels of statement S and all its neighboring statements in the statement dependency graph. Here, we use $S_1 \rightsquigarrow S_2$ to denote the fact that "S_2 depends on S_1", and $\rightsquigarrow^*$ to denote the transitive closure of $\rightsquigarrow$.

Gallium uses the following label-removing rules during the labeling process:

$$\begin{aligned}
&(1) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge \mathit{post} \notin L(S)) \implies \mathit{post} \notin L(S') \\
&(2) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge \mathit{pre} \notin L(S')) \implies \mathit{pre} \notin L(S) \\
&(3) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge S, S' \textit{ access same global state} \\
&\qquad \wedge\, \mathit{pre} \in L(S')) \implies \mathit{pre} \notin L(S) \\
&(4) \quad \forall S, S', (S' \rightsquigarrow^* S \wedge S, S' \textit{ access same global state} \\
&\qquad \wedge\, \mathit{post} \in L(S)) \implies \mathit{post} \notin L(S') \\
&(5) \quad \forall S, S \rightsquigarrow^* S \implies L(S) = \{\mathit{non\_off}\}
\end{aligned}$$

The first two rules ensure that the partitions are consistent with the dependencies. The third and fourth rules are required due to the P4 language's limitation that a global state can only be accessed once inside the switch pipeline. In particular, if there is some global state accessed in multiple different program locations, then at most one of the statements can be executed on the switch.[2] The last rule prevents the offloading of statements that appear in a loop body as P4 does not support loops. Gallium repeatedly applies these rules to eliminate labels until no label can be further removed. This algorithm always converges as the total number of labels monotonically decreases and terminates when the constraints are satisfied for every pair of statements in the dependency graph.

4.2.2 Satisfying Resource Constraints The labels assigned to each statement indicate the partitions a statement can be assigned to, given the dependencies but without considering the resource constraints. We consider several types of resource constraints. We notice that there are two types of state a middlebox needs to store: per-packet state and global state. The per-packet state is pieces of information whose lifetimes only last during the processing of a single packet, such as the `hash32` variable in `MiniLB`. The global state has to be maintained across all packets, such as the `map` variable in `MiniLB`.

We need to enforce the following resource constraints:

- *Constraint 1:* The total size of the global state maintained by the switch does not exceed the size of the switch memory.
- *Constraint 2:* The length of the longest dependency chain in the offloaded code cannot exceed the switch's pipeline depth.[3]
- *Constraint 3:* Each element of the global state maintained on the switch can only be accessed once during packet processing.
- *Constraint 4:* The total size of the per-packet state does not exceed the maximum allowed per-packet metadata.
- *Constraint 5:* The additional per-packet information transferred between the server and the switch is bounded.

The first, second, and fourth constraints are due to the limited memory and computational resources on a programmable switch. The third one reflects the limited expressiveness of the pipeline architecture in traditional P4 devices; match action tables can only be accessed once during packet processing to ensure efficient packet processing at line rate. The last one is because the Ethernet frame size is limited, and we want to use a large fraction of the frame size to deliver actual packet content. We set this constraint to be 20 bytes.

We can meet all of the five constraints by moving more of the code to the non-offloaded partition. (Note that executing all of the code on the server trivially satisfies the constraints.) Gallium first deals with Constraints 1, 2, and 3. Because once Constraints 1, 2, and 3 are met, moving more code to the non-offloaded partition never violates them again. For (1) and (2), Gallium first computes the following dependency distance metric of various statements from both the beginning and the end of the program. The dependency distance between two program points is the length of the longest dependency chain connecting the two points. Given a programmable pipeline with k stages, it removes "pre" labels from all statements that are at a dependency distance of greater than k from the program's entry point and "post" labels from all statements that are at a dependency distance of greater than k from the program's

[2]The third and fourth rules can be relaxed if the switch supports a disaggregated RMT architecture [8].

[3]We use a conservative constraint on the length, as the actual number of operations that could be performed on the switch is embedded in the P4 compiler and is not publicly available. We choose a conservative value based on empirical experiments.

exit point. This transformation helps us satisfy Constraint 2. Gallium then gradually removes "pre" labels from statements in the reverse source program order and "post" labels from statements in the source program order until Constraint 1 is satisfied.

For Constraint 3, Gallium uses an exhaustive search to find the placement that maximizes the number of statements on the programmable switch. For each global state, Gallium first locates all the accesses to the state that are labeled with *pre* (or *post*). It then enumerates all possible placements of those accesses where only one of them is executed on the switch. Gallium computes the number of statements that could be put on a programmable switch in each case and choose the placement with the maximum number of statements.

