ClickINC: In-network Computing as a Service in Heterogeneous Programmable Data-center Networks

Wenquan Xu[†], Zijian Zhang[†], Yong Feng[†], Haoyu Song[⋆], Zhikang Chen[†], Wenfei Wu[§], Guyue Liu[‡], Yinchao Zhang[†], Shuxin Liu[†], Zerui Tian[†], Bin Liu[†*]
[†]Tsinghua University, [⋆]Futurewei, [§]Peking University, [‡]New York University Shanghai

ABSTRACT

In-Network Computing (INC) has found many applications for performance boosts or cost reduction. However, given heterogeneous devices, diverse applications, and multi-path network typologies, it is cumbersome and error-prone for application developers to effectively utilize the available network resources and gain predictable benefits without impeding normal network functions. Previous work is oriented to network operators more than application developers. We develop ClickINC to streamline the INC programming and deployment using a unified and automated workflow. ClickINC provides INC developers a modular programming abstractions, without concerning to the states of the devices and the network topology. We describe the ClickINC framework, model, language, workflow, and corresponding algorithms. Experiments on both an emulator and a prototype system demonstrate its feasibility and benefits.

CCS CONCEPTS

• **Networks** → **In-network processing**; **Programmable networks**; *Programming interfaces.*

KEYWORDS

In-Network Computing, Programmable networks, Programming abstraction, Program compilation, Program placement

ACM Reference Format:
Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, Bin Liu . 2023. ClickINC: In-network Computing as a Service in Heterogeneous Programmable Data-center Networks. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3603269.3604835

1 INTRODUCTION

Defying the conventional wisdom, network is no longer considered as dumb pipe but also a computation-facilitating infrastructure which can help boost application performance (e.g., latency and throughput) or reduce system cost (e.g., power and engaged servers). Such a paradigm shift, dubbed as *In-Network Computing (INC)*, has benefited many applications (e.g., key-value store [17, 22], machine learning (ML) aggregation [20, 29, 30], consensus [5, 6], coordination [16], and streaming [15]). These applications are typically enabled by the programmable switches (e.g., Tofino [12]) which however is limited by hardware capability and capacity [19], arising a trend to extend on *heterogeneous programmable network devices* [2, 4, 13, 35] (e.g., Tiara [35] achieves a layer-4 load balancer), where the switch is used to perform throughput-intensive task (packet encap/decap) and FPGA is used for memory-intensive task (physical server selection).

While this momentum is inspiring, a closer look reveals a less optimistic reality: the adoption of INC is currently limited to network operators and has not yet to be embraced by application developers, which hinders the development of new applications and their large-scale deployment. The fundamental reason, we believe, is the lack of a high-level programming framework that can abstract away the complexities associated with issues such as device heterogeneity, network topology, and function mapping. Early efforts [8, 11] attempted to improve the programming abstraction by hiding hardware details. Although this is a valuable first step, there are still three major barriers. To see why, consider the state-of-the-art framework Lyra [8].

Limited to low-level abstractions. Lyra progresses from low-level and chip-specific languages (e.g., P4 [28] and NPL [25]) to a more general and cross-platform language. However, it still requires programmers to handle low-level details such as packet header processing and network protocol handling, and is limited to basic statements (e.g., if-else), rather than more advanced features (e.g., for-loop). Crucial features such as network transparency, cross-device correctness, and program isolation are missing and need to be implemented by INC programmers. These burdens discourage application developers from adopting the INC programming paradigm.

Limited to a small-scale deployment. Lyra can run a data plane program on multiple heterogeneous ASICs in a distributed way (e.g., load balancer [9], in-band network telemetry [7]). It achieves this by encoding the logic and different resource constraints into a satisfiability modulo theories (SMT) problem, and using an SMT solver (e.g., Z3 [24] and cvc5 [1]) to find the deployment strategy. However, this approach is prohibitively slow (e.g., Z3 takes 30+ minutes to allocate ML Aggregation program on only 5 Tofino devices). Furthermore, it can only find a *feasible* deployment without considering resource utilization, thus limiting it to running a small number of applications with fixed resources.

Limited to a single user. Lyra, along with other prior work [11, 32], is designed for network operators who have complete control over all network devices and run a *monolithic* program in the target

*Bin Liu, Wenfei Wu, and Guyue Liu are the corresponding authors

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604835

network. In case of any changes, the entire program must be recompiled from scratch and reinstalled on affected devices, leading to inefficiencies in terms of compilation and installation time. Furthermore, this approach is unsuitable for running programs from multiple users, as coordination between users is necessary and traffic from different users must be interrupted for every change.

Given these issues, we argue the need of a new framework that offers high-level abstractions for writing applications, and automatically handles low-level system concerns such as placement, cross-device communication, resource isolation, fault tolerance, and more. With such a framework, developers would be able to offload routine tasks to the framework and focus on the critical logic of their applications.

In this paper, we present ClickINC, a framework for INC application developers (referred to as "users") to develop, deploy, and manage programs on heterogeneous programmable network devices in data centers. At a high level, ClickINC offers the following capabilities: (i) ClickINC allows users to develop applications in a high-level, Python-style language; (ii) ClickINC's compiler frontend compiles each user's program into a platform-independent intermediate representation (IR) program and determines the optimal placement strategy across the network; then the backend translates IR programs into chip-specific programs and launch them on the target network; (iii) at runtime, ClickINC isolates resources for different users and allows for dynamically adding and removing programs. Compared to prior work, ClickINC makes the following three notable contributions:

1) Modular programming abstractions. ClickINC encapsulates common INC functionality into *modules* such as various sketches, hash functions, providing users with a library. Users can work at a higher level of abstraction and use a simple Python-style syntax to import modules they need to write applications. This design eliminates the need for users to worry about the low-level details (e.g., packet-level processing and implementation of data structures), reducing the amount of code (at least 10 times lower), and enabling them to reuse code across multiple projects. The comparison between ClickINC and other operator-oriented languages such as Lyra can be drawn to that of Python and C/C++. While C/C++ is fast and efficient, it is suited for low-level system development, whereas Python is easier to learn and use, better suited for application development.

2) Scalable placement algorithm. Efficiently placing programs on a network of heterogeneous programmable devices is challenging. In addition to different hardware features and resource constraints considered by prior work [8, 32], we take into account three new factors: (i) the network may consist of multiple paths for an application; (ii) the interaction between program segments distributed across multiple devices may result in extra overhead; (iii) users may add or remove applications dynamically, but the placement recomputing from scratch should be avoided. We propose a program partition theory and based on it, we develop a dynamic programming (DP) algorithm to solve the placement problem in polynomial time and scale up to ~1,000 switches.

3) Incremental program compilation. To effectively support the multi-user scenario where each user dynamically adds or removes a program, ClickINC provides the incremental program compilation feature (not runtime). Unlike prior work, we need to consider not only the programs run by the operator for routine packet processing and forwarding, but also programs from users for high-level applications such as key-value stores. Our key idea is to maintain the operator's program as the *base program*. By applying an annotation-based method, multiple user programs can be correctly identified and incrementally integrated to or stripped from the base program. When synthesizing the base program and multiple user programs, ClickINC isolates both programs' states and control flows, ensuring each user's traffic is processed by the corresponding program.

We build an end-to-end system and implement three common INC applications. Our evaluations show that ClickINC is 10X better than the state-of-the-art programming languages in lines of code, as well as near 1000X faster than SMT solver for program placement, and 50%-75% less traffic is affected to deploy a new program.

2 BACKGROUND AND MOTIVATION

We provide the background of INC in data centers, and then discuss the pain points of application developers to motivate the need of a new INC framework.

2.1 INC in Data Centers

INC Applications. We use two common applications, key-value store and ML gradient aggregation, to illustrate the process and benefits of adopting the INC paradigm.

