Runtime Programmable Switches

Jiarong Xing and Kuo-Feng Hsu, *Rice University;* Matty Kadosh, Alan Lo, and Yonatan Piasetzky, *Nvidia;* Arvind Krishnamurthy, *University of Washington;* Ang Chen, *Rice University*

https://www.usenix.org/conference/nsdi22/presentation/xing

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Runtime Programmable Switches

Jiarong Xing   Kuo-Feng Hsu   Matty Kadosh[†]   Alan Lo[†]
Yonatan Piasetzky[†]   Arvind Krishnamurthy[‡]   Ang Chen

Rice University   [†]*Nvidia*   [‡]*University of Washington*

Abstract

Programming the network to add, remove, and modify functions has been a longstanding goal in our community. Unfortunately, in today's programmable networks, the velocity of change is restricted by a practical yet fundamental barrier: reprogramming network devices is an intrusive change, requiring management operations such as draining and rerouting traffic from the target node, re-imaging the data plane, and redirecting traffic back to its original route. This project investigates design techniques to make future networks *runtime programmable*. FlexCore enables partial reconfiguration of switch data planes at runtime with minimum resource overheads, without service disruption, while processing packets with consistency guarantees. It involves design considerations in switch architectures, partial reconfiguration primitives, reconfiguration algorithms, as well as consistency guarantees. Our evaluation results demonstrate the feasibility and benefits of runtime programmable switches.

1 Introduction

Programming the network to add, remove, and modify functions has been a longstanding goal in the networking community. Programmable switches [3, 8] represent the latest step toward this vision. Using high-level languages like P4 [3], network operators can customize packet processing behaviors at the switch program level. To change network processing, operators can deploy a different P4 program to the data plane, without the need for hardware changes or device upgrades. Programmable switches have enabled a host of new network applications in telemetry [15, 27, 35], measurement [40], security [39], and application offloading [18, 19].

Unfortunately, in today's programmable networks, the velocity of change is restricted by a practical yet fundamental barrier: switch functions are only programmable at compile-time, but they effectively become fixed functions at runtime. The switch program cannot be easily modified at runtime without reflashing the data plane hardware and carefully managing network-wide changes. To reprogram a network switch, operators need to first drain and reroute traffic from the target, install the new program image, and then redirect traffic back to its route. The error-prone nature of network maintenance procedures, the amount of manual coordination required, and the need to satisfy stringent SLAs pose severe constraints on runtime program changes. To the extent that functions can be "hard-coded" in the device, they can be invoked for runtime response [41]. However, new functions that haven't been accounted for, or functions that cannot fit into the switch resources, are difficult to deploy at runtime. This stands in stark contrast to software data planes on host servers, where changes are easily accommodated and functions go through frequent upgrades [12]. The *ultimate* vision of programmable networks that seamlessly incorporates function changes *at any time* (e.g., based on traffic workloads or multi-tenancy requirements) still remains an elusive goal.

In this project, we pave the way toward *runtime programmable switches* by investigating the necessary building blocks and proposing concrete designs for each of them. FlexCore enables switch functions to be *continuously* programmable throughout the lifetime of the network. It develops a new set of control plane API to modify P4 program elements—match/action tables, control flow branches, and parsing graphs—while the switch data plane serves live traffic. These operations precisely instrument the switch program using *partial reconfiguration* primitives without affecting the rest of the data plane. This new modality of network programmability introduces an array of applications:

- Just-in-time network optimizations: When an optimization (e.g., network-accelerated multicast) is needed, it can be added just-in-time to serve the traffic workloads, and removed soon afterwards to keep the network lean.
- Real-time attack mitigation: If network attacks (e.g., DDoS, data exfiltration) are detected, we can inject mitigation modules exactly where needed; new attack patterns would trigger removal of expired modules and the insertion of new program components.
- Scenario-specific network extensions: A tenant can inject switch program extensions to the network. VM migration will carve out and graft the relevant program components to a different location of the network.

Also, telemetry applications do not have to commit to a fixed set of queries [27]; new network protocols can be added and removed dynamically; load-aware routing algorithms can be injected when needed [17]; different versions of switch programs can be deployed for canarying [42]. In fact, many (if not all) of today's programmable network applications will have more powerful, runtime programmable equivalents.

Achieving this goal requires a range of research challenges to be addressed: switch architecture designs that make runtime programmability natural, partial reconfiguration primitives for modifying live switch programs, atomicity and consistency guarantees on runtime changes, and algorithms for effectively computing reconfiguration plans. FlexCore makes contributions in all these dimensions.

Switch architecture. We base our FlexCore design upon a variant of disaggregated RMT (dRMT) [11]. dRMT separates switch memory from compute, and our architecture introduces another twist in its *partial* disaggregation design, where a small compute-local memory holds a indirection data structure that we call a *program description table* (PDT). This table contains metadata about the program control flow and is our target for reconfiguration. Decoupling program logic from its physical realization separates concerns: physical resources can be allocated and deallocated in scratch areas before program elements are modified for the changes to be visible.

Partial reconfiguration primitives. We develop a set of new primitives for adding, removing, and modifying program elements—this includes match/action tables, control flow branches and parser states. Unlike today's control plane API, which manipulates switch memory (e.g., adding/removing entries), the new API reconfigures switch compute.

Consistency guarantees. We propose three consistency guarantees for runtime reconfiguration: program consistency, element consistency, and execution consistency, with increasingly relaxed guarantees. These guarantees constrain the kind of "intermediate programs" that packets are allowed to encounter during partial reconfiguration. Program consistency states that all program modifications must take effect simultaneously. Element consistency is weaker, and states that modifications can be made visible in an element-by-element basis (e.g., one table at a reconfiguration step). Execution consistency is the weakest, but it still guarantees useful properties: packets never traverse execution paths that mix old and new program elements. In all cases, reconfigurations are atomic and do not disrupt data plane forwarding.

Algorithms. We develop algorithms for computing reconfiguration plans for different levels of guarantees.

Evaluation. We implement our design on a 12.8 Tbps merchant silicon (Nvidia Spectrum-2 SN3000 series), as well as a software simulator based upon bmv2. We evaluate the scalability of the reconfiguration algorithms and demonstrate a set of use cases in hardware and software platforms, showing that FlexCore enables a truly adaptive network core.

2 A Case for Runtime Programmability

The quest for network programmability has been an important undertaking in the community. Network switches used to be blackboxes, with opacity at both control and data planes. OpenFlow SDN opened up the control plane for programmatical control, and as of late, programmable data planes enable flexible packet processing pipelines without hardware upgrade. Operators can customize the data plane by removing unnecessary switch functions or adding new ones at the program level. P4 switch programs are compiled into a binary image, which is flashed to data plane hardware for deployment. Researchers have seized this opportunity to systematically rearchitect network telemetry [15, 27, 35], measurement [40], security [39], and application offloading [19].

However, today's programmable data planes have a notable limitation—they cannot be reprogrammed at runtime. If an operator can anticipate all required functions at compile time, and if these functions can fit into the switch resource constraints, then they can be combined and deployed together in the switch. But once deployed, the switch is committed to the hardcoded behaviors as specified at compile time, until the next program reflash. At runtime, only 'micro' changes are permitted, such as modifying flow table entries or register values from the control plane. This affords some flexibility [41]; however, as macro-level program logic changes are hard to make, accommodating requirements that truly arise 'on-demand' (e.g., security incidents) remain an elusive goal. Also, since switches have constrained resources, even if we had an 'oracle' planner that anticipates all needed functions, they may not fit into the switch together at compile time.

To remedy this problem, we need runtime programmable switches. This not only enables new use cases as motivated above, but also calls for a rethink as to how networks can be specialized. The operator can, at any point in time, aggressively optimize the network data plane to only retain a minimal amount of processing logic. This reduces switch resource footprints, improves network energy efficiency, and also keeps network latency at a minimum. If extra functionality is required, the program elements can be injected precisely where and when they are needed. If a functionality is no longer in use, it can be removed to ensure that the data plane stays at its leanest. Viewed from the lens of the classic 'end-to-end' arguments [31], in-network processing no longer incurs a common overhead to all applications.

3 The FlexCore Switch Architecture

Our switch architecture adopts a disaggregated RMT model [11], where compute resources (i.e., match/action processors) are split from memory (i.e., SRAM/TCAM), and they are interconnected via a crossbar. Each MA processor holds a copy of the P4 program, and processes packets in a run-to-completion manner.