Gallium then moves more code to the non-offloaded partition, performing a variable liveness test after each removal to determine the amount of program state that needs to be transferred across partition boundaries, and repeats this until the partitioned code satisfies Constraints 4 and 5. Each time a statement is moved, Gallium runs the label-removing algorithm to ensure that the dependency constraints are met.

However, note that moving code from the offloaded partition does not always reduce the data that has to be transferred to the non-offloaded partition. For example, moving an integer addition from the switch to the server requires the offloaded code to send two integers to the middlebox server instead of one). Finding the minimal set of code that, when moved to the server, could satisfy Constraint 5, requires enumerating all possible combinations of code movements. Gallium chooses to use a greedy approach: It tries to move code to the non-offloaded partition based on a fixed topological order of the data dependencies. This heuristic will give us a sub-optimal result when there are multiple branches in the offloaded code. For instance, the greedy approach only checks the second branch after the first branch is all removed, instead of considering all possible orders of moving code fragments. Nevertheless, it only requires one linear scan of the offloaded code, and therefore, it can find a code partition that satisfies Constraints 4 and 5 in a reasonable amount of time.

Gallium assigns a partition for each statement by looking at the labels. When a statement has both "pre" and "post" labels, Gallium assigns the statement to the pre-processing partition. When a statement has the "post" label but not the "pre" label, Gallium assigns it to the post-processing partition. The rest of the code is assigned to the non-offloaded partition. Gallium then splits the input program's control flow graph (CFG) into separate CFGs that correspond to the three partitions. Figure 4 shows the resulting CFGs for `MiniLB`.

4.3 Code Generation and Runtime Execution

We now describe the code generation process and the mechanisms used for communicating state between the switch and the server. First, we explain how we transform the CFGs of the pre- and post-processing partitions to a P4 program. For the non-offloaded partition, Gallium needs to convert the corresponding CFG back to C++. Transforming from a CFG to C++ is easy because C++ is very expressive, and we omit the details for this transformation. Second, we describe how the temporary state can be communicated in-band by customizing the packet format used between the switch and the server. Third, we detail the runtime mechanisms for synchronizing global state and how we can provide run-to-completion semantics for concurrent packet processing.

4.3.1 Mapping CFG to P4 We now describe how we map a CFG to a P4 program. Figure 6 shows how we map different states and instructions in the CFG to corresponding P4 primitives.

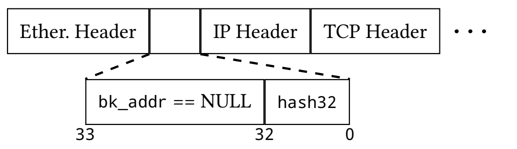
The per-packet state, such as temporary variables, are mapped to metadata fields in the scratchpad memory. Since the amount of metadata that can be allocated is less than 100 bytes to conserve on scratchpad memory, Gallium records when temporary variables are first and last used. Gallium reuses the memory consumed by variables that are no longer useful. In `MiniLB`, `key` and `bk_addr` are temporary variables.

For the global state, Gallium chooses different representations based on the access pattern. States that are accessed exclusively by the switch will be maintained on the switch if there is a P4 realization of the state. Gallium supports two types of global state on the programmable switch: global variables and maps. Gallium maps these two types of switch state into P4 match-action tables and P4 registers, respectively. Note that Gallium choose to use P4 registers for global variables only if the switch alone accesses the state. Similarly, states that are exclusively accessed by the server are represented with the same data structure as the original program. For the program state accessed by both the switch and the server, Gallium maintains a copy of the state on both devices. State replication enables faster access but complicates updates, as we will discuss later. In `MiniLB`, `map` is a HashMap in the input middlebox, and it is offloaded as a P4 match-action table. Because a C++ `HashMap` can be unbounded in size, but programmable switches have a limited amount of memory, Gallium requires a middlebox developer to annotate a maximum size for each `HashMap` that the developer wishes to offload.

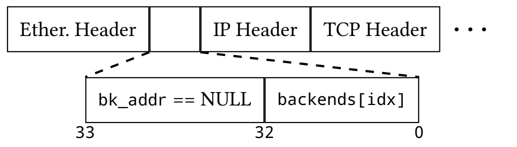
Once states are mapped to their P4 counterparts, Gallium start to maps each instruction to P4. Branches, packet header accesses, and ALU operations are directly mapped to their P4 counterparts. Map lookups in the input middlebox program are mapped to P4 match-action table lookups. In `MiniLB`, `map.find(&key)` is mapped to a P4 table lookup.