(1) Key-Value Store (KVS). Traditional KVS nodes are inefficient in handling skewed, dynamic workloads due to limited server performance. With the programmable network devices, the in-network KVS, typified by NetCache [17], can accelerate KVS with 3-10x throughput improvement and lower latency. To deploy KVS on a capability-limited switch, the advanced data structure on servers needs to be replaced with a hash based key-value table, a hit counter, and a combination of Count-Min Sketch and Bloom Filter, to support cache read/write and statistics of queries for cache update.

(2) ML Gradient Aggregation (MLAgg). Traditional distributed MLAgg relies on a parameter server (PS) or allreduce, which have performance bottleneck on servers. To accelerate it, in-network aggregation [20, 30] maintains a stateful structure called aggregator array on switch to aggregate gradients from different workers, which greatly improves the aggregation throughput. Packets are addressed to aggregators by their job id and sequence number, and an aggregator sums up the data from all workers and returns the results back to workers. Due to the limited switch resource and capability, the data type conversion from floating-point to integer may be needed, and several other data structures are used to ensure correct aggregation.

Heterogeneous Programmable Devices. The heterogeneous devices in DCN (e.g., switches, FPGA, and smartNIC) can be roughly classified as pipeline or multi-core devices. The former has a number of stages with each running a piece of the program and can provide a throughput guarantee; the latter has multiple cores working in parallel and can support more complex functions. It may be infeasible to deploy an INC application on a single device or the same type of devices due to resource and feature constraints. For example, to aggregate the ML parameter with 64 integers in a packet, at least two Tofino switches are needed due to the limited on-switch memory. Further, if the parameters have large sparsity,

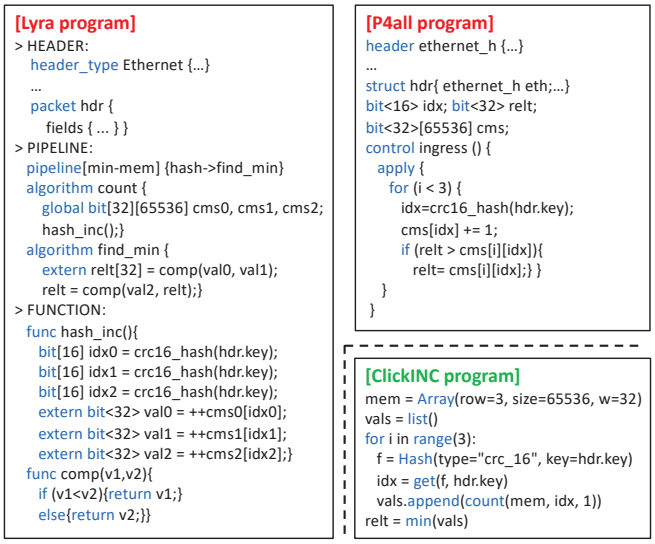


Figure 1: Language comparison for count-min sketch.

the sparsity detection and elimination function cannot be run on the programmable switches and require another type of device (e.g., SmartNIC or FPGA).

2.2 Pain Points for Application Developers

We discussed the three problems of INC program development in §1, and we further illustrate these problems using examples of state-of-the-art solutions.

Low-level architecture and network details. The existing INC programming is not friendly to application developers. Fig. 1 shows an example of the implementation of an in-network count-min sketch in Lyra, P4all, and ClickINC. Lyra and P4all are network operator-oriented and preserve device specific concepts such as pipeline, bit width, and CRC. In contrast, ClickINC's basic programming elements are `for` loop and Array, which are organized following a Python-like high-level language syntax. The ClickINC program is easier to learn and write, and needs fewer lines of code.

Limited number of devices and applications. It is error-prone to place a program spanning multiple heterogeneous devices for applications with multi-path traffic. For example, to deploy an MLAgg on a fat-tree network, due to complex network topology and unbalanced resources, manual placement may cause: (1) some paths with an inadequate resource cannot be covered by MLAgg, so a lot of traffic cannot be aggregated; (2) cross-device interaction overhead as well as extra resource usage is high due to improper partition. SMT solver (used by prior work [8]) should have been a good tool to deal with this problem as the placement task can be modeled in SMT. However, such solvers need to traverse the entire solution space which has an exponential time complexity on both the number of instructions in the program and the number of devices.

Lack of user isolation. Existing network devices do not provide isolation between user programs. These devices were designed for a single-party operator, and thus do not have mechanisms (e.g., resource virtualization) to support multiple programs from different users. For example, if two users deploy the same Count-Min Sketch program (Fig. 1) as two instances, with naïve program splicing, both users' traffic will be monitored at the same memory region `vals.append()`. This may impact the accuracy of measurement and expose sensitive data (can be read by each other in `relt`).

3 CLICKINC OVERVIEW

Our goal in designing ClickINC is to provide a framework for developing INC applications and automatically deploying them on heterogeneous programmable network devices in data centers. In practice, we also want (i) the developing environment to be friendly to developers, minimizing the effort required to apply the new INC programming paradigm, and (ii) the deployment to be compatible with existing INC deployments controlled by network operators.

3.1 Key Ideas

We first discuss the main ideas that enable ClickINC to tackle the three pain points discussed in §2.2 while meeting the above two practical requirements.

1) A high-level, Python-style language with built-in modules. We observed that the key obstacle to using the existing languages is it requires extensive architecture and network specific details. Inspired by the success of the high-level language Python, ClickINC provides users with Python-style syntax elements. Meanwhile, ClickINC encapsulates widely-used basic data structures (e.g., key-value matching table) and functions (e.g., hash) as ClickINC modules and builds a library for code reuse.

2) A scalable placement algorithm based on the program partition theory and topology compression. The large number of heterogeneous devices compounded with a substantial amount of program instructions makes the placement problem challenging. To reduce the search space, ClickINC merges dependent instructions into blocks to reduce the entities for placement, and leverages the symmetry of the fat-tree topology (the most common data center topology) to reduce the number of devices under placement consideration. Such optimizations enable our efficient DP algorithm to handle up to ~1,000 switches.

3) An annotation-based approach to incremental program compilation. Running multiple programs from different users is challenging due to the potential resource conflicts; supporting *dynamic* user requests which may add or remove a program is even harder. ClickINC enables dynamic user requests while accommodating existing operator programs by providing an incremental compilation feature. Our idea is to treat the operator's programs as *base programs*. When synthesizing the base programs and user programs, ClickINC uses an *annotation-based* approach to provide both memory isolation and control flow isolation.

3.2 ClickINC's Workflow

Fig. 2 shows the overall architecture and workflow of ClickINC. At a high level, using ClickINC entails four steps:

(i) Writing a user program: ClickINC provides users a high-level, Python-style language to write INC programs. Users can use *built-in modules* which encapsulate common functions. Meanwhile, users can specify application performance requirements through the module parameters.

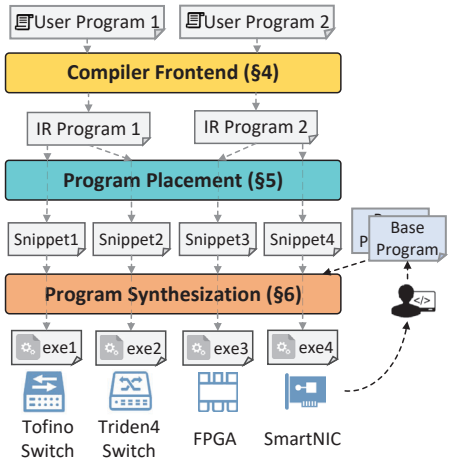


Figure 2: ClickINC architecture and workflow.

(ii) Compiling user programs to IR programs: The ClickINC compiler frontend compiles each user's program into an Intermediate Representation (IR) program, where the IR instruction set is platform-independent. We choose the representative IR instructions from each platform and merge the common ones.

(iii) Placing IR programs: Then ClickINC decides a placement plan to deploy IR programs distributedly on network-wide heterogeneous devices. To handle a large number of programs and devices, ClickINC uses a dynamic programming algorithm to find the placement plan with the highest gain. Each user's program may be split into multiple *snippets*, one for each device.