In the RMT architecture [8], each stage contains a slice of compute and memory resources that cannot be reassigned to other stages. This tight coupling makes runtime reconfigurations challenging. For instance, inserting an MA table to a stage may require device-wide table shuffling and reallocation to make space. Removing an MA table from a stage will leave 'holes'—fragmented resources that cannot be easily reused by other program elements. These operations can be intrusive.

A disaggregated architecture, on the other hand, breaks

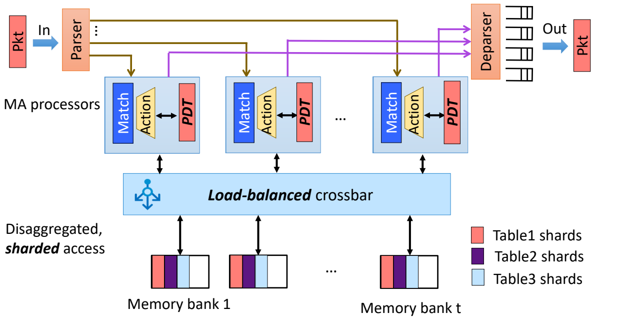


Figure 1: The FlexCore switch architecture. Highlighted in bold+italic are the customizations to dRMT [11].

resource allocation boundaries and enables reconfigurations to be performed locally—i.e., it enables *partial reconfiguration*. If a reconfiguration releases a table, the deallocated resources can be dedicated to *any* other program elements irrespective of their 'locations'. New tables can be inserted to any part of the program without having to change existing resource allocation decisions. Similar properties hold for resources that implement control flow branches and parsing graphs.

dRMT customizations. Our silicon also implements several customizations for performance, flexibility, and usability. Figure 1 shows the high-level architecture.

(i) Sharded resource allocation. In the dRMT architecture, an MA table is allocated in one specific SRAM/TCAM bank. Simultaneous accesses to the same table (or different tables in the same memory bank) from different processors creates contention at the crossbar. In FlexCore, all tables are t-way sharded, where t is the number of memory banks. When inserting a table entry, FlexCore first computes a hash h from the match key as the shard ID, and then allocates the entry in the h-th SRAM/TCAM bank. When performing a match, the same hash function is computed to retrieve the shard ID. This allows FlexCore to sustain linerate without complex mechanisms to detect and avoid access contention. The crossbar is always load-balanced and has uniform access patterns.

(ii) Hybrid programmability. Our switch exports a set of fixed-function ASIC modules as common building blocks (e.g., L2 bridging, L3 routing). These functions can be called by or bypassed from the P4-programmable logic. The fixed blocks are more resource- and energy-efficient, as their implementations are heavily-optimized, hardwired ASIC. By providing these building blocks, P4 programmers don't need to redevelop them from scratch. Moreover, they also represent a minimum "baseline" program that, if necessary, traffic can always fallback on during reconfiguration.

(iii) Indirection. FlexCore employs a partially disaggregated design, where each processor has a small amount of local SRAM to store a special *program description table (PDT)* for indirection. Accesses to PDT do not go through the crossbar and enjoy lower latency. PDT stores the 'program skeleton'—the control flow graph—and decouples the control flow operations from main SRAM accesses. Our partial reconfiguration mechanisms make heavy use of the PDT to modify program elements. Similarly, a parser state table (PST) serves as indirection for the parsing hardware.

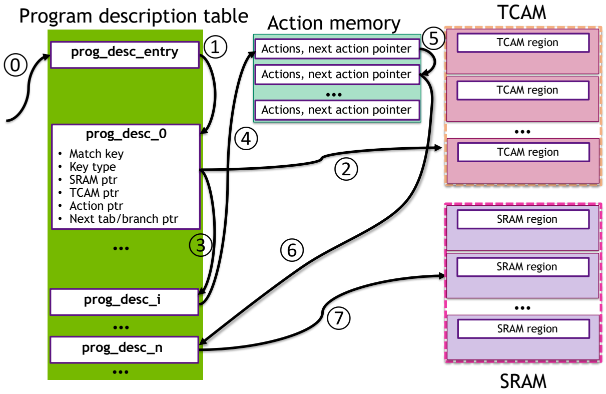


Figure 2: Runtime reconfigurable tables and control flow branches, the indirection mechanism via the program description table (PDT), and an example execution as illustration.

4 Runtime Reconfiguration Primitives

FlexCore introduces a set of novel primitives that, when invoked by the control plane, partially reconfigure a P4 switch program. These primitives operate on a graph representation of a P4 program by adding, removing, or modifying nodes and edges. In a P4 program, the match/action logic is captured by the 'table flow graph' [8], where nodes represent MA tables or conditional branches (realized in table-independent actions), and edges represent non-conditional, table dependency control flow. For the parser logic, the nodes represent parser states (which also contain header extraction rules), and the transition rules are the edges. Next, we first describe the primitives on the table flow graph and then the parsing graph.

4.1 Program description table

A key indirection data structure that enables partial reconfiguration of the table flow graph is what we call a *program description table (PDT)*, as shown in Figure 2. Each match/action processor maintains a local PDT and it is dedicated to a specific switch port. All packets arriving at a port will first hit a default entry in the PDT to activate packet processing.

The entries in the PDT are compiled from a P4 program. Each entry stores metadata about a *program element*, which could be a match/action table or a table-independent ALU action that implements conditional control flow. The metadata contains entry type, match key/type, and a resource pointer that refers to the physical realization of that program element. The pointer address could be an SRAM location (for exact and algorithmic ternary matches), a TCAM location (for ternary matches), or an action location (for conditional branches)—with a 'union' semantics as only one pointer type can be valid for a PDT entry. The address is specified by the base address of a memory region, the size of the region, as well as the offset from the base address. Each PDT entry also contains a 'next' pointer, which encodes unconditional control flow to the next program element (i.e., MA table or conditional branch).

This indirection provides several advantages for runtime programmability: a) operations for adding and removing a program element are decoupled from resource allocation operations, as the first occur in the PDT and the second in the memory regions; b) PDT entries serve as a local scratch—entry modifications are lightweight and do not touch switch-wide shared resources, and they can be changed in a transactional manner. The PDT enables runtime reconfiguration of match/action tables and the control flow graph, which we discuss next.

4.2 Runtime reconfigurable tables

MA tables are the key processing elements in a P4 program. FlexCore enables the addition and deletion of tables using several partial reconfiguration primitives.

Allocation+deallocation. AllocTbl(t) allocates a new table, and DeallocTbl(t) deallocates an existing one. Both are control plane operations that have a centralized view of PDT tables, and they accept the table definition t as the input argument. Allocations first identify free slots to create new PDT entries. In a new PDT entry, the match key and type are filled in with the specified table attributes. SRAM and TCAM resources are then allocated based on the table attributes, and both are sharded across all memory banks. Finally, the control plane fills in the resource pointer, finishing the table allocation. Deallocations could directly remove the entry and its resources, or it may defer their removal to a later garbage collection phase. (Actual table entries are added/removed just like in today's switches, via existing control plane API such as that defined in P4Runtime [5].) Importantly, allocation/deallocation operations are not visible to network traffic until we invoke insertion/deletion primitives.

Insertion+deletion. Changes are made visible via another primitive: SetPtr(t,nxt) modifies t's next pointer to nxt. Table insertions invoke multiple SetPtr calls to place t in the program; deletions perform the opposite operations. Insertions must happen after resources have been allocated, and deletions before deallocation. Each pointer change is atomic in hardware. (To ensure atomicity for a collection of changes, we need another mechanism called a 'flex branch' as discussed later.) Insertions and deletions alter the view of the program state from the perspective of network traffic.

4.3 Runtime reconfigurable control flow

Conditional branches are implemented in ALUs as table-independent actions. Like tables, a conditional branch takes up one PDT entry, but its resource pointer addresses the action memory instead of SRAM/TCAM. In addition, the PDT entry for a conditional branch has a null 'next' pointer; its two jump addresses are instead encoded in the ALU action, one for each branch condition. N-way conditionals are implemented as cascading binary branches. Control flow branch modifications are performed using the following primitives.

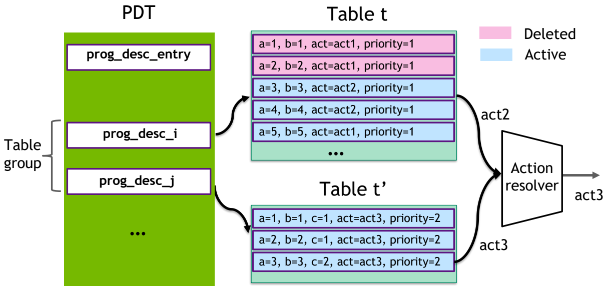

PDT
prog_desc_entry
Table group
prog_desc_i
prog_desc_j
...
Table t
a=1, b=1, act=act1, priority=1
a=2, b=2, act=act1, priority=1
a=3, b=3, act=act2, priority=1
a=4, b=4, act=act2, priority=1
a=5, b=5, act=act1, priority=1
...
Deleted
Active
act2
Table t'
a=1, b=1, c=1, act=act3, priority=2
a=2, b=2, c=1, act=act3, priority=2
a=3, b=3, c=2, act=act3, priority=2
act3
Action resolver
act3

Figure 3: Primitives for in-place table modification.