We also combine the pre- and post-processing partitions into a single P4 program. Gallium includes all the match-action table and register definitions from the two partitions. The instructions from the two partitions are also placed in the combined P4 program. To determine which partition should execute when receiving a packet, Gallium creates a match-action table that matches on the ingress interface of the packet at the beginning of the processing pipeline. If the packet is coming from the interface connected to the middlebox server, Gallium invokes the post-processing partition. Otherwise, the pre-processing partition handles the packet.

4.3.2 Communicating temporary state between the switch and the server Some temporary state has to be communicated between the switch and the server. Gallium transmits this state along with the packet data. We determine the packet format for packets going from the pre-processing partition to the non-offloaded partition and from the non-offloaded partition to the post-processing partition. Gallium does a variable liveness test on the partition boundary to decide what variables need to be transferred across



(a) Packet from the pre-processing to the offloaded partition



(b) Packet from the non-offloaded to the post-processing partition

Figure 5: The packet format used between programmable switch and middlebox server in `MiniLB`.

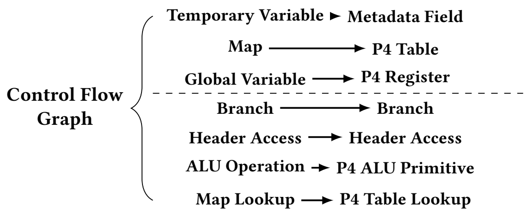

Control Flow Graph
Temporary Variable → Metadata Field
Map → P4 Table
Global Variable → P4 Register
Branch → Branch
Header Access → Header Access
ALU Operation → P4 ALU Primitive
Map Lookup → P4 Table Lookup

Figure 6: Mapping a control-flow graph's states and instructions to their P4 counterparts.

partition boundaries. [4] Gallium allocates space in the packet header to store these variables and generates the corresponding packet header parser specification in P4.

We insert these additional packet header fields between the Ethernet header and the IP header. The Ethernet header is expected by the receiving NIC and ensures that the packet is delivered over the wire to the middlebox server's NIC. We don't need to put the additional header after the IP header because we expect the programmable switch to have a direct connection to the middlebox server; thus, a destination Ethernet address is sufficient for routing. To accommodate this additional header, we use a slightly larger maximum transmission unit (MTU) for the link between the programmable switch and the middlebox server compared with the rest of the network.

Figure 5 shows the packet format for `MiniLB`. The condition for branching, `bk_addr==NULL`, has to be shared across all three partitions. Gallium allocates 1 bit in the packet header for this information. Also, `hash32` is communicated from the pre-processing partition to the non-offloaded partition. `backends[idx]` is communicated from the non-offloaded partition to the post-processing partition. Gallium allocates 32 bits in the packet header for storing these variables.

4.3.3 Synchronizing global state

We now describe the state synchronization techniques required to provide the desired run-to-completion semantics. Recall that this execution semantics requires a packet's state updates to be atomic, i.e., other packets observe all or none of the state updates performed by a packet. Further, a packet is required to observe all of the state updates performed by causally antecedent packets.

It is worth noting some aspects of the desired run-to-completion semantics, which we had described earlier in §3. The correctness criteria does not impose execution restrictions on packets that are traversing the network at about the same time, since those packets could be processed by a middlebox server in any order. Instead, a packet P's causally antecedent packets are precisely those packets that have already been received by one of the end-hosts before P is transmitted. We just need to ensure that the state updates of these causally antecedent packets have been consistently replicated across the switch and the server.

Further, the correctness criteria does not provide ordering guarantees. That is, packet P that is not causally dependent on packet Q might observe the state changes performed by packet Q, but might still be transmitted by the middlebox (i.e., the server-switch pair) before packet Q. We believe that this is not unreasonable as this is equivalent to packet reordering introduced on multi-threaded middlebox implementations or even by the network itself.

Distributed state management: We first note that our partitioning and code generation techniques ensure that, when a global variable or map is replicated across the switch and the server, any updates will only be made by the server. Given this placement and access restriction, the required run-to-completion semantics is equivalent to supporting transactional (or serializable) operations on the state, with the switch operations being limited to read-only accesses. Given this observation, we borrow two techniques from the distributed systems literature on primary-backup systems to enforce the desired semantics. *Atomic update* ensures that the server updates on the shared state are atomically reflected on the switch. *Output commit* ensures that the packet causing the updates is buffered on the server until the updates are reflected on the switch. When used together, the techniques will guarantee the desired run-to-completion semantics.