(iv) Deploying on heterogeneous devices: Finally, ClickINC compiler backend compiles snippets and the base programs (from the operator) to executable device programs in device-specific languages. Each executable includes the base program and one or more user snippets running user-specific applications.

4 PROGRAMMING ABSTRACTIONS

4.1 User Programming

Abstraction/Interfaces. INC programming can be cumbersome. At the device level, the heterogeneous resources, network topology, and target languages need to be considered; at the program level, a complete INC program needs to tend every packet handling detail including the inter and intra-device communication protocol. To hide the complexity, ClickINC is built on the One Big INC (OBI) abstraction (Fig. 3) which contains elements in three levels.

One Big Device. In OBI, the entire network is abstracted as a single virtual programmable device $\mathcal{D}$ to INC developers. The target devices comprise switch ASICs, multi-core smartNICs, FPGA smartNICs, and FPGA accelerator card, denoted as A, N_S, N_F, and F, respectively. Especially, a switch ASIC can be equipped with a bypass accelerator cards, denoted as $\hat{A}$, to enhance its memory and processing capacity. Thus, $\mathcal{D} = \{A, \hat{A}, N_S, N_F, F\}$.

Transparent Network. The above elements make an INC program a piece of standalone software. Behind the scenes, packet modification on devices (e.g., INC header insertion, removal, and update) is needed. ClickINC handles all such works with a generic internal

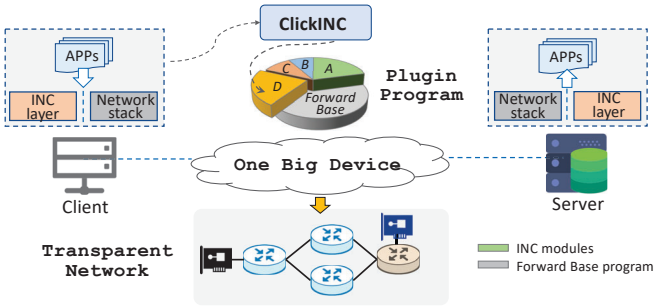


Figure 3: ClickINC OBI Abstraction.

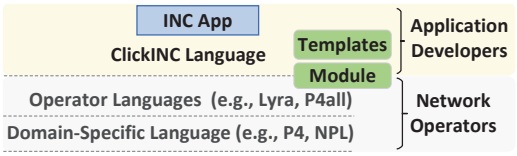


Figure 4: Languages to program network devices.

header structure by the "INC layer" maintained on each end device, and makes these issues transparent to both INC developers and end-host applications.

Plugin Program. Although One Big Device frees developers from dealing with device heterogeneity and network topology, an INC program still needs to integrate with the underlying forwarding function and the existing INC applications. OBI allows developers to focus on the INC function alone and deem an INC program as a standalone plugin on the One Big Device. The heavy lifting for program partition, mapping, and integration is handled behind the scenes. An INC program is plugged in or unplugged from the base forwarding program without affecting existing INC functions.

ClickINC Language. For best appeal to users, ClickINC preserves the high-level language abstraction for application developers as illustrated in Fig. 4, which differs from operator language (e.g., Lyra) and domain-specific language (e.g., P4). Fig. 5 shows the grammar of the ClickINC language. A program consists of simple and compound statements. A simple statement can assign an expression to a variable. A compound statement can control branching or looping. A branching statement is composed of a condition expression and two branch bodies containing further statements. A loop statement is composed of a condition and a body to be executed if the condition is met. An expression is composed of basic operators (Python built-in) and operand (an expression, a variable, or a constant). A function is treated as an expression which outputs a result by computing on arguments. ClickINC supports a Python-like coding style.

The ClickINC language introduces some INC specific elements to ease the programming on network devices. The *Fields*, *Objects* and *Primitives* abstractions are commonly used in INC applications [16, 17, 20, 33]. A field is a data type that can be used to declare variables with the packet header semantic. An object is a collective data type used to declare variable for five INC objects: Table, Array, Seq, Hash, and Crypto. INC primitives, including Get, Write, Clear, Count, Drop, Fwd, and Copy, operates on the INC objects.

Program G :== $var{=}E \mid G \mid$ if C: G else: $G \mid$ for C: G
Predicate C :== $(E\&E) \mid (E|E) \mid \sim E$
Expression E :== $V \mid var \mid const \mid F \mid E \odot E$
Function F :== max() | min() | range() | slice() | << | $\cdots$
Field V :== **value** | **header**
Object O :== Table | Array | Hash | Seq | Sketch | Crypto
Primitive P :== get(O) | write(O) | clear(O) | count(O) | del(O) | drop() | fwd() | copy(O, V)

Figure 5: ClickINC grammar. $\odot$ denotes arithmetic or bit operations, and underlined elements are ClickINC specific (see Table 7 in Appendix A for "Function F").

Each INC module is internally encoded in a platform-independent language (i.e., the IR in §4.2). When compiling user programs, the ClickINC toolchain links the INC modules to their IR implementations.

Modular Programming. The INC service provider implements the INC specific elements as modules. With such modular programming, we incorporate the INC-related data structures and operations into a user-friendly high-level programming environment. A user can assemble a program with the ClickINC language and the INC modules. Fig. 1 shows an example of implementing a Count-Min Sketch using the INC object Array and Hash function.

Template. The service provider can also define common INC programs as *templates*, and provide them to users as libraries. ClickINC provides the templates for MLAgg, KVS, and DQAcc (for SQL DISTINCT function), which cover a broad range of INC applications.

To use a template, users need to provide a configuration profile, so that to configure the module/template parameters. Users can configure module/template data structures, e.g., Array size, directly. Certain modules may need hardware-specific configurations that are obscure to users. In this case, ClickINC provides the objective function API of application performance for the user. For example, a key-value search user may use $\max(0.7hit + 0.3acc)$ to indicate the preference on the hit ratio and the accuracy of statistics for missed queries, with weight of 0.7 and 0.3 respectively. Especially, as the OBI abstraction makes device transparent to users, leading to the difficulty of setting resource-related parameters. Therefore, ClickINC pre-learns a model to automatically set parameters based on empirical experimental results. The details of the templates and their configuration can be found in Appendix A.

Moreover, users can also incrementally add new logic to the existing templates, saving the efforts to "re-invent wheels". For example, Fig. 6 shows how a user can build a customized sparse gradient aggregation based on the MLAgg template: The user program first imports and customizes a MLAgg template as an instance (line 1); then detects the sparse part of the parameter vector and drops the sparse one (line 5-9); only the dense one will be aggregated by MLAgg instance (line 10).

User-defined Module. Although we suggest the modules to be implemented by the service provider for simplicity, ClickINC reserves the flexibility for users to design their own INC modules, called user-defined modules.

To develop a user-defined INC module (i.e., object and primitive shown in Fig. 5), a user needs to use the "low-level" instructions to write the module program. These low-level instructions could be IR instructions or operator-level instructions (like Lyra, and shown in Table 8–Appendix A.4).

```
1  agg = MLAgg(row, dim, is_convert, scale)
2  for i in range(BlockNum):
3    sparse = 1
4    for j in range(BlockSize):
5      index = BlockNum * i + j
6      if hdr.feat[index]!=0:
7        sparse = 0
8    if sparse = 0:
9      del(hdr.feat[index])
10 agg(hdr)
```

Figure 6: The user program based on the MLAgg template, performing sparse gradient aggregation.

4.2 Program Intermediate Representation

Platform-Independent Intermediate Representation. To compile a user program to machine code on heterogeneous devices, ClickINC first compiles the program into an IR program.

ClickINC summarizes the IR instruction set from the different platforms it supports. The IR instruction set is listed in Fig. 17 in Appendix A.4. Some instructions are common on all platforms and the others only run on certain platforms: there are 13 classes of them, and each platform supports a subset of them as shown in Table 9 in Appendix A.4. Such instruction constraints will take effect in later program placement.