Allocation+deallocation. FlexCore introduces a primitive, AllocCond(b, pred, br$_1$, br$_2$), to allocate a control flow branch based on pred, where br$_1$ and br$_2$ are the jump addresses for the true and false branches, respectively. Allocation of an N-way conditional is performed by successive invocations of AllocCond with cascading jump addresses. A predicate pred corresponds to an ALU action that checks the condition and produces a true/false evaluation. This binary result is consumed by a hardware 'goto' microcode that jumps to the next program element. If pred evaluates to true, 'goto br$_1$' directs the control flow to the next table or a cascading branch; otherwise, it branches to br$_2$. Deallocations free action memory and PDT entries.

Insertion+deletion. A conditional branch can be activated by a) SetPtr(t,b), which points a table's next pointer to the new branch b, and b) SetCondPtr(b,n$_1$,n$_2$), which sets one or both of the jump addresses of a branch. In the case where SetCondPtr modifies two pointers, the operation is not atomic. Atomicity is achieved similarly using 'flex branches' that we will discuss later. Deletions achieve opposite effects.

4.4 In-place table modifications

So far, all primitives that we have described can be used at any level of consistency guarantees. In this and the next subsections, we describe two special sets of primitives for table modifications and parser reconfigurations as well as their respective consistency properties.

Table modifications can be performed by adding a new table and deleting the old, in which case the intermediate state has size $2\times|T|$ (assuming both tables have size $|T|$). But FlexCore also exposes a more efficient primitive to reformat a table in-situ with an intermediate state of $|T|$. A ModTbl(t,t$'$) primitive reformats t using the definition as specified in the new table definition t$'$, which could include new match key/type and actions. This is achieved by a PDT mechanism called *table groups*. Several PDT entries can be 'grouped' together and processed in parallel at the MA processor. ModTbl creates a new PDT entry using t$'$ and groups it with the entry for t. It then gradually moves entries from t to t$'$, reformatting each entry using the new key or action, and setting the entries in t$'$ with higher priority. In this transient state, the MA processor looks up both tables and resolves

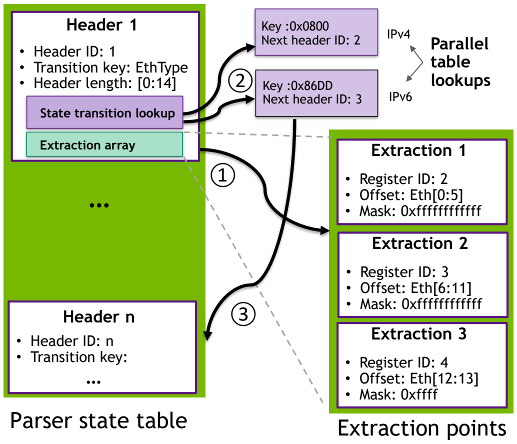


Figure 4: Runtime reconfigurable parsers and the indirection mechanism via the parser state table (PST).

them using an *action resolver* that chooses the higher-priority result. When T become empty, the PDT entries are de-grouped and T gets deleted. ModTbl triggers simultaneous applications of parallel tables, so this mechanism is different from the 'flex branch'. We will discuss its consistency guarantees later in Section 4.6.

4.5 Runtime reconfigurable parsers

Header parsing logic requires different mechanisms for reconfiguration. We describe the parser hardware next, and then the reconfiguration primitives for the parser graph.

The parser state table (PST). Figure 4 presents the hardware architecture for the reconfigurable parser. The key indirection data structure is a *parser state table (PST)*, which stores an array of parser states. Each entry stores a) parsing information for that header, b) an extraction array that extracts header fields, as well as c) a parallel transition lookup component that determines the next state based on the current header values. Similar as the PDT, this indirection ensures that state additions and removals are easily achieved at runtime.

The PST implements a finite state machine, where each entry represents one state and contains transition rules to other states. This array is indexed by a logically assigned header ID that starts with one and ends with the maximum state ID as constructed from the program. When a packet comes in, it first matches against the default entry (ID=1) for parsing. At every step, the hardware uses the 'header length' and 'transition key' defined in the current entry (as well as a base register that remembers how much data has been parsed) to identify the correct offset into the packet. A chunk of data of the size 'header length' is then sent to extraction logic, which uses shift-and-mask to further segment the data chunk into multiple fields (e.g., EtherType, SMAC, DMAC) of varied sizes. These extracted fields are stored in an *extraction array* that is associated with the current header entry. These are further combined using a recombiner into a PHV (packet header vector) and streamed to the ingress blocks.

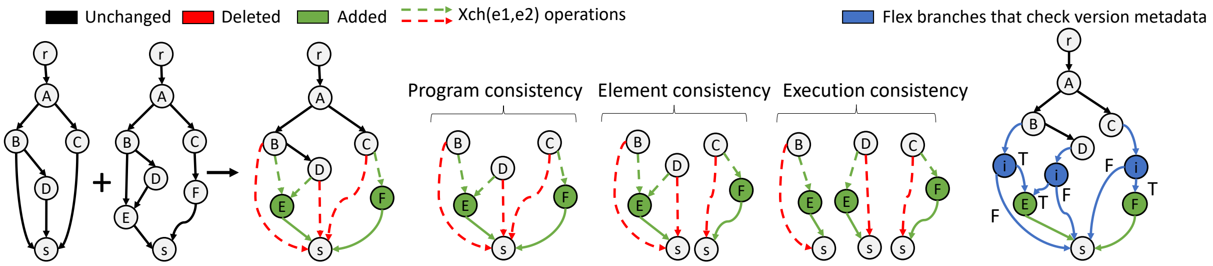
Simultaneously with header extraction, FlexCore uses a parallel set of logic to identify relevant headers to compute the next parsing state. This relies on a similar extraction logic but does not materialize header fields in the extraction array. Rather, it uses the preconfigured 'transition key' to perform a parallelized lookup. It muxes the key through a lookup table that contains all transition rules as compiled from the parser—e.g., IPv4 packets transition to ID=2, and IPv6 to ID=3. A demux combines the lookup results from all rules and computes the next state ID. Parsing continues until it encounters an accept state, at which point the extracted headers are sent to the ingress logic for MA processing.

Reconfiguration primitives. Runtime parser reconfiguration modifies the parser states, extraction rules, and transitions. FlexCore exports AllocState(s), AllocTrans(s1,s2), and AllocEx(r) for allocation of new states, transitions, and extraction rules, respectively. AllocState(s) creates a new PST entry and the respective transition key and header length. AllocTrans(s1, s2) sets up transition rules in the transition matching mux and demux. AllocEx(r) sets up an extraction rule in a parser state that locates a certain offset in the current header and outputs the result to an extraction register. Each primitive has its Dealloc analogue.

Edits to the transition rules with AllocTrans are immediately visible to network traffic, so for multiple changes, FlexCore requires the parser diff or the new parser to be prepared in PST scratch, before they are activated together in a single atomic step. Otherwise, network traffic will be parsed with a mix of old and new parsing logic. In the current hardware, parser changes are only possible with 'program consistency', which, as we will discuss later, requires higher resource headroom to maneuver. This limitation stems from the lack of a 'flex branch' equivalent in the current parser hardware, which is necessary for using the version metadata for transactions. In future hardware generations, this can be incorporated by adding transition version numbers as well as match logic using the versions.

4.6 Summary

We now discuss the two special-case primitives: table modifications and parser changes. ModTbl relies on 'table groups' instead of 'flex branches'. When a ModTbl operation is in progress, it guarantees that each packet is only processed with the old or new version of the table; in this sense, the intermediate states as seen by the packets satisfy 'execution consistency'. However, ModTbl cannot be parallelized with other program modifications, as 'table groups' do not atomically control which version is encountered by packets. Parser changes, on the other hand, satisfy program consistency; but the current hardware doesn't support weaker guarantees, which require 'flex branches'. In the next section, our reconfiguration algorithms primarily focus on changing MA tables and control flow branches, where all three consistency guarantees apply and are achievable at different overheads.