Atomic update: We now describe the implementation of the atomic update in Gallium. Similar to journaling in file systems, Gallium's runtime system first puts all the modifications into a dedicated switch memory and then uses a single (atomic) write operation to make them visible to subsequent packets. For each match table stored on the programmable switch, a smaller-sized "write-back" table is created. Besides that, a single bit is also added to the switch state, indicating whether the write-back table should be used during table lookup. When the P4 program performs a lookup to the table and observes that the bit is set to true, it first reads the write-back table. If there is a matching entry, it will be used as the result of the table lookup. Otherwise, the main match table will be used. The middlebox server performs switch state updates in three steps. First, the server uses the switch control plane API to insert entries to the write-back tables. (A special value indicates table entry deletion.) Then, the server flips the bit by performing an additional control plane operation. This operation makes the entries in the write-back tables visible to subsequent packets. Finally, the middlebox server writes the updates to the main tables and toggles the bit after all updates are performed.

[4] In the partitioning step, Gallium has already verified that the additional packet header space needed for transferring these variables is below 20 bytes.

Analysis: We now discuss the correctness and performance implications of the scheme described above. The combined use of transactional (or atomic) updates and the output commit protocols ensure that the switch-server pair exhibits the same consistency properties as a chain-replicated, primary-backup system. Switch operations are restricted to read-only operations on replicated state, just as with the tail of a chain-replicated system [27]. A packet that performs updates to replicated state is only released after the switch has performed the updates. Subsequent packets generated after an end-host has received would see the updated state even when processed on the switch.

A Gallium middlebox could reorder packets sent through it, and this reordering could result in performance issues for TCP flows. Consider, for instance, a packet that is forwarded to the server for slow-path processing. If it were to make updates to the replicated state, then it would be buffered until the updates are propagated to the switch, and subsequent packets could be processed and transmitted by the switch before the server releases the slow-path packet. Fortunately, for most middleboxes, the slow-path processing is often invoked on a small number of packets or just control packets, such as SYN, SYN-ACK, FIN, and RST packets, and this reduces the occurrence of reordered data packets. It is also worth noting that features providing dataplane management of the traffic manager, expected in upcoming Tofino switches [25], can help eliminate these reordering issues. With this hardware support, flows with outstanding packets requiring slow-path processing can be buffered on separate queues and then released by the packet transmitted by the server. We leave the exploration of this mechanism for future work.

5 Implementation

We implement Gallium using 5712 lines of C++. Gallium uses *Clang* (version 6.00) to generate an LLVM Intermediate Representation (LLVM IR) of the input program. All the functionalities described in §4 are implemented as analysis passes on the LLVM IR as it has a simpler syntax than C++. Also, because LLVM IR itself is in a Static Single Assignment (SSA) form, it eases the tracking of when variables are assigned and used. We use the *llvm-dev* library to extract a CFG of the input program. The generated P4 code is compiled and deployed using the SDK provided by the Barefoot Tofino switch. The non-offloaded partition (C++) is compiled and linked with the DPDK library [1] and is deployed as a DPDK application.

As we mentioned in §4, Gallium has a set of annotations on Click APIs in order to perform dependency extraction. We have manually annotated the Click APIs to access data structures, including `Vector` and `HashMap`, and the APIs to access packet headers. Gallium models the read and write set for each LLVM instruction. For ALU instructions, the read set of the instruction consists of all the instruction operands, and the write set is simply the destination register. For memory access instructions—load/store—the read/write set of the instruction is the data the pointer points to. Gallium leverages LLVM IR's type metadata to determine the type and size of the dereferenced pointer.

6 Evaluation

We aim to answer the following questions in this section:

Middlebox	Input (C++)	Output (P4)	Output (C++)
MazuNAT	1687	516	579
Load Balancer	1447	522	602
Firewall	1151	506	403
Proxy	953	292	279
Trojan Detector	882	571	418

Table 1: Comparison of lines of code for Click-based middleboxes before and after Gallium compiles them.

- Can Gallium enable automated software middlebox offloading to programmable switches?
- How much performance benefits do the offloaded middleboxes provide?

6.1 Case Study

We use five Click-based middleboxes to evaluate Gallium: (1) MazuNAT, (2) an L4 load balancer, (3) a firewall, (4) a transparent proxy, and (5) a Trojan detector.