The ClickINC IR instruction set includes declaration instructions and operation instructions, where the former defines variables and the latter operates on variables. As the ClickINC IR instruction set needs to adapt to programmable network devices (e.g., in pipeline switches, a packet must sequentially traverse the pipeline stages without rewinding back in one pass), it does not support control flow transition (i.e., instructions like `goto` or `jump`). An IR program is therefore executed sequentially.

Compiler Frontend. The ClickINC frontend compiler compiles a user program into an IR program in the following passes: (1) inline all the bodies of the functions in the main program from the unified library; (2) unroll the loops if it makes constant pass of the iteration, e.g., `for i in range(3)`, otherwise an error will be reported; (3) convert the if-else branches to ternary operators in the format of `condition? instr`; (4) split the instructions into single-operand ones. Especially, instructions with temporary variables are transformed into Static Single Assignment (SSA) to eliminate the write-after-read and write-after-write dependencies, helping the IR Directed Acyclic Graph (DAG) construction in later program placement (§5.2).

5 PROGRAM PLACEMENT

We formulate the problem of distributing an IR program on multiple devices as an optimization problem and solve it by a dynamic programming algorithm.

5.1 Problem Statement

The network contains multiple programmable devices, and a program's instructions can be distributed on multiple devices. Placing an IR program on network devices is an *optimization problem*, where we wish to maximize the traffic volume served by INC while minimizing the resource consumption as well as the inter-device

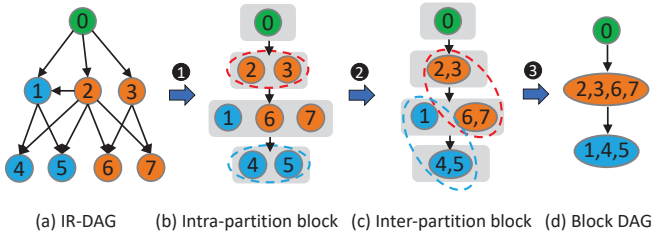


Figure 7: An example of block construction.

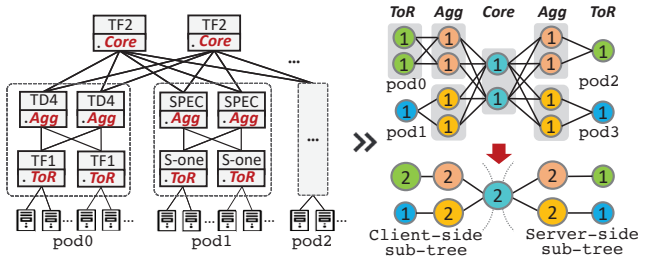


Figure 8: Topology Simplification (number in a circle: the number of merged devices, color: device type).

communication overhead. The solution needs to make a few trade-offs: placing more blocks on a device is simpler but limits the INC capability and capacity, and distributing blocks on multiple devices incurs inter-device communication overhead. Another key invariant in program placement is to keep the program execution equivalent as being executed on a single device.

Complexity. To place a program on multiple devices is to find devices for each instruction of the program. The problem's searching space exponential of program size and network scope, i.e., $O((M \cdot S)^N)$ where M, S, and N are the number of devices, pipeline stages or cores for SoC device, and IR instructions. Naïve methods may find sub-optimal results: greedily choosing a single path cannot utilize the multi-path resources; simply replicating the program on all paths could lead to device overloaded. Existing methods usually cannot handle the problem in a large scale. Lyra needs manually labeling the candidate devices of a program, which limits the result optimality; if "all" devices are labeled as candidate devices, its SMT solver approach cannot give the result in an acceptable time.

Intuition. We take three intuitions to reduce the algorithm complexity in ClickINC. First, we group IR program instructions into *blocks*, where all instructions in a block are executed all or none, and thus, one block can represent all its instructions in the algorithm (reducing N, §5.2). In a DCN, there could be multiple paths between two communicating INC hosts. On each path, the IR program blocks must be placed sequentially; among the paths, blocks are replicated on devices to guarantee the traffic on different paths is processed by the same program; two paths' intersection segment can hold blocks shared by both paths.

Second, we group DCN devices into equivalent classes, and use a class to represent all its devices in the algorithm (reducing M). Third, we find that the placement problem can be divided into isomorphic sub-problems, and thus propose a dynamic programming (DP) algorithm to search for the optimal solution, which gives a solution in polynomial time.

5.2 IR Block DAG Construction

ClickINC first transforms the IR program into a Directed Acyclic Graph (DAG) of disjoint instruction blocks to comply with the sequential instruction execution. A block is a basic placement unit. Each block contains instructions in the original order as in the IR program, and the union of blocks equals the IR program.

In ClickINC, the IR block DAG construction should also comply with several practical principles to ensure correctness. First, the instructions operating on the same state should be in the same block to avoid inconsistency. Second, the instructions in the same block should be of the same type to ensure the block can be placed on some devices (not all devices support all instruction types). Third, a block's size should be limited by a threshold parameter decided by the device capability. Appendix B.1 formalizes these principles.

ClickINC initializes each instruction as a block and gradually merges the blocks complying with the above constraints. The algorithm takes three steps.

Step 1: construct instruction dependency graph. If an instruction i reads a variable whose value is written by a previous instruction j, i *depends on* j. INC applications have a subtle pitfall: the program is driven by packet arrival events and there are inter-packet *states* (e.g., a packet counter). *All instructions that write or read the same state are mutually dependent.* The other variables with a life span of a packet are called *temporary variables*.

Step 2: merge instructions within a loop. The IR program can be viewed as a directed graph G, with the instruction as the node V and the dependency as the edge E. ClickINC iteratively merges nodes that form a loop. When multiple nodes (denoted as N) are merged as one, a new node (i.e., block) forms to replaced the old ones, and the edges between the merged nodes N and the other nodes $V - N$ are replaced by edges between the new node and other nodes. The algorithm repeats until there is no loop in the graph.

Step 3: merge non-exclusive blocks to compact the DAG. After eliminating loops, the graph becomes a DAG. ClickINC further runs Kahn's topological sort algorithm to partition the graph and merges non-exclusive blocks. Fig. 7 illustrates an example. Kahn's algorithm takes iterations to partition a DAG: each iteration takes the nodes whose input degree is 0 as one partition and removes these nodes and their related edges (Fig. 7b). After the Kahn partition, ClickINC further merges blocks whose instructions are of the same type, i.e., being non-exclusive, within the same partition (Fig. 7b-c) and the adjacent partitions (Fig. 7c-d) without exceeding the block size limitation. The process repeats until no more blocks can be merged.

5.3 Topology Simplification

ClickINC further reduces the search space for program placement by simplifying the network topology. ClickINC leverages the DCN's topological characteristics to make the reduction. The network devices in a DCN can be divided into several *equivalent classes (EC)*, where devices in the same class have the same physical wiring with the other classes.

For a three-tier fat-tree topology, all its ECs can be computed bottom-up in the topology. All ToR switches connecting with the same servers form an EC, all aggregation switches connecting with the same ToR switches form an EC, and all core switches form an

EC (as they connects to the same aggregation switches). Based on the proof of the device equality in EC for program placement (see Appendix B.2), we can merge the switches in an EC as one virtual node, and thus the DCN topology is simplified to a tree (Fig. 8).

In the later program placement, ClickINC also takes advantage of the path symmetry. All physical servers are at the leaf nodes of the topology, and traffic goes upwards to a root and goes downwards along the tree. Thus the tree is segmented into two parts by the root, i.e., the client-side sub-tree and server-side sub-tree.