(a) Minimal common supergraph (b) Weaker consistency levels permit finer-grained transactions (c) The use of flex branches

Figure 5: (a) FlexCore constructs a minimum common supergraph between two programs. (b) Weaker consistency guarantees reduce resource requirements for reconfigurations, and allow more intermediate states to be exposed to network traffic. (c) To ensure atomicity, FlexCore inserts 'flex branches' that can branch to the old or new versions depending on the version metadata. These branches are deleted after reconfiguration completes. Nodes A-F represent MA tables or conditional control flow branches. Virtual nodes r and s are added as the sources and sinks of the DAGs, respectively. Virtual nodes i denote flex branches.

5 Runtime Reconfiguration Algorithms

The FlexCore reconfiguration algorithms rely on the partial reconfiguration primitives to transform an existing switch program *prog* to a new one *prog**. We represent each P4 program as a directed acyclic graph (DAG), G for *prog* and G^* for *prog**. Nodes are the MA tables and conditional branches, and edges represent unconditional dependencies (or packet dataflow through the program). Our goal is to compute a *reconfiguration script* [9], a series of graph edit operations to nodes and edges to transform G into G^*. We denote the reconfiguration sequence as $G \rightarrow S_1 \rightarrow \cdots \rightarrow S_n \rightarrow G^*$, where $S_i, in \in [1..n]$ are the intermediate DAGs and each step from S_i to the next state is atomic. Depending on whether (or what types of) intermediate states are allowed to be exposed to network traffic, we propose three levels of consistency guarantees: program consistency, element consistency, and execution consistency, with a decreasing order of strictness. Stronger guarantees are achieved by preparing larger portions of the program diff in scratch memory, requiring that the switch resources must have enough slack for the reconfiguration. Weaker guarantees allow FlexCore to operate within more restricted headroom. Figure 5 includes an illustrative example.

5.1 Program consistency

This is the strongest level of consistency guarantees: no intermediate state is exposed to any packets. The switch program as encountered by network traffic is either G or G^*. This is important for any scenario that requires strong network processing guarantees, where exposing intermediate state would cause operational disruption. For instance, a load balancer or NAT may contain two match/action tables, one for mapping DIP to VIP and another for the reverse direction [25]. Updates to the program (e.g., rehashing) should not take effect until both tables have been reconfigured.

Program consistency. *A sequence* $G \rightarrow S_1 \rightarrow \cdots \rightarrow S_n \rightarrow G^*$ *achieves program consistency if the following property holds for all* $S_i, i \in [1..n]$. *For any element t (node or edge) in* S_i*, if* $t \in G^*$ *and* $t \notin G$*, then* $S_i = G^*$*. Similarly, for any element t in* S_i*, if* $t \in G$ *and* $t \notin G^*$*, then* $S_i = G$.

Put in simpler terms, reconfigured program elements aren't visible to network traffic until all reconfigurations finish: an "all-or-nothing" guarantee. To achieve this, all edits must be prepared in an 'offline' scratch area. They are made visible in an atomic transaction that, from the packets' perspective, changes G to G^* in one single step. Without the partial reconfiguration primitives in FlexCore, one would need to instantiate the entire program *prog** in the scratch while the old one *prog* is still active. Therefore, the switch resources must have enough slack to accommodate the co-existence of both programs—i.e., there must be a headroom of $|G|+|G^*|$. Supposing that $|G| \approx |G^*|$, then the switch resource utilization must be kept to $\leq 50\%$ for runtime changes to be feasible. This is a stringent requirement.

Algorithm. Our new primitives enable FlexCore to only prepare the 'diff' while reusing shared program elements, so the switch only needs to accommodate $|G|$ and newly inserted elements of the size $\Delta \ll |G^*|$. In order to compute the diff, FlexCore merges two DAGs G and G^* into a *minimum common supergraph (MCS)* [9]. An MCS is the union of the input DAGs that minimizes the diff as caused by mismatched elements. In our context, only nodes take up resources and edges are pointer fields in the nodes and do not consume physical resources; so our MCS algorithm primarily extracts node-level diff. Using this MCS, we compute a set of edit operations as our reconfiguration script. INS(v) and DEL(v) inserts and deletes a node, respectively; and INS(e) and DEL(e) operate on edges. A special edge substitution operation XCH(e,e′) is allowed if both edges share the same source node and are of the same type (i.e., both are 'next' pointers or both are true/false jump addresses). In terms of resource overheads, INS(v) reduces and DEL(v) increases switch headroom by |v|, respectively, where |v| is the table size (for MA tables) or action memory size (for conditional branches). Edge operations do not affect resource headroom. Figure 6 shows the algorithm, which colors the MCS: shared elements in black, new elements in green, and deleted elements in red.

```
function ProgramConsistency(prog, prog*)
    // Compute minimum common supergraph
    G ← GetP4DAG(prog); G* ← GetP4DAG(prog*)
    G ← MergeDAGs(G, G*)
    // Compute reconfiguration script
    Script ← ∅
    for node or edge $t \in G$ do
        if $t \in G \wedge t \in G^*$ then
            ColorBlk(t)
        else if $t \in G \wedge t \notin G^*$ then
            ColorRed(t); Script.Add(Del(t))
        else if $t \notin G \wedge t \in G^*$ then
            ColorGrn(t); Script.Add(Ins(t))
    Script.IdentifyEdgeXch(G)
    return Script
```

Figure 6: The program consistency algorithm.

Atomicity. To ensure that intermediate states are not visible until all reconfigurations complete, FlexCore groups the edits in a transaction to achieve atomicity. We use a hardware mechanism that we call a *flex branch*. During the transaction, inserted program elements are guarded by an extra conditional branch that implements a check on special version metadata: 'if (meta.v==0)' branches to the old program elements and 'if (meta.v==1)' to the new. Deletions are also guarded by flex branches instead of being deleted right away. The transaction is committed when FlexCore modifies the version metadata, after which deleted elements can be safely removed.

5.2 Element consistency

A relaxed consistency guarantee, which allows reconfigurations to proceed within more restricted headroom. In program consistency, preparing the diff in scratch area leads to a resource spike of Δ. Therefore, in order to accommodate runtime reconfigurations, the switch utilization must be upper-bounded to leave sufficient headroom Δ.

Element consistency breaks the reconfiguration into several finer-grained transactions that can be performed with lower headroom $\delta \ll \Delta$. This allows FlexCore to drive up switch utilization even further while still preserving the ability to make runtime reconfigurations. Every smaller transaction will add and remove certain program elements, with the goal of releasing some switch resources to accommodate subsequent transactions. Under this guarantee, intermediate states can be exposed to traffic, but only if there is a consistent view as to which program elements have been updated (inserted or deleted). If program elements (nodes or edges) are reachable from each other, they must be updated together. Unreachable edits are partitioned to different transactions as they are independent from the view of network traffic. This property is useful when program updates can be applied incrementally with well-defined semantics. For instance, a firewall that uses independent ACL tables for different types of traffic (e.g., TCP vs. UDP) can be added or removed on a table-by-table basis. A traffic normalizer [1, 22, 39] may apply different security functions for incoming and outgoing traffic—e.g., normalizing TTL fields for incoming packets, but clearing TCP options for outgoing ones.

```
function ElementConsistency(prog, prog*)
    // Compute overall script
    Script ← ProgramConsistency(prog, prog*)
    // Reachability analysis. Optimization using Xch operations.
    for all Xch(u→v, u→v') ∈ Script do
        u.Reachability ← DFS(u, G)
    // Partition script by reachability
    Partitions ← InitPartitionForEachEdit(Script)
    while ∃ reachable partitions p, q do
        MergePartitions(p, q)
    return Partitions
```

Figure 7: The element consistency algorithm.

Element consistency. *For any intermediate state $S_i, i \in [1..n]$, we require the following properties to hold. For any element t in S_i, if $t \in G^*$ and $t \notin G$, then for any other element t' in G^* where $t' \rightsquigarrow^* t$ (i.e., t' can reach t in G^*) or $t \rightsquigarrow^* t'$, we require that $t' \in S_i$. Similarly, for any element t in S_i, if $t \in G$ and $t \notin G^*$, then for any other element t' in G where $t' \rightsquigarrow t$ (t' can reach t in G) or $t \rightsquigarrow t'$, we require that $t' \in S_i$.*

Stated simply, if a new program element is visible in the intermediate state, it should be visible to all packets that traverse this element in the new program, even if they follow different execution paths through the program. A deleted element is no longer visible to packets regardless of their execution paths.