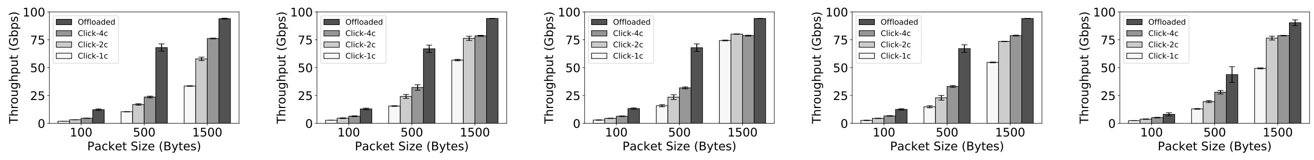
MazuNAT. MazuNAT is a NAT implementation used by Mazu networks. At a high level, MazuNAT is a gateway middlebox that separates two network spaces, an internal network and an external network. For traffic going from the internal to the external network, MazuNAT allocates a new port and rewrites the packet header, and the flow itself appears to be sourced by MazuNAT. The port allocation is performed using a monotonically increasing counter. MazuNAT memorizes the mapping from addresses to ports for existing connections and enforces the mapping for subsequent packets of existing connections.

When MazuNAT receives a packet from the external network, MazuNAT checks if there is a corresponding mapping created by connections from the internal network. If not, MazuNAT drops the packets from the external network. If a corresponding mapping is found, MazuNAT rewrites the packet according to the mapping and forwards it into the internal network to reach its destination.

L4 Load Balancer. The load balancer application is similar to the `MiniLB` example show in §4. Similar to `MiniLB`, the load balancer assigns incoming TCP and UDP traffic to a list of backend servers. It uses the hash value of the five-tuple (i.e., source IP address/port, destination IP address/port, transport protocol) to determine the backend server to which the connection is assigned. It also uses a map to keep track of the assigned flows to ensure that packets in the same connection are steered to the same backend server. In addition to the functionalities in `MiniLB`, the L4 load balancer garbage-collects finished connections by intercepting TCP control packets, such as RST (reset) and FIN (finish). The L4 load balancer also has a time-out mechanism: idle connections are garbage-collected after 5 minutes without the FIN packet.

Firewall. The firewall is adapted from an example middlebox in the Click paper [21]. It filters packets using a whitelist mechanism. Each entry specifies a five-tuple that is allowed to go through the firewall. When a packet arrives, it is dropped if its five-tuple cannot be found in the whitelist.

Transparent Proxy. The transparent proxy is also adapted from an example in the Click paper [21]. The transparent proxy redirects traffic to a web proxy based on the TCP destination port. The proxy



(a) MazuNAT (b) L4 Load Balancer (c) Firewall (d) Proxy (e) Trojan Detector

Figure 7: Throughput comparison between Gallium middleboxes and their FastClick counterparts. Gallium middleboxes only use a single core in the middlebox server. FastClick versions of the middleboxes use 1, 2, and 4 cores, respectively. Error bars denote standard deviations.

internally keeps a list of TCP destination ports. Upon receiving a packet, the proxy checks whether the TCP destination port is in the list. If the destination port is in the list, instead of forwarding the packet, the proxy rewrites the packet header to steer the packet to a designated web proxy.

Trojan Detector. The Trojan detector [9] keeps track of TCP connection states of each endhost. It identifies an endhost as a Trojan if the following sequence of events is observed: (1) The endhost first creates an SSH connection. (2) It then downloads a HTML file from a web server, or a `.zip` or `.exe` file from a FTP server. (3) Finally, it generates Internet Relay Chat (IRC) traffic.

6.2 What's offloaded?

To evaluate how much middlebox functionality can be offloaded, we examine the lines of code before and after the five middleboxes are compiled by Gallium. Table 1 shows the result. Here, the total lines of code do not include Click data structure implementations. After compilation, the generated code (i.e., the combination of the P4 and C++ code) has fewer instructions than the input program, as P4 abstracts away several types of packet processing. For example, the `IPClassifier` element, which performs generic packet classification, could be abstracted using a single match action table in P4.

After compilation, MazuNAT's address translation tables—the maps that store the five-tuple rewriting rules for both internal and external TCP flows—are offloaded to the programmable switch. Besides that, the counter used for port allocation is also offloaded to the switch as a P4 register. When new rewriting rules have to be added to the address translation table, the pre-processing code will pack the current counter value into the packet header and send it to the middlebox server, where the table update is performed.

When applying Gallium to MazuNAT, we added the annotation that the address translation mapping would not have more than 65536 (2^{16}) entries, since each port number can have at most one entry in the map. This annotation allows Gallium to place the address translation maps on the switch.

Similar to the offloaded version of `MiniLB`, Gallium produces an offloaded version of the load balancer where the connection consistency map is stored in the switch. New incoming connections and packets with TCP control flags (RST and FIN) will be forwarded to the middlebox server, as handling those packets requires an update to the map.