5.4 Placement Algorithm

Optimization Goal. The program placement algorithm aims to find a solution to maximize the traffic served by INC with the minimum resource consumption and network bandwidth for passing parameters between blocks (§6). With $x_{v,d} \in \{0, 1\}$ indicating whether block v is placed on device d, the objective $G(x)$ can be formalized as:

$$G(x) = \omega_t h_t(x) - \omega_r h_r(x) - \omega_p h_p(x), \tag{1}$$

where h_t is the ratio of traffic served by INC, h_r is the ratio of resource consumed on devices, and h_p is the ratio of data transferred across devices. The parameters ω_t, ω_r, and ω_p balance the three factors. We empirically set ω_t as 1/2 to prefer high throughput, and tune ω_r and ω_p dynamically according to the resource availability as the algorithm proceeds.

$$h_t(x) = \sum_{l \in L_p} \left(\bigwedge_{v \in P} \sum_{d \in l} x_{v,d} \right) \times \frac{t_l}{\sum_{l \in L_p} t_l},$$

i.e., the overall normalized traffic volume on the selected paths, where t_l is the traffic volume on each path;

$$h_r(x) = \sum_{d \in D} \sum_{v \in V} x_{v,d} \times \frac{r(v)}{\sum_{v \in V} r_v},$$

i.e., the overall normalized resources on the selected devices;

$$h_p(x) = \sum_{d_i, d_j \in D} \sum_{v_k, v_l \in V} x_{v_k,d_i} x_{v_l,d_j} \times \frac{\phi_{v_k,v_l}}{\sum_{d \in D} \sum_{v \in V} x_{v,d} \phi(v)},$$

i.e., the overall normalized volume of extra parameter incurred due to program partition between selected devices, where ϕ_{v_k,v_l} denotes the amount of extra data transferred between devices v_k and v_l, and ϕ_v refers to all extra data incurred by the block v.

Dynamic Programming Algorithm for Placement. Even with the reduced topology, the searching space to find an optimal placement of IR program is still too large due to the possible multiple flow paths from multiple pods. SMT or ILP solvers cannot give the solution in an acceptable time. ClickINC uses an innovative dynamic programming algorithm with pruning. In detail, for the two sub-trees illustrated in Fig. 8, we try to allocate the program but from different directions (i.e., sequentially allocate instruction blocks from leaf to root for the client-side sub-tree and do it in the reverse order for the server-side sub-tree, so that the problem is translated into two sub-tree-based program placement). Then we link the two sub-tree placement results by the root node, i.e., traverse all partial placement results of sub-trees, and choose the one with the largest gain of Eq. 1 from all feasible combinations. The placement task on each sub-tree devices can be discomposed as two sub-tasks: (1) place the instruction blocks across devices for multi-path traffic; (2) decide the placement of instructions in each block within a particular device. We illustrate how ClickINC addresses these two sub-tasks.

Algorithm 1: Multi-path allocation

```
   Input: R, S, D: the set of resources, stages, all available devices, B: the set of
          instruction block to be allocated.
   Output: s: the allocation solution.
1  ω_t, ω_p, ω_r ← adjust(R, D, S);
2  CDP←DFS_DP(CTree.root, 1);
3  SDP←DFS_DP(STree.root, -1);
4  for B ∈ CDP[CTree.root] do
5      B' ← B-B;
6      if B'∈ SDP[STree.root] then
7          s ← min(s, CDP[CTree.root]+SDP[STree.root])
8  return s;
9  Function DFS_DP(r,d):
10     A ← ∅;
11     if r = ∅ then
12         return
13     for c ∈ r.child do
14         DP_sub[c] ← DFS_DP(r)
15     sub_G[∅] ← 0;
16     for i ∈ ∪ DP_sub[r.child].keys do
17         sub_G[i] = sum(DP_sub[i])
18     for i ∈ sub_G.keys do
19         B_ava ← {b|b ∈ B − i; in_degree(b, d) = 0};
20         for B ∈ enum B_ava do
21             curr ← call Algorithm 2(S[r], R[r], B);
22             DP[r][i+B] ← max(DP[r][i+B], sub_G[i]+curr+calc_hp(i,B));
23 return DP;
```

- *Cross-device multi-path solution.* Let H_{B,D_i} denote the maximum gain of placing block(s) B on a tree with D_i as the root. When the tree is a single device D_i, H_{B,D_i} equals the gain of Eq. 1. When the tree has subtrees, ClickINC places a partition B' on the root node (the partition can be $\emptyset$ or B) and the remaining onto the subtrees, and the gain H_{B,D_i} is the sum of that on the root and the subtrees; by iterating all possible partitions, ClickINC finds the one which gives the maximum H_{B,D_i}, i.e.,

$$H_{B,D_i} = \max_{B' \in Partion(B)} \left(\sum_{j \in son(D_i)} H_{B-B',D_j} + G(D_i, B') \right). \tag{2}$$

The problem can be recursively divided into isomorphic sub-problems. We design a dynamic programming algorithm to compute the problem bottom-up. The pseudo-code is shown in Algorithm 1: line 1 adjusts weights; line 2-3 uses Depth First Search (DFS) to traverse two sub-trees and performs allocation; then for a leaf node, line 20-21 enumerates instruction blocks and calls Algorithm 2 to place instruction in blocks within a device; for internal nodes, line 16-17 integrates allocation results of possible branches and line 22 executes the DP following Eq. 2 where $calc\_hp(\cdot)$ computes the cross-device communication overhead. Especially, as illustrated in line 10, we prune the illegal enumeration results that violate block dependency to reduce the solution space. This algorithm can be applied to a fat-tree or a spine-leaf topology with any number of layers.

- *Intra-device solution.* To place instructions within a device, we use another DP algorithm to ensure (1) the instructions satisfy resource constraints; and (2) the placement has the largest gain according to Eq. 1. Thus, we can derive:

$$H_{p,S_i} = \max_{p' \in Partition(p)} \left(H_{p-p',S_{i-1}} + G(S_i, p') \right), \tag{3}$$

Algorithm 2: Instruction allocation within a device

Input: S_d, R_d: the stages, resources of device d, P: set of instructions to be placed.
Output: I: Instruction allocation results.

```
1  I[-1] ← {∅ : 0};
2  for s← 0 to S_d do
3      for i ∈ I[s − 1] do
4          if calc_resource(p)≤ R_d[s] then
5              I[s][i] ← max(I[s][i], I[s − 1][i]);
6              P_nd ← {p|p ∈ P − i; in_degree(p) = 0};
7              for p ⊆ enum P_nd do
8                  if ∃i' ∈ I[s].keys && i+p ⊆ i' then
9                      continue;
10                 if ∃i' ∈ I[s].keys && i' ⊆ i+p then
11                     del I[s][i'];
12                 I[s][i+p] ← max(I[s][i+p], I[s-1][i]+G(p));
13 return I;
```

where p is the instructions that are placed, $S_i = [s_1, s_2, \cdots, s_i]$ is the set of stages for pipeline devices ($S_i = s_0$ for non-pipeline device). On a pipeline, the instruction-to-stage mapping has a huge solution space. To improve efficiency, ❶ the infeasible solutions violating the instruction dependency are pruned (line 6 of Algorithm 3); ❷ the target function Eq. 1 prefers solutions with more compact placement (i.e., each stage should use up at least one type of resource), so inadequate solutions are pruned (line 8-12). With these, the DP algorithm achieves the similar solution as SMT in a much shorter time. The pseudo-code is shown in Algorithm 2.

Adaptive Weight. As the algorithm proceeds, ω_r is set as $\omega_r = 1 - 2^{r-1}$ and $\omega_p = 1/2 - \omega_r$, where r is the ratio of remaining resources. The adaptive weight could raise the importance of device resource allocation as the remaining resource decreases (a smaller r leading to a larger ω_r).

Placement Constraints and Pruning. As the DP algorithm searches the solution with the highest gain, the following pruning techniques are applied to reduce the search space. When one of the following constraints is violated, the algorithm sets H_{B,D_i} as negative infinity ($-\infty$) and stops exploring the branch: (1) if a device's resource capacity cannot satisfy the block; (2) if an instruction placement violates the instruction dependency; (3) if a device's computation capability fails to satisfy the block's instruction type. Besides, the target function Eq. 1 prefers solutions with more compact placement (i.e., each stage should use up at least one type of resource), so inadequate solutions are pruned.