Algorithm. As Figure 7 shows, we first invoke the program consistency algorithm to compute the overall reconfiguration script, and then partition this script into independent, smaller transactions. This relies on a DFS search on **G** to compute whether one edit may affect another. If two edits operate on unreachable regions of the graph, they may proceed independently; otherwise they belong to the same partition. Initially, each edit is in its own partition. Partitions are merged if they are reachable from one another—*p* and *q* are said to be reachable if their edit operations involve elements that are reachable in either direction in **G**. This implies that the algorithm scales quadratically with the number of edit operations.

Although we can perform DFS from all nodes and edges in **G** in polynomial time, in practice we only need to do so from nodes that are involved in an Xch operation. This computes all needed reachability information to merge the partitions, because such nodes are the boundaries between the new and old graphs. Red nodes/edges are reachable from at least one such Xch node by following its red outgoing edges, and similar properties hold for the green color. When no further merges are possible, the algorithm returns a partition of the reconfiguration script.

Atomicity. Each smaller transaction begins with 'meta.v==0'. Flex branches guard intermediate changes or make them visible by changing 'meta.v'. The reconfiguration finishes after all constituent transactions are committed.

```
function ExecutionConsistency(prog, prog*)
    // Compute overall script
    Script ← ProgramConsistency(prog, prog*)
    // Bounded reachability analysis
    for each Xch(u→v, u→v′) ∈ Script do
        Xch.Reachability ← BoundedDFS(u, G)
        InitSubpartition(Xch.Reachability)
    // Order partitions
    for each Xch1, Xch2 ∈ Script do
        if Xch1⇝*Xch2 then
            AddConstraint(Xch1 ≥ Xch2)
        if Xch1⇝Xch2 then
            AddConstraint(Xch1 ≤ Xch2)
        if Xch1 ≤ Xch2 ∧ Xch1 ≥ Xch2 then
            MergeSubpartitions(Xch1, Xch2)
    Subpartitions ← ConstrainedSort(Subpartitions)
    Subpartitions ← DedupEdits(Subpartitions)
    return Subpartitions
```

Figure 8: The execution consistency algorithm.

5.3 Execution consistency

We next consider an even more relaxed guarantee with more finer-grained transactions. Under execution consistency, a new program element may only be visible to some execution paths but not others. Likewise, if an element is deleted from some execution paths, other executions may still use this element until all reconfigurations finish. Such intermediate states are still consistent in that a packet never experiences an execution path that mixes old and new elements. This is the weakest level of consistency that we consider in FlexCore. It is a suitable guarantee for program changes that are in nature non-disruptive—e.g., functions that do not interfere with packet processing decisions, or functions where inaccuracy is tolerable. For instance, a telemetry module that samples or aggregates traffic can be added or removed using execution consistency. The intermediate states merely introduce noise to the monitoring data, but do not break functionality.

Execution consistency. *For any intermediate state $S_i, i \in [1..n]$, any execution path through this program, $p \in S_i$, should satisfy that $p \in G$ or $p \in G^*$.*

This allows reconfigurations to proceed at a per execution path basis. Paths are added to the program as a whole, or they are deleted as a whole. But packets will not encounter partial paths or paths that mix old and new elements.

Algorithm. Figure 8 shows the pseudocode. As before, we perform a reachability analysis from Xch nodes; but unlike in element consistency, the DFS terminates when encountering other Xch nodes or shared (black) nodes. The visited elements form a subpartition for each Xch node. In element consistency, if Xch1 reaches Xch2, they are merged into the same transaction. But execution consistency only requires the merge of certain Xch regions, but not all. If independent reconfigurations of Xch1 and Xch2 do not lead to partial or mixed paths, then their edits can be performed separately.

Specifically, we analyze the ordering relation between all pairs of Xch nodes. If Xch1 can reach Xch2 via a green (new) path p_g, then reconfiguring Xch1 before Xch2 will lead to a situation where the part of p_g in Xch1 is activated but its extension into Xch2 is not, leading to a mixed path. Reconfiguring Xch2 before Xch1, on the other hand, is safe because the changes are not reachable from Xch1. Of course, this reconfiguration will not enable p_g, but this may enable other paths elsewhere so it is a valid plan to be considered. Similarly, if Xch1 reaches Xch2 via a red (old) path p_r, then reconfiguring Xch2 before Xch1 will delete p_r from its end while its earlier part is still in use, resulting in mixed colors. Reconfiguring Xch1 before Xch2, on the other hand, is valid because it simply removes p_r. If Xch1 can reach Xch2 via green and red paths, then the only valid plan is to reconfigure both regions atomically.

This above ordering relation generates a set of constraints across Xch nodes, as well as an ordered set of subpartitions. These subpartitions are finer-grained than the partitions in element consistency, so they enable smaller transactions. One final care must be taken: since subpartitions may be reachable from each other, the bounded DFS may reach shared elements from different Xch nodes. The edit operations in two subpartitions, therefore, may have overlaps. A deduplication step over the subpartitions ensures that a deletion operation is deferred to the last subpartition where the deleted element is used, and that an insertion operation is performed in the first subpartition where the new element occurs. This concludes the execution consistency algorithm, whose complexity is quadratic with regard to the number of Xch nodes.

Atomicity. The use of flex branches makes each subpartition visible to network traffic atomically. The entire transaction finishes when all subpartitions have been reconfigured.

5.4 Summary

The reconfiguration script is then realized by the partial reconfiguration primitives in Section 4—e.g., an operation on v will translate into a table or branch operation depending on v's type. For program consistency, all edits are applied in one single, atomic step, but for element and execution consistency, the (sub)partitions are applied sequentially. This raises another consideration as to the ordering of the transactions in the latter two algorithms to minimize the maximum utilization peak. We perform an exhaustive search over the order. This search terminates when it has identified a feasible order or when it concludes that no such order exists.

6 Limitations and Discussions

Program equivalence. The FlexCore partial reconfiguration primitives and algorithms operate on P4 program elements, relying on the structural differences between two P4 programs. It currently doesn't analyze whether structurally different programs may have the same semantics [10, 13], which is an interesting avenue for future work.

Stateful packet processing. FlexCore currently does not support stateful switch programs. The P4 standard defines persistent state as an "extern" feature that is up to the individual architectures to implement (e.g., registers in PSA). Partial reconfiguration of stateful features raises additional questions as to how network state should be ported to the new program, e.g., with programmer-supplied state transformation functions, much like in SDN software controller upgrades [32].

Resource headroom. The FlexCore algorithms require that the switches have sufficient resource slack, but there could be scenarios where even the weakest consistency would require more resource headroom than available. To address this, one could relax execution consistency even further to capture which types of 'mixed' executions are still semantically meaningful; alternatively, one could also migrate certain resources to other devices to make room for the reconfiguration.

Other architectures. The FlexCore primitives target P4 program changes, so they are in principle architecture-independent. The dRMT variant that FlexCore uses makes runtime reconfiguration particularly natural, but most P4 targets have some degree of runtime flexibility. The RMT architecture, for instance, may be augmented with the ability to reconfigure each stage independently. Software switch targets (e.g., for the host or NIC) expose even more runtime flexibility than switch ASICs. Although the original dRMT project [11] didn't provide an ASIC implementation, we believe that our indirection structures are compatible with its outlined design.

Other languages. FlexCore's reconfiguration primitives target P4 programs, but for other languages (e.g., NPL [2], PoF [36]), one should be able to develop analogous reconfiguration primitives based upon their respective language features. The property of runtime programmability is not tied to a specific language.

7 Implementation

We have implemented FlexCore in several components. The reconfiguration primitives are implemented by manipulating the hardware ASIC control registers via the PCIe interconnect from the control plane. The indirection structures are implemented in the Spectrum-2 silicon design, and FlexCore is the first effort to leverage them for runtime, partial reconfiguration. Our compiler uses p4c [4] as the frontend; it implements incremental compilation of P4 program elements, generating an individual binary image for each component, instead of outputting a monolithic binary for the entire program. The consistency algorithms are implemented at the control plane.

The hardware cost to enable runtime programmability comes from the use of indirection structures, including the PDT and PST. The PDT supports full reconfigurability at all consistency levels, but the PST only supports program consistency. We estimate the cost of the current PDT and the cost for making the PST fully reconfigurable at runtime.