The P4 program generated for the firewall middlebox contains two match-action tables to filter the traffic from both directions. These tables offload the functionality performed by the `IPClassifier` elements used by the firewall. The 403 lines of code in the non-offloaded part are mainly for constructing and inserting the firewall rules.

For the proxy, the pre-processing code contains one match-action table that checks the incoming TCP packets' destination port. A packet rewriting action is also included in the P4 program to rewrite the TCP packet's destination to the web proxy server.

Gallium places Trojan detector's TCP flow state table on the programmable switch. TCP control packets, such as SYN or SYNACK, triggers a table update. These packets are forwarded to the middlebox server. Besides that, HTTP requests from an endhost that have received SSH traffic before are also be processed by the middlebox server to determine the type of requested file. Most of the TCP data packets that do not require deep packet inspection are handled solely by the programmable switch.

6.3 Performance

We first microbenchmark, for each of our five middleboxes, the throughput, latency, and CPU overheads. We then evaluate the performance overhead introduced by performing state synchronization when updating the state replicated on the switch. After that, we evaluate the five middleboxes using realistic workloads.

Experiment Setup. Our testbed consists of three servers and a Barefoot Tofino switch. Each server has an Intel Xeon E5-2680 CPU (2.5GHz, 12 cores) and a Mellanox ConnectX-4 100 Gbps NIC. Servers run Ubuntu 18.04 with Linux kernel version 4.15. All the three servers are connected to a Barefoot Tofino switch via 100 Gbps links. We dedicate one server to be the middlebox server. The middlebox server runs DPDK version 17.11. These two servers use traditional Linux networking stacks to generate and receive packets.

To compare performance with non-offloaded middleboxes, we use FastClick [5] to run non-offloaded middleboxes in the middlebox server and configure the routing table in the switch to ensure all packets go through the server.

TCP Microbenchmark. We generate ten parallel TCP connections using `iperf` to test the maximum achievable throughput of the middleboxes. When we run Gallium middleboxes, we restrict the usage of the processing in the middlebox server to be on a single core. For FastClick, we test the middleboxes with 1, 2, and 4 cores. We also test different packet sizes (e.g., 100, 500, and 1500 bytes). We test the throughput ten times and measure the average and standard deviation.

Gallium substantially improves middlebox throughput and reduces CPU overheads on the middlebox server. Figure 7 compares

Middlebox	FastClick	Gallium
MazuNAT	23.16 ± 0.53 μs	15.98 ± 0.21 μs
Load balancer	23.09 ± 0.31 μs	15.96 ± 0.20 μs
Firewall	22.45 ± 0.27 μs	15.96 ± 0.20 μs
Proxy	22.72 ± 0.87,μs	15.64 ± 0.85 μs
Trojan Detector	22.58 ± 0.74,μs	14.80 ± 0.43 μs

Table 2: Latency comparison of Gallium middleboxes and their FastClick counterparts. The numbers after ± denote the standard deviations.

# tables	Insert	Modify	Delete
1	135.2 ± 22.0 μs	128.6 ± 23.6 μs	131.3 ± 18.8 μs
2	270.1 ± 33.0 μs	258.3 ± 34.9 μs	262.7 ± 29.8 μs
4	371.0 ± 39.2 μs	363.0 ± 37.3 μs	366.1 ± 37.7 μs

Table 3: Latency of updating offloaded P4 tables from middlebox server.

the maximum achievable throughput of Gallium middleboxes and their FastClick-based counterparts. Overall, compared with FastClick running on 4 cores, Gallium with a single core outperforms by 20-187%. If we constrain the throughput to be identical, Gallium saves processing cycles by 21-79%. These performance benefits are because the non-offloaded partitions in the Gallium middleboxes are rarely used. In the Gallium version of MazuNAT and load balancer, only 0.1% of the packets in TCP flows are processed by the middlebox server. For the firewall and the proxy, all packet processing happens in the programmable switch.

Gallium reduces latency by 31%. We use `Nptcp` to test TCP packet latency. Table 2 shows the result. Like the reduction in processing overheads, the latency reduction comes from the fact that most packets do not go through the server in Gallium.

State Synchronization Overhead. We create programs with different table sizes and different numbers of tables to test the latency required to update offloaded tables from middlebox servers. We measure the latency of updating 1, 2, and 4 tables. Table 3 shows the latency of insertion, modification, and deletion for offloaded tables. A single table update is about 5x the end-to-end latency of a packet sent through a software middlebox. Gallium can provide overall performance benefits as long as the slow path and corresponding state synchronization operations are invoked infrequently. We consider this next using realistic traffic traces through the generated middleboxes.