To map the program on the devices with various constraints, we propose device modeling based on different architectures (i.e., pipeline, multi-core) to formalize the device-level instruction placement and describe the chip-specific constraints in Appendix D.

6 PROGRAM SYNTHESIS

Each device runs a network operator-deployed program, called *base program*, to perform the basic network functions such as packet validation, forwarding, etc. Multiple users' INC programs (snippets) placed on the device rely on ClickINC to synthesize them as one big program.

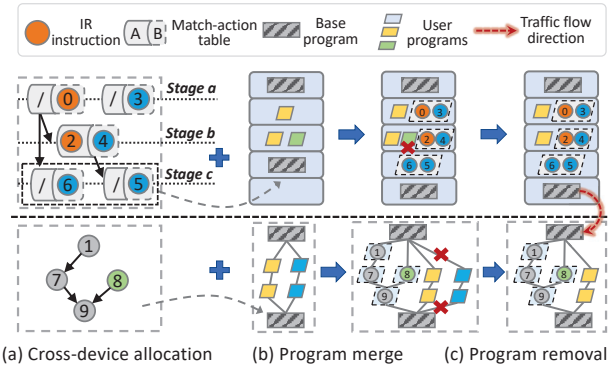


Figure 9: Program Synthesis.

A program typically consists of a header parsing snippet and a packet processing snippet. User programs and the base program could parse different header fields for their own packet processing.

Refine Runtime Data Plane. The network data plane are refined to support program execution on distributed devices. The two refinements are transparent to the users: for a user's traffic, the first network device inserts a special header for the following refinement, and the last one removes it.

First, temporary variables may be shared by multiple devices. ClickINC allows the user packets to carry the shared variables from one device to its downstream devices. ClickINC packet header has a field `Param` to store the temporary variables. Note that persistent variables are only used and placed on one device, and the static single assignment transformation makes temporary variables only have dependency from the successor device to the predecessor along the DAG.

Second, ClickINC allows placing replicated blocks along a path. For example, a program with blocks 1, 2, and 3 may be placed along a four-hop path as 1, 2, 2, 3. Thus, ClickINC needs to decide and tell the devices with replicated blocks which of them processes a packet. ClickINC assigns each block in the DAG program a step number, and adds a `step` field in the packet header. A device attempts to match the packet `step` field with its own block's step, if they match, the block is executed and the packet step is increased to the next step, otherwise, the packet skips the processing (if the packet step number is larger) or dropped (if the packet step number is smaller). Allowing replicated blocks in the network also provides another advantage: if the network experiences a transient failure, a packet can skip the faulty device and get processed by the successor device with replicated blocks.

Compiler Backend. ClickINC first isolates user programs from each other and the base program. It renames variable in the user programs, so that after compilation their programs access isolated memory region, without violating each others' data. For example, the `mtb` variable in a KVS program `kvs_0` is renamed as `kvs_0_mtb`. Then it adds a user ID match to filter out the user's traffic for its own program.

```
1  if (INC_1_hdr.isValid()) {logic1;}
```

ClickINC compiles each program individually into device-specific instructions, called *device program*. These device programs are merged with the following optimization, and eventually compiled as an executable.

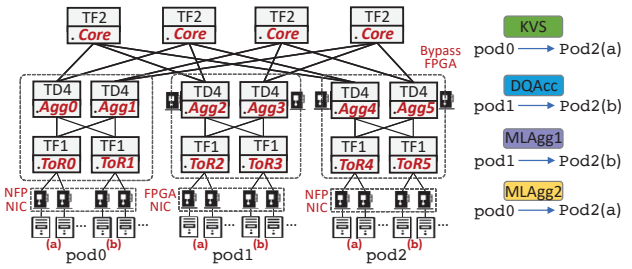


Figure 10: Network Topology in Emulation.

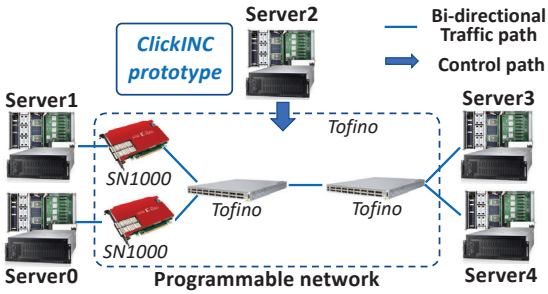


Figure 11: Testbed.

Program Merge. ClickINC merges header parsing snippets and packet processing snippets separately. The header parsing follows a tree structure. When merging two programs' header parsing, ClickINC scans both trees, merges the different branches, and eventually outputs a merged tree.

Merging packet processing snippets is more complex due to the dependency between the user programs and the base program. For example, the forwarding function in the base program depends on the user program if the user program changes the packet's IP addresses (e.g., NetCache [17]); the user programs depend on the packet integrity check function in the base program, because only valid packets should be handed to the user programs. Thus, the base program is divided into a *head* part and a *tail* part, where *head* contains functions depended on by the user programs and *tail* contains functions depending on the user programs.

For pipeline devices, as the upper part of Fig. 9(b) shows, the user program is placed between *head* and *tail* of the base program. The user program is moved to stages as early as possible to reduce the overall stages. For multi-core devices, ClickINC merges the dependency graphs of the user program and the base program according to node dependency, and then merges the corresponding code pieces based on the topological sorting order on the merged graph, as illustrated in the lower part of Fig. 9(b).

Incremental Compilation for Dynamic Program Merge & Removal. ClickINC applies an annotation-based method to support incremental user program merging and removal. ClickINC associates each user program with an annotation indicating its ownership. During the compilation, the annotation is associated with each instruction. When merging a user program into the base program, ClickINC incrementally adds the new user annotation to the shared instructions and sets the new user's own instructions with its annotation.

When a user revokes its INC service request, ClickINC iterates the synthetic program's instructions, and removes the user's annotation; if an instruction has no annotation, the instruction is removed.

At runtime, ClickINC makes lazy enforcement for program removal to reduce the service interruption. To remove a user program, the program instruction dependency graph is updated and the resource is recorded as released in ClickINC without immediate enforcement. Meanwhile, the traffic matching rules are updated so that the user program is not effective anymore. When a request for adding a new program is submitted, ClickINC enforces the new updated graph as the executable on the device.

7 EVALUATION

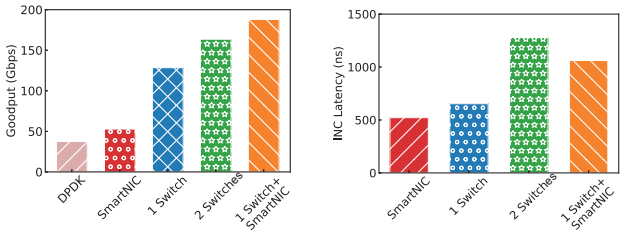
We conduct experiments to display ClickINC's advantages. (1) ClickINC makes use of resources on heterogeneous devices to achieve high INC performance (§7.2); (2) The modular programming abstraction allows more efficient INC development for users than the other solutions do, including Lyra, P4all, and P4, in terms of line of code and programming efficiency (§7.3); (3) the cross-device INC program allocation outperforms the current practices; (4) ClickINC uses an efficient DP algorithm to perform program placement, achieving very short compiling time and high scalability over both the number of devices and program size; (5) With incremental deployment, ClickINC achieves minimal impact on the network devices, traffic, and other deployed INC programs.

7.1 Experiment Setting

Implementation. The ClickINC framework is implemented in C++ and Python with 8,755 and 3,133 lines of code, respectively, and runs on a desktop with an Intel Core i7 4GHz CPU and 16GB RAM. It currently supports Tofino, Tofino2, TD4, Netronome smartNIC, Xilinx FPGA, covering the target DSL of $P4_{16}$, NPL, Micro-C, and Verilog HDL.