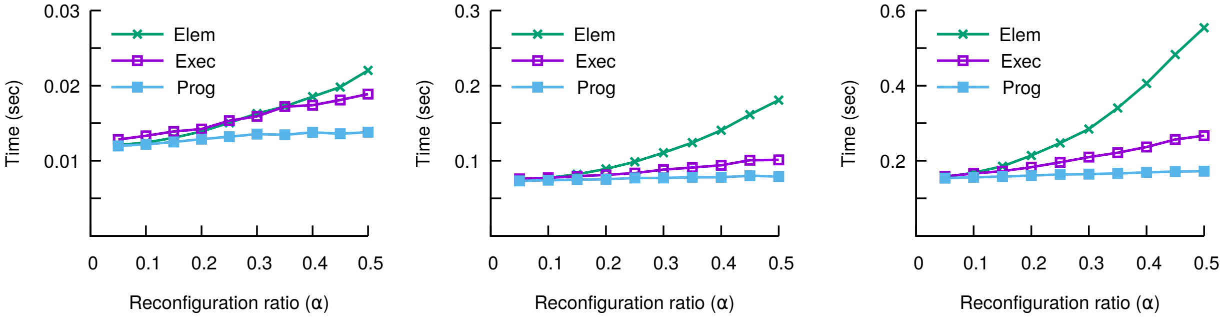
Each MA processor has a local PDT, which holds roughly 1k entries—i.e., the largest P4 program it supports should have no more than $\sim$ 1k MA tables and conditional branches. The ASIC supports up to 128 MA processors overall. Recall the PDT format as shown in Figure 2: each entry contains a) a description of the match key, b) entry type (SRAM/TCAM/Actions), c) resource pointer, and d) next table/branch pointer. In the worst-case scenario, each MA table has a different key, resulting in 1k distinct keys that the switch needs to support; this requires 10 bits to represent each distinct key in the PDT entry. The entry type field distinguishes between three types, requiring 2 bits. The resource pointer requires 20 bits, which is able to index one million distinct memory lines for SRAM/TCAM/Actions, roughly 20MB in size (the 'main database'). The next table/branch pointer requires 10 bits to index another PDT entry as the next hop. Overall, each PDT entry requires 42 bits, each PDT table consumes 5.25kB for 1k entries, and across 128 PDT tables the hardware overhead is 0.67MB, or 3.3% of the main database. The flex branch mechanism is implemented using existing ALUs, so it doesn't require dedicated hardware. For the PST, the Spectrum-2 parser hardware only supports runtime reconfiguration at program consistency level. This does not contain the 'flex branch' equivalent and the 'version' support for transition rules, which would be necessary for other consistency levels. We estimate the overhead of these additional structures to be under 1% of the main database.

8 Evaluation

We present a comprehensive evaluation of FlexCore, by applying our design to a 12.8 Tbps hardware ASIC and also a software simulator (a fork of bmv2) that has been integrated with the same reconfiguration primitives. To evaluate scalability, we have used a set of synthetic and real-world P4 programs. To synthesize the P4 corpus, our tool takes a specified program size and generates a random control flow graph. For real-world programs, we have used switch.p4, NetCache [19], and NetHCF [23], which represent large, medium, and small programs, respectively. The program edits are also generated randomly, which may mutate, insert, remove, or swap program elements. The edits are controlled by a parameter α, the reconfiguration ratio. If a program has 100 program elements, and a reconfiguration adds, removes, or exchanges 10 of them, we say that $\alpha = 10\%$. To evaluate realistic reconfiguration scenarios, we perform case studies using switch-based multicast, telemetry, attack mitigation, and tenant-specific extensions, on hardware and software platforms.

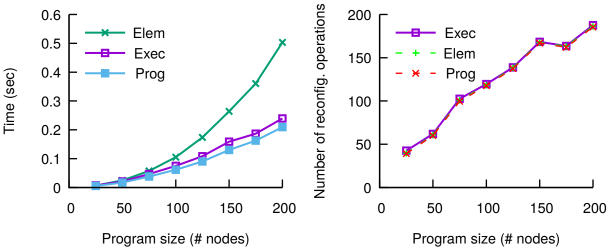
8.1 Reconfiguration primitives

We start by measuring the number of hardware operations that each reconfiguration primitive involves. These primitives are invoked by the control plane, and they modify a memory-mapped region of the PCIe device (i.e., the data plane). The PCIe bus sustains a peak throughput of $\sim$1 million operations per second. The control plane, however, is bottlenecked by the software speed; each operation took several milliseconds to



(a) NetHCF (V=43, E=58). (b) NetCache (V=109, E=129). (c) switch.p4 (V=168, E=242).

Figure 9: Scalability of FlexCore on three real-world programs. V: number of nodes, E: number of edges.



(a) Turnaround time (b) Number of primitives

Figure 10: The FlexCore algorithms scale well.

complete with software overhead. Table 1 shows the number of hardware register DWORD writes for each reconfiguration primitive. As we can see, table operations are the most heavy-weight, control flow branch operations follow, then parser operations, and finally, edge edits complete within one write and are atomic. Deallocations have the same number of operations as their allocation analogues.

Primitive	RegAccess	Primitive	RegAccess
AllocTbl	112	GroupTbl	112
AllocCond	43	AllocState	22
AllocTrans	5	AllocEx	3
SetPtr	1	SetCondPtr	2

Table 1: The number of hardware register accesses (in DWORDS) for each reconfiguration primitive. Allocation and deallocation primitives as measured only operate on metadata (i.e., PDT and PST), not including SRAM/TCAM/action memory resources. The cost for the latter varies depending on the allocation/deallocation sizes.

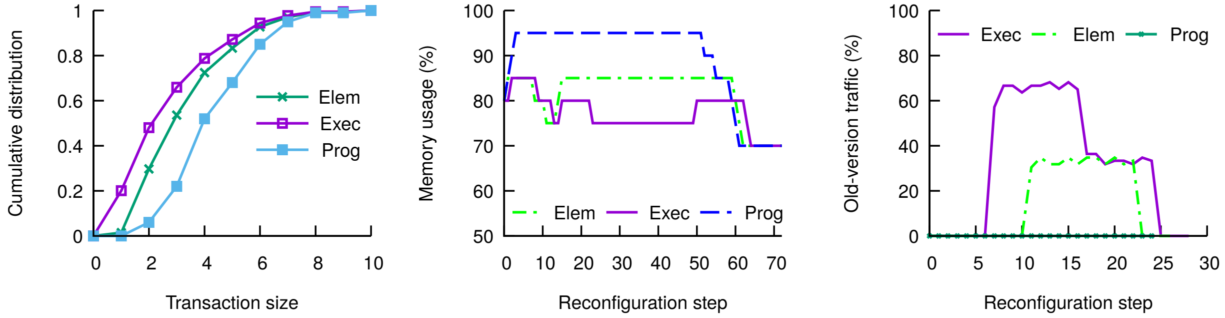
8.2 Consistency algorithms

Synthesized programs. We evaluate the scalability of FlexCore in generating reconfiguration scripts for programs of different sizes. We generated 100 programs of each size (800 in total), and set $\alpha = 40\%$ for FlexCore to generate reconfiguration scripts. Figure 10a shows the results. As expected, program consistency took the least amount of time, as the only analysis is on the program diff; all edits are then grouped as a whole. The turnaround time for element consistency grows roughly quadratically with regard to the program size (more strictly, to the size of the diff, which is fixed to 40% of the program size). Execution consistency algorithm lies in between, as it scales with the number of Xch nodes, which is smaller than the program diff. Overall, FlexCore generated reconfiguration scripts for all programs within one second.

Next, we measure the number of invocations of the partial reconfiguration primitives as well as the version metadata operations. As shown in Figure 10b, the numbers of operations for different consistency levels are roughly the same. This is because the number of reconfiguration operations are the same regardless of the consistency level. But the number of transactions increases for weaker guarantees due to the extra version metadata operations.

Real-world programs. We then tested FlexCore on three real-world programs of different sizes, and further varied the reconfiguration ratio α from 5% to 50%. As Figure 9 shows, the FlexCore turnaround time is longer for larger programs and higher reconfiguration ratios. But the overall takeaways are similar as before: FlexCore algorithms scale well for computing reconfiguration scripts. In the Appendix, we further include scalability results for ordering the transactions.

Consistency levels. Figure 11a shows the CDF of the transaction sizes under different consistency guarantees for the synthetic programs with different sizes and α. Under stronger consistency guarantees, the transactions have larger sizes (we fix all tables to the same size). We also measure the headroom requirements. Figure 11b visualizes the step-by-step reconfiguration for one such program: program consistency requires a large peak headroom, but weaker guarantees have less stringent requirements. All consistency levels eventually converge to the same utilization level after reconfiguration completes. Figure 11c tests another program in the software simulator, which plots the percentage of traffic that experiences the old program after the first update is enabled during the reconfiguration under different consistency levels. As we can see, program consistency does not expose any intermediate state, but weaker guarantees lead to more traffic that is



(a) Transaction sizes (b) Resource headroom (c) Consistency

Figure 11: (a) Weaker consistency guarantees lead to smaller transaction sizes; they require lower resource headroom, but more traffic will encounter the old program during reconfiguration. In (b), the initial switch utilization when the reconfiguration starts is 80%. In (c), we use a program whose control flow graph is presented in Figure 5; Y-axis shows the traffic ratio processed by the old program after the first reconfiguration transaction is committed at the 6th step.

processed using the old program during the reconfiguration.