Realistic Workloads. We evaluate two realistic workloads: an enterprise workload and a data-mining workload. The workloads (i.e., flow size distributions) are drawn from the CONGA work on datacenter traffic load balancing [4]. These workloads have both short flows and long flows. The majority of flows in both the enterprise and the data-mining workload are small; 90% of the flows in both workloads contain less than ten packets. We draw 100000 flow sizes from the flow size distributions and use 100 threads to send traffic. A thread sends a single connection at a time and starts a new connection when the current connection finishes.

To evaluate Gallium's throughput and CPU benefits, we fix the number of cores used in the middlebox server and use the RX counter of the receiver side's NIC to compute the average throughput during the experiment.

Gallium improves the overall throughput and reduces CPU overheads on both workloads. Figure 8 shows the result. Compared to the 4-core version of FastClick, Gallium using one core can achieve 1-35% more throughput on the enterprise workload and 18-46% more throughput on the data-mining workload. If we constrain the throughput to be the same as what 4-core FastClick can achieve, this means we can save 3-81% processing cycles or 0.03-4.39 server cores. We do better on the data-mining workload because the long flows are longer than that in the enterprise workloads.

Which connections do Gallium speed up in these realistic workloads? We measure the flow completion time for connections of different sizes. We bin flows based on their sizes and compute an average flow completion time for each bin. Figure 9 provides the comparison between Gallium and FastClick. We see that the reduction in flow completion time is concentrated on the long flows. This behavior is because long flows will have the majority of their packets handled by the programmable switch instead of the server.

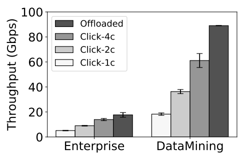
7 Discussion

Reducing memory usage of programmable switches. One optimization to reduce memory usage of programmable switches is to let the programmable switch store only a fraction of any table, e.g., `MiniLB`'s `map` that stores the mapping from IP addresses to ports. For any packet that the programmable switch does not know how to handle, the middlebox server handles it instead. This means we need to handle the additional challenge of synchronizing the table in the software middlebox and its "cache" table in the programmable switch. We leave it to future work.

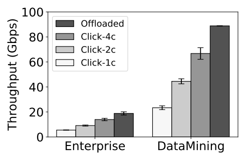
Extra functionalities in programmable switches. The P4 program generated by Gallium only uses registers and match-action tables with exact matches. Besides these functionalities, the P4 language provides other useful abstractions for packet processing, such as the longest prefix matching when matching on packet header fields, such as IP addresses. Programmable devices may also provide platform-specific P4 primitives such as advanced ALU operations (e.g., swapping the lower- and higher-32 bits of a 64-bit integer) or hardware-accelerated hash functions. Currently, these functionalities are not used by Gallium as some abstractions, such as LPM, do not exist in software middleboxes.

Dealing with complex data structures. Currently, Gallium can only handle two data structures, `HashMap` and `Vector`. This is because we know how to map them to P4 native primitives (§4.3). We find that these two data structures are the most commonly used ones in Click. Middleboxes that use complex data structures (e.g., linked list, tree) have to be processed as part of the non-offloaded partition or require code modifications to enable offloading.

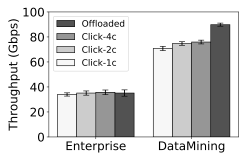
Cost model of offloading. Currently, Gallium tries to offload as many lines of code (LLVM instructions) as possible to the programmable switch. This approach simplifies the algorithm design, as discussed in §4.2. This model does not consider that the offloading of different operations will give the middlebox different performance benefits. For example, offloading a table lookup to the programmable switch will provide more performance benefits than offloading an integer addition operation. Therefore, Gallium's partitioning algorithm may produce sub-optimal partitions as the



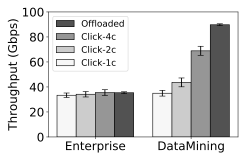
(a) MazuNAT



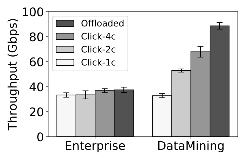
(b) L4 Load Balancer



(c) Firewall

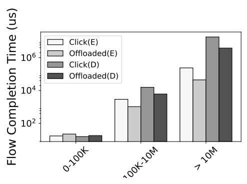


(d) Proxy

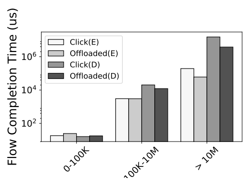


(e) Trojan Detector

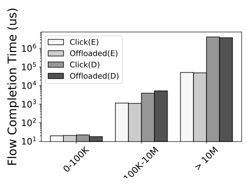
Figure 8: Throughput comparison of Gallium and FastClick on the enterprise workload and the data-mining workload.