Emulator. We construct a software emulation platform for evaluating ClickINC on large networks with heterogeneous devices. A server equipped with the switch SDE [14] for Tofino series ASIC and BCM simulator for TD4 can emulate all the chip functions. Using virtual NIC pairs to act as switch ports, the emulator presents the same resource constraints as a real switch and can be controlled using the same API. Xilinx and Netronome also provide the software behavioral model/simulator to emulate hardware FPGA/NFP smartNIC which takes PCAP files as input and output. We set up an emulator using 4 servers with 16 Virtual Machines (VM) (4 for Tofino2, 6 for TD4, and 6 for Tofino), 4 VNetP4 behavior model instances, and 8 NFP simulator instances which are organized in the topology as shown in Fig. 10. The communication between VMs is bridged through the physical NIC. Communication with the VNetP4 behavior models or NFP simulators is achieved by using a script program to generate and interpret the PCAP files.

Testbed. As shown in Fig. 11, Server2 runs the ClickINC controller and serves as the switch controller as well. Server3 and Server4 run DPDK on Mellanox ConnectX-5 dual-port 100G NIC. Equipped with Xilinx Alveo U280 FPGA and Netronome Agilio LX smartNIC, respectively, Server0 and Server1 generate the traffic of integer parameters. Two Edgecore Wedge100BF-32X switches are interconnected, and each switch further connects with the two smartNICs



(a) Goodput (b) In-network latency

Figure 12: Performance comparison.

Table 1: Comparison between ClickINC and other peers

Language	LoC (KVS/ MLAgg/DQAcc)	Modular Programming	Incremental Compilation	Cross-Device Placement
ClickINC	16/56/13	Y	Y	Y
Lyra [8]	125/232/243	N	N	Y
P4all [11]	202/233/138	Y	N	N
P4$_{16}$ [28]	571/1564/403	N	N	N

Table 2: Trials and manhour in programming

Language	KVS		MLAgg		MLAcc	
	# of trials	time	# of trials	time	# of trials	time
P4$_{16}$	12	~1h	14	~3h	6	~30m
ClickINC	1	~10m	2	~25m	0	~5m

Table 3: Developer Productivity of Placing Multi-user Program over Multi-devices

Metrics	Method	KVS0	DQAcc0	MLAgg0	DQAcc1	MLAgg1	KVS1
# of trials	**P4$_{16}$**	2	16	25	31	24	13
	ClickINC	1					
Time	**P4$_{16}$**	~5m	>1h	>4h	>3h	>2h	~1h
	ClickINC	<10s					
Device	**P4$_{16}$**	ToR5	ToR0,1; Agg0,1	Agg0,1; Agg4,5	ToR1,2; Agg4,5	ToR2,3; Agg2,3	Cores
	ClickINC	ToR5	ToR0,1; ToR5	Agg4,5; ToR5	ToR2; Agg0,1	ToR2,3; Agg2,3	Cores
Resource	**P4$_{16}$**	1	2	2.25	2	2	4
	ClickINC	1	1.71	1.5	3	2	4
Comm.	**P4$_{16}$**	0	0.75	0.14	0.63	0.14	0
	ClickINC	0	0.33	0.16	0	0.14	0

or the two ConnectX-5 NICs, respectively. The link capacity is 100Gbps.

7.2 Application Performance

INC programs can achieve performance gain when compiled and deployed by ClickINC. We control the network with (1) no programmable device, (2) only smartNICs, (3) only one Tofino switch, (4) two Tofino switches, and (5) the smartNIC and a Tofino switch. We deploy the sparse gradient aggregation program in Fig. 6 via ClickINC in the five network configurations. Fig. 12(a) shows the aggregation goodput and Fig. 12(b) is the corresponding INC processing latency. Using setting (1) as the baseline, ClickINC compiles the sparse gradient compression on the smartNICs in case (2), which increases the goodput by reducing traffic volume. ClickINC compiles the aggregation on the switch in case (3), which increases the goodput by in-network traffic aggregation. The program performs better with two switches in case (4) than one in case (3), because the packet size can be larger in case (4), and ClickINC places the program on two switches, each processing a part of packets. And finally, with a combination of two heterogeneous devices, the program achieves the highest runtime goodput in case (5).

7.3 Program Development Workload

We develop three INC applications with ClickINC, Lyra, P4all, and P4$_{16}$. The applications are (1) a KVS program using a 5K-entry cache for 128b key and 16×32b value vector, and a 3×1K heavy-hitter for statistics of missed queries; (2) an MLAgg program with 5K aggregators for 24×32b integer parameter vector; (3) an SQL DISTINCT program with a 5K×8 rolling cache which filters queries with 32b value.

Program Complexity (LoC). Table 1 illustrates the Lines of Code (LoC) of the three programs in four frameworks. ClickINC programs are 4-18, 4-12, and 28-35 times shorter than that Lyra, P4all, and P4$_{16}$ ones, respectively. ClickINC's modular programming reuses existing modules (outperforming Lyra), its high-level language features (e.g., loop) are more concise, and its multi-user programming and synthesis allows user to only write INC specific logic (outperforming Lyra and P4all), and thus, the overall LoC is much shorter.

Developer Productivity.

• *Individual Program Development.* As a preliminary validation that ClickINC can improve the programming productivity, one of our authors with experience in P4 programming on Tofino writes the three programs respectively using P4$_{16}$ and ClickINC on a single device. Lyra and P4all's compilers are not publicly available when this work is done. A full study of the programmability of ClickINC is outside the scope of this paper. Table 2 shows the number of trials (a trial denotes a cycle of development, compilation, test, and debug) and time spent in development. ClickINC can reduce the development time by 6-7.2 times, and the developer makes very few errors when developing in ClickINC (0 or 2 for three applications).

• *Multi-user Program Placement and Synthesis.* With the three individual programs ready, we further let two students place multiple instances of the programs into the network, one with ClickINC and another manually. The topology is in Fig. 10, and all devices are assumed to be Tofino switches. There are six INC program instances: (1) KVS0, processing traffic {pod0(a), pod1(a)} → {pod2(b)}, (2) DQAcc0, {pod0(a), pod0(b)}→{pod2(b)}, (3) MLAgg0, {pod0(b), pod1(b)}→{pod2(b)}, (4) DQAcc1, {pod0(b), pod1(a)}→{pod2(b)}, (5) MLAgg1, {pod1(a), pod1(b)}→{pod2(b)}, and (6) KVS1, {pod0(b), pod1(b)}→{pod2(b)}. Table 3 shows the final placement results, including the time consumption and trials, and the placed devices, normalized resource consumption, and communication overhead.

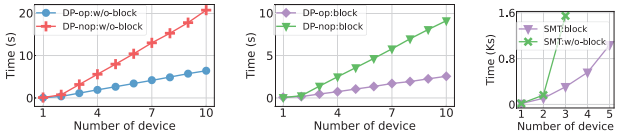
In the beginning, manually placing a program instance on multiple devices is trivial, e.g., KVS0 on ToR5, because all devices have abundant resources and the program does not need partition. But the placement process gradually slows down as the resource usage among devices becomes unbalanced, and the placement needs to jointly consider partition legality, resources availability, communication overhead, and load balancing. For example, it takes more than one and four hours to place DQAcc0 and MLAgg0, respectively.

In contrast, ClickINC automatically finds the optimal placement plan, and synthesizes the programs. The process is fast (< 10*s* for six instances), and error-free.

Table 4: Placement Plan from DP and SMT algorithms

INC program	depen-dency	stages		instructions		time (s)	
		SMT	DP	SMT	DP	SMT	DP
KVS	6	8	8	42	42	961	1.306
MLAgg	14	[8,6]	[6,8]	[14,11]	[10,15]	559	0.754
DQAcc	6	[8,8,1]	[6,8,3]	[39,21,1]	[35,16,10]	160	0.081

'$[x, y, ...]$' in the stage column means that the devices in the chain use x, y, ... stages, respectively; '$[x, y, ...]$' in the instructions column means that the devices in the chain are assigned x, y, ... instructions, respectively.



(a) DP: w/o-Block denotes no block construction (b) DP: with block construction (nop: no pruning) (c) SMT

Figure 13: Compiling time on the number of devices.

7.4 Effectiveness of Placement Algorithm

Optimality. We compare the result of ClickINC's DP-based allocation algorithm with the Z3 [24] SMT-based one that is used in existing solutions [8]. As the SMT solver is unable to handle a multi-path topology in an acceptable time, we use a simple chain with four Tofino switches, each switch with 8 pipeline stages. We place the three programs (§7.3) and measure the algorithm execution time and resource usage in the placement plan. We set the same optimization goal as Eq. 1 for both algorithms. The result is shown in Table 4.

The DP algorithm has a similar effect as the Z3 one in terms of resource consumption and the number of involved devices. But DP algorithm runs nearly one thousand times faster, thanks to the pruning technique.

Usually, a longer instruction dependency with fewer instructions indicates a smaller enumeration space and thus a lower processing time. This explains why MLAgg has a much shorter processing time than KVS. On the other hand, KVS has many independent stateful operations (for realizing cache) per dependency level, which degrades the pre-pruning effect, and thus consumes more compiling time than DQAcc, even though it has fewer instructions than DQAcc.

In addition, we also test the SMT algorithm without the optimization goal. As a result, it saves about half of the searching time as the algorithm only searches for a feasible solution; but it incurs larger communication overhead as the program is partitioned across all devices.

Impact of Block Construction and Pruning. We compile MLAgg with different settings of enabling/disabling block construction and pruning and measure the compilation time. Fig. 13(a) and Fig. 13(b) show the results. The two approaches can reduce the DP algorithm execution time by more than 50% separately, and by more than 80% together. Fig. 13(c) further shows that the DP algorithm has a linear processing time with the number of devices while the SMT solver has an exponential complexity.

Impact of Adaptive Weights. We place six instances of the three programs MLAgg, KVS, and DQAcc on the path from pod0(a) to pod2(b) in Fig. 10. The six instances are in the order of the second row in Table 5. We turn on and off the Adaptive Weight (AW) to observe its effects.

Table 5: Placement results with adaptive weights

	Devices/(instructions)						
	MLAgg0	**KVS0**	**DQAcc0**	**MLAgg1**	**KVS1**	**DQAcc1**	**MLAgg2**
Fixed weight	ToR0: ToR5 /(6:60)	ToR0: ToR5 /(34:47)	ToR0: [Agg0,1] /(3:25)	[Agg0,1]: [Agg4,5] /(7:59)	[Cores]: [Agg4,5] /(27:54)	[Cores] /(28)	/
Adapt. weight	[Cores] /(66)	ToR0 /(81)	ToR5 /(28)	[ToR0:5] /(33:33)	ToR0:[Agg 0,1]:ToR5 /(13:49:19)	[Agg4,5] /(28)	ToR0:[Agg0,1] :[Cores]:[Agg4, 5]/(10:4:20:32)

'$[\cdot,\cdot]$' indicates that instructions are duplicated on devices;
':' indicates that instructions are partitioned on devices;
'/' represents INC plugin cannot be placed on any device.

Table 6: The impact of incremental deployment

Step	Incremental deployment			Monolithic deployment		
	Affected Devices	**Affected INC**	**Affected traffic**	**Affected Devices**	**Affected INC**	**Affected traffic**
+KVS	2	0	3 pods	2	0	3 pods
+DQAcc	2	0	1 pod	2	0	1 pod
+MLAgg1	4	1	1 pod	8	2	3 pods
+MLAgg2	2	1	1 pod	4	3	3 pods
-MLAgg1	4	1	1 pod	8	4	3 pods

'+' or '-' mean to merge or remove an INC program.

In the beginning, all devices run only the base program with spare resources, and thus ω_r in AW is near zero, making KVS0 be placed on the four *Core* switches due to the dominance of ω_p; but for the fixed weight (FW), it is divided on the *ToR0* and *ToR5* switches to balance both communication overhead and resource consumption.

As the placement proceeds, the remaining resources decrease, the ω_r in AW increases, and the resource consumption begins to dominate the placement. MLAgg1 could have been fully placed on the *Core* switches but it is divided on *ToR0* and *ToR5*. In addition to the lower communication overhead, AW also has the advantage that the remaining resources are more concentrated on several devices than FW does, so it is more likely to hold a complete INC program in one device in the future. This explains why MLAgg3 can be deployed in the AW experiment but not in the FW one.

7.5 Incremental Program Synthesis

We configure the INC programs to make them resource intensive – KVS with a cache size of 100,000, MLAgg1 with 16-dimension floating-point parameters, and MLAgg1 with 16-dimensional integer parameters.

KVS and MLAgg2 serve applications from `pod0` (client) to `pod2(a)` (server) while DQAcc and MLAgg1 serve applications from `pod1` to `pod2(b)`. We assume there is always the background traffic from `pod0` and `pod1` to `pod2`.

We place KVS, DQAcc, MLAgg1, and MLAgg2 one by one. ClickINC performs incremental deployment (named ID), and we compare it with monolithic deployment (named MD). MD synthesizes and recompiles old and new programs each time. Table 6 shows the placement results.

In the beginning, ID and MD behave in the same way. KVS is placed on *Agg4,5* which have a bypassed FPGA to help host a huge cache. As *Agg4,5* are sitting on the path of traffic from {`pod0`, `pod1`} → `pod2`, all traffic will be interrupted during program loading on *Agg4,5*. DQAcc is placed on *Agg2,3*, and thus only affects traffic of `pod1` but not KVS in `pod0`.

When MLAgg1 is deployed, ID and MD start to behave differently. ID chooses *ToR2,3* with the FPGA NIC (for floating-point

calculation) and only affects traffic of pod1 including DQAcc program; MD decomposes the synthesized program of MLAgg1 and the old DQAcc (both from pod1 to pod2), which leads to instruction removal from *Agg2,3* and replacement on *FNIC1,2* and *ToR2,3,5* (because using *ToR5* and *ToR2,3* is more resource-efficient than using *Agg2,3* and *ToR2,3*), affecting all traffic and INC programs. To place MLAgg2, ID only changes device of *Agg0,1*, and affects only the traffic of pod0 and KVS; MD needs to synthesize KVS and MLAgg2, which changes *Agg0,1,4,5*, thus affecting all traffic. In summary, incremental program synthesis has a much smaller impact on traffic than that of monolithic deployment which is more likely to incur global traffic interruption.

8 DISCUSSION

This section discusses ClickINC's scope and limitations.

Program isolation. For different INC programs on the same device, ClickINC already achieves the function isolation and partial security isolation, but it lacks the performance isolation. The function and security isolation ensures the functions and resources of different INC programs on the data plane are independent, i.e., a buggy INC program cannot access the data and code of the other programs. However, ClickINC cannot defend a malicious INC program from tampering with the other programs intentionally by usurping the system resource and bandwidth in a disguised way. Measures should be taken to ensure the performance fairness. Performance isolation can be achieved by QoS rules and enforcement between users.

Parameter setting. Toward a user-friendly programming environment, ClickINC adopts a high-level abstraction of network devices, making device hardware, resource, and topology transparent to users. However, without such knowledge, some users may be puzzled in setting parameters for program especially for resource-related parameters. ClickINC currently provides a primary parameter automatical-setting model for programs derived from the provided templates by a pre-learned empirical estimation function but cannot set parameters for user-written programs, as illustrated in Appendix A.3. In the future, we will design a more general model to set parameters for user-written programs according to user's performance metrics and available network resources.

Target users. ClickINC makes INC easy-to-use by application developers, isolating the roles of network operator and application developer. Although in this paper, the ClickINC framework is proposed mainly for application developers to eliminate their burden of using INC, but it is also a good programming tool for network operators. Next, we will focus on addressing developing difficulties for network operators, and integrate the programming interfaces.

Program placement. Although ClickINC supports multi-path program placement, it assumes the topology is fat-tree or spine-leaf, and the devices in the same EC are the same in device type and resources, so that the topology can