8.3 Case study: Accelerated multicast

Just-in-time optimization. Next, we present a case study using the hardware ASIC, where program elements are injected to the switch pipeline at runtime to accelerate multicast applications. Initially, the switch is configured with a baseline program without any multicast optimizations, and it connects one ZeroMQ unicast sender and multiple receivers. Just before the ZeroMQ application starts, we initiate a partial reconfiguration to extend the switch program with Elmo [34], a switch-based multicast function. Elmo performs source-routed multicast in hardware with customized protocol headers to improve scalability. After ZeroMQ finishes, another reconfiguration removes the Elmo components from the switch pipeline. These changes are performed under program consistency. Each reconfiguration took less than 0.5 s to complete in control plane software.

Just-in-time telemetry. Just before removing Elmo, we inject a co-located telemetry application to observe the effect of Elmo removal, by monitoring the average pipeline latency of randomly sampled packets. This telemetry application is unloaded after the removal of Elmo.

Reconfiguration. Figure 12a plots the throughput of a third background iPerf application during the entire reconfiguration. As we can see, the reconfigurations did not cause any service interruption, as the iPerf throughput was stable throughout the experiment; the switch drop counters also showed no packet loss. Figure 12b plots the additional resource usage in terms of PDT memory, PST memory, and table entries during the runtime reconfigurations. The insertion of Elmo caused a resource usage increase, as did the insertion of the telemetry application. But in both cases, the extra resource overheads for PDT and PST are under 200 bytes. Table rules for multicast and telemetry, on the other hand, are the dominant overheads. All resources are released after the program modules are removed from the pipeline.

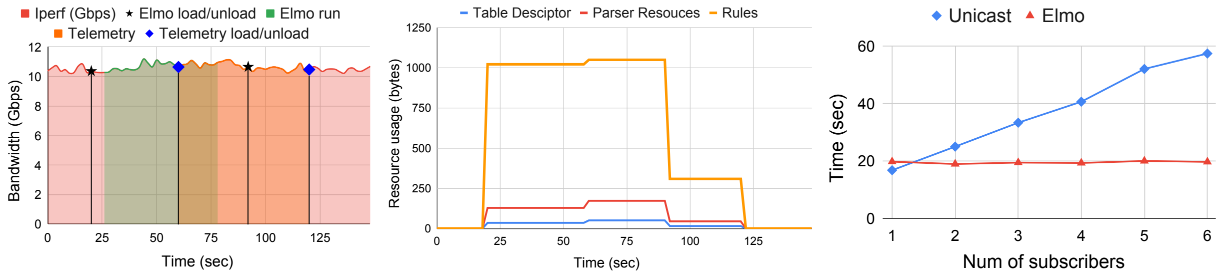
Performance. Figure 12c shows that the injection of Elmo improves multicast scalability, where we measure the completion time to send 200 k ZeroMQ messages. Before injecting Elmo, a preceding ZeroMQ run via unicast took up to nearly 60 seconds for six receivers; and the completion time grows roughly linearly with the number of receivers. After injecting Elmo, the switch-based multicast scales independently of the number of receivers, finishing at roughly 20 seconds across all tested configurations. The injected telemetry application detected that the pipeline latency experienced a 20 ns decrease after Elmo was removed from the pipeline.

8.4 Case study: Dynamic telemetry upgrade

In-place application upgrade. We perform another ASIC-based case study under execution consistency. The operator modifies the telemetry application discussed earlier to use different flow keys. Initially, the application uses the IPv4 five tuple as the match key and is configured with 30 k entries. Packets of interest are sampled to software for telemetry processing. The operator issues a reconfiguration to modify the match key to the source and destination addresses instead, using the MODTBL primitive. This modification also reduces the resource usage, as entries become smaller.

Reconfiguration. Figure 13 plots the performance of a background iPerf application, which shows stable throughput. The blue area further shows an additional IPv4 test trace that we generated to specifically trigger the telemetry table. The switch counters indicated zero packet loss for iPerf and the IPv4 test traffic. We have set the migration rate to be 3k entries per second, so the modified table was populated in ~10s. The PDT operations at the control plane software took 400 ms.

Utilization. Figure 14 shows the intermediate program sizes using MODTBL, and compares it with the baseline that inserts the new table and then deletes the old. The baseline incurs a resource utilization spike, which occurs when both tables are co-resident in the switch. As the old table is gradually deallocated, the resource usage drops to the size of the new



(a) Hitless reconfigurations. (b) Resource overheads. (c) Performance improvements.

Figure 12: FlexCore inserts Elmo, a switch-based multicast program, just-in-time to accelerate ZeroMQ performance. It also inserts a telemetry application to observe the effect of the removal of Elmo.

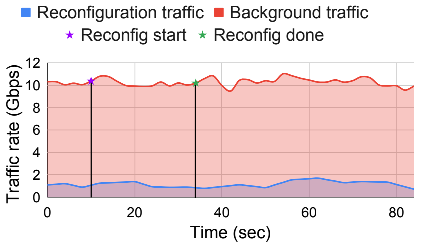


Figure 13: Hitless table modification.

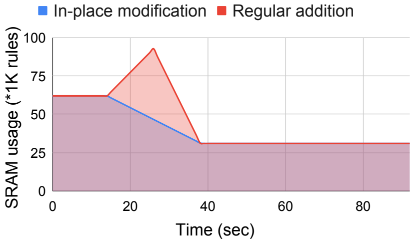


Figure 14: Resource usage with in-place table modification.

table. The final SRAM usage shows the resource reduction in changing the flow keys. In contrast, our in-place modification consistently reduces resource usage reduction right from the beginning, until resource usage approaches its final state.

8.5 Simulator case studies

We have performed two case studies in the software simulator with element consistency. The appendix includes concrete results. We highlight here that all reconfigurations were effective and free of interruptions.

Real-time attack mitigation. This case study injects a *TCP normalizing firewall* [1] and a *covert channel defense* [39] upon attack detection. The normalizer pads all TTL values to avoid inconsistent views at host IDS [22], and the covert channel defense clears TCP reserved bits to avoid data leakage [39]. The normalizer inspects incoming traffic, but the covert channel defense inspects outgoing traffic.

Tenant-specific network extensions. VM migration triggers FlexCore to carve out the tenant's ACL functions from the original switch and inject it to the destination switch.

9 Related work

Programmable networks. Network programmability has been a longstanding goal in the community—starting with 'active networks' [6, 33, 37], each step in this direction has led to significant innovation in the networking ecosystem. FlexCore takes the next step to enable *runtime programmable switches*. Recent projects P4Visor [42] and Hyper4 [16] also use DAG merging algorithms on P4 programs, but our focus is on partial program reconfiguration.

Consistent updates. Network updates are common to datacenters [20, 28], and ensuring the absence of service interrupt is a key goal [24]. Researchers have considered live migration of BPG sessions and virtual routers [21, 38], and per-packet and per-flow consistency guarantees for OpenFlow network updates [29, 30]. Our work tackles the problem of achieving reliable switch program updates at runtime, and proposes a new set of consistency guarantees.

OS+network specialization. The vision of FlexCore is inspired by prior work in OS and network specialization. SPIN [7] is an OS that allows applications to inject safe and dynamic extensions to the kernel. Exokernel [14] enables applications to specialize OS functions at user level. ESwitch [26] specializes OpenFlow software data planes to achieve higher performance for a given workload. Our work aims to enable similar goals for programmable switches.

10 Conclusion

FlexCore argues that runtime programmability should be a first-order goal in future networks, allowing functions to be added or removed dynamically. FlexCore contributes design considerations on switch architectures, partial reconfiguration primitives, reconfiguration algorithms and consistency guarantees. Our evaluation shows that the FlexCore reconfiguration algorithms are scalable, and that runtime reconfigurations are beneficial and free of disruption.

Acknowledgments: We thank our shepherd Laurent Vanbever and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by CNS-1801884, CNS-1942219, CNS-2016727, CNS-2106388, and CNS-2106751.

References

[1] Cisco: TCP normalization. https://www.cisco.com/c/en/us/td/docs/security/asa/asa96/configuration/firewall/asa-96-firewall-config/conns-connlimits.html.

[2] nplang. https://github.com/nplang.

[3] The P4 language repositories. https://github.com/p4lang.

[4] The p4c compiler. https://github.com/p4lang/p4c.

[5] The P4Runtime Specification. https://github.com/p4lang/p4runtime.

[6] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Panka J. Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network*, 12(3):29–36, 1998.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *SOSP*, 1995.

[8] 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. *ACM SIGCOMM CCR*, 43(4):99–110, 2013.

[9] H. Bunke, X. Jiang, and A. Kandel. On the minimum common supergraph of two graphs. *Computing*, 65(1):13–26, 2020.

[10] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *Proc. NSDI*, 2021.

[11] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated programmable switching. In *Proc. SIGCOMM*, 2017.

[12] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. NSDI*, 2018.

[13] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *Proc. NSDI*, 2019.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*, 1995.

[15] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[16] David Hancock and Jacobus van der Merwe. HyPer4: Using P4 to virtualize the programmable data plane. In *Proc. CoNEXT*, 2016.

[17] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. NSDI*, 2018.

[19] 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 *Proc. SOSP*, 2017.

[20] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proc. HotNets*, 2013.

[21] Eric Keller, Jennifer Rexford, and Jacobus E van der Merwe. Seamless BGP migration with router grafting. In *Proc. NSDI*, 2010.

[22] Christian Kreibich, Mark Handley, and V Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security*, 2001.

[23] Guanyu Li, Menghao Zhang, Chang Liu, Xiao Kong, Ang Chen, Guofei Gu, and Haixin Duan. NetHCF: Enabling line-rate and adaptive spoofed IP traffic filtering. In *Proc. ICNP*, 2019.

[24] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: updating data center networks with zero loss. In *Proc. SIGCOMM*, 2013.

[25] 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 *Proc. SIGCOMM*, 2017.

[26] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance OpenFlow software switching. In *Proc. SIGCOMM*, 2016.

[27] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[28] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. Decentralized consistent updates in SDN. In *Proc.*

SOSR, 2017.

[29] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[30] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. HotNets*, 2011.

[31] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4), 1984.

[32] Karla Saur, Joseph Collard, Nate Foster, Arjun Guha, Laurent Vanbever, and Michael Hicks. Safe and flexible controller upgrades for SDNs. In *Proc. SOSR*, 2016.

[33] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets for active networks. In *Proc. OpenArch*, 1999.

[34] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source routed multicast for public clouds. In *Proc. SIGCOMM*, 2019.

[35] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. USENIX ATC*, 2018.

[36] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proc. HotSDN*, 2013.

[37] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM CCR*, 26(2):5–18, 1996.

[38] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus Van Der Merwe, and Jennifer Rexford. Virtual routers on the move: live router migration as a network-management primitive. *ACM SIGCOMM CCR*, 38(4):231–242, 2008.

[39] Jiarong Xing, Qiao Kang, and Ang Chen. Netwarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[40] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[41] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proc. SIGCOMM*, 2020.

[42] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proc. CoNEXT*, 2018.

11 Appendix

11.1 Case study: Real-time attack mitigation

In this case study, we present how FlexCore facilitates real-time attack mitigation by reconfiguring two defense functions to the software switch simulator.

Traffic normalizers [22] are firewall utilities that prevent inconsistent views between network IDS and end hosts. As an example, some packets may be seen by the IDS, but their TTL values are crafted in such a way that they are dropped soon after the network IDS and do not trigger processing at end hosts. This leads to vulnerabilities [22]. A normalizer firewall can pad TTL values to ensure that the IDS and the hosts always have the same view.

Covert channels [39] leak secret data by repurposing packet header fields as data carrier. For instance, an attacker that compromise a server that hosts confidential data may leak the secret by padding them into the TCP reserved bits of network traffic. A defense needs to clear such optional header fields to prevent leakage.

Real-time attack mitigation. In our case study, we inject a *TCP normalizing firewall* [1] and a *covert channel defense* [39] upon attack detection. Since these two defenses are independent, they can be reconfigured under element consistency. Figure 15 shows the workflow for the reconfiguration. After each defense is deployed, the attack traffic can be recognized and blocked; its throughput drops to zero. The normal traffic does not experience any loss or interruption during the reconfiguration.

11.2 Case study: Tenant-specific network extensions

In this case study, we focus on multi-tenant datacenters where each tenant can inject her own network extensions to the switch. Upon VM migration, the switch modules are carved out from the source and grafted to the new destination switch.

Program grafting in VM migration. In this scenario, a tenant has her ACL module injected to the ToR switch, and her VM migration will bring this module to a different destination rack. This is achieved by carving out the ACL components and grafting them to the destination switch using partial reconfigurations. Figure 16 shows the traffic rate of the tenant's traffic and the background traffic. The migration is achieved in several steps. It first inserts the ACL module to the new switch, and then routes traffic to the new switch by updating the routing rules of upstream switches. Finally, it removes the ACL module in the old switch. As we can see, the migration does not cause throughput drops of the background traffic during the reconfiguration, and the tenant's traffic is migrated to the new switch without service interruption.

11.3 Evaluation: Ordering the transactions

For element and execution consistency, the reconfiguration proceeds in multiple steps. So FlexCore additionally performs an exhaustive search to identify a feasible sequence under the current headroom. The problem can be stated as: given

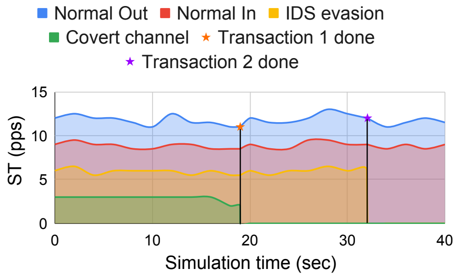


Figure 15: Simulation traffic rates (ST) when reconfiguring the switch using element consistency to inject real-time network defenses.

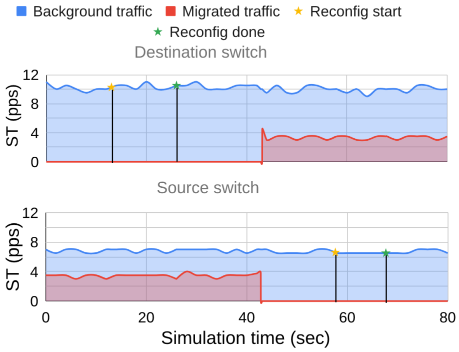
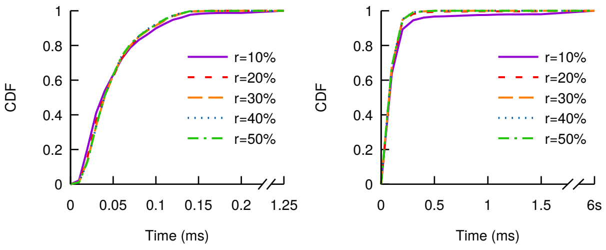


Figure 16: Simulation traffic rates (ST) during a reconfiguration triggered by VM migration, which carves out an tenant-specific ACL module from the source switch and grafts it to the destination switch.



(a) Element consistency (b) Execution consistency

Figure 17: The turnaround time for finding a feasible sequence for the transactions, under element and execution consistency.

a set of transactions $tx_1, tx_2, \cdots, tx_k$, find a feasible sequence that fits into the current resource headroom, or conclude that such a sequence doesn't exist. For element consistency, the transactions do not have a hard constraint as to their order. The search only focuses on optimizing for resource headroom. Execution consistency has hard constraints as to which transactions should be ordered before others. The search also encodes such constraints as induced from the XCH nodes.

In Section 8.2, we have evaluated the scalability of the algorithm in generating reconfiguration scripts. Here, we further evaluate the turnaround time for identifying a feasible sequence of transactions that can be applied within the available headroom. We have used an α ranging from 10% to 50% on the synthesized programs, and tried different switch resource headrooms as denoted by r (ranging from 10% to 50%).

Figure 17a shows the results for element consistency. We can see that FlexCore finishes within 0.2ms for all programs across different headrooms except for r=10%. With 10% headroom, the switch has very small slack for the reconfiguration, so it takes more time to search for a feasible plan or determine that no solution exists.

Figure 17b shows the results for execution consistency, where the turnaround time is higher because of two reasons. First, execution consistency needs to merge and deduplicate the subpartitions following their constraints. Second, execution consistency could generate more candidate solutions, resulting in longer searching time. However, the algorithm can still complete within 1.5ms for 98% programs and within 6s for all programs.