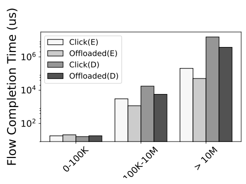
(a) MazuNAT



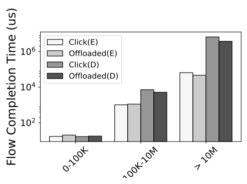
(b) Load Balancer



(c) Firewall



(d) Proxy



(e) Trojan Detector

Figure 9: Flow completion time comparison of Gallium and FastClick on the enterprise(E) and data-mining(D) workload.

algorithm may choose to offload an integer addition over a table lookup. One possible solution to this is to assign a weight to each operation, which would be a measure of its associated performance benefit when executed on the switch. We would then try to maximize the weight of instructions offloaded to the switch as we move operations into the non-offloaded partition in the algorithm described in §4.2.2. We leave it to future work.

8 Related Work

Offloading to programmable network hardware. Hardware offloading is a well-known technique to improve performance. There have been many studies that use programmable network hardware to accelerate specific applications. For example, KV-Direct [14] is a key-value store on FPGA NICs. IncBricks [19] and NetCache [13] use programmable network hardware to build an in-network cache. NOPaxos [17], Eris [16], and NetChain [12] use programmable switches to do in-network coordination. DAIET [24] aggregates data along network paths using programmable switches. Silkroad [20] leverages a programmable switch's match action table to scale up the number of concurrent connections for load balancers. The current offloading approaches need manual effort in partitioning the source program, while Gallium provides a compiler-oriented approach to partition the source program automatically.

Frameworks for programmable network hardware. Our work is in the category of helping developers move their applications to programmable network devices. Floem [22] allows developers to explore different offloading methods for programmable NICs. iPipe [18] is an actor-based framework for offloading distributed applications onto programmable NICs. Both Floem and iPipe require manually partitioning applications, and they assume the smart NIC can accept programs written in C. ClickNP [15] is another framework using FPGA-based smart NICs for offloading network functions. Similar to Gallium, ClickNP uses the Click [21] dataflow programming model. Different from all this work, our focus is on P4, a hardware-independent target for a class of programmable networking hardware. P4's expressiveness is more restricted than C and FPGA, and Gallium's goal is to partition middlebox source code and deal with resource constraints of P4-based programmable hardware. Domino [26] provides a DSL for developers to write packet processing programs that could be compiled to run on programmable line-rate switches. A major difference between Domino and Gallium comes from the choice of deployment model. While Domino deploys the entire program onto the switch and rejects programs that could not be deployed, Gallium aims at partitioning the program and makes a portion of it deploy-able on the switch.

Program slicing. The compilation techniques to partition a source program based on the dependency relations and control-flow graphs are similar to program slicing [11, 23, 28]. However, program slicing aims to abstract the program: extract a minimal program whose logic is similar to the source program for a subset of the source program's variables. Our goal is to partition the source program into multiple partitions, where the cumulative effect is the same as the source program.

9 Conclusion

Offloading software middleboxes to programmable switches can yield orders-of-magnitude performance gains; however, manually rewriting software middleboxes is hard and time-consuming. In this paper, we have designed and implemented Gallium. Gallium uses program partitioning and compilation to transform software middleboxes to their functionally-equivalent versions, which leverage programmable switches for high performance. Our evaluations show that Gallium can save 21-79% of processing cycles and reduce latency by about 31% across various types of software middleboxes. Gallium's source code is available at https://github.com/Kaiyuan-Zhang/Gallium-public. This work does not raise any ethical issues.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their helpful feedback on the paper. This work was partially supported by NSF (CNS-1714508) and Futurewei.

References

[1] Data Plane Development Kit (DPDK). https://software.intel.com/en-us/networking/dpdk.

[2] In-Network DDoS Detection. https://barefootnetworks.com/use-cases/in-nw-DDoS-detection/.

[3] Tofino: World's fastest P4-programmable Ethernet switch ASICs. https://barefootnetworks.com/products/brief-tofino/.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014.

[5] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[8] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. Drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[9] Lorenzo De Carli, Robin Sommer, and Somesh Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390, 2014.

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[11] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[12] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 35–49, USA, 2018. USENIX Association.

[13] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[14] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[15] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[16] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 104–120, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 467–483, USA, 2016. USENIX Association.

[18] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[20] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[21] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. ACM.

[22] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[23] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52, 1995.

[24] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017.

[25] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G. Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[26] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM.

[27] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, 2004.

[28] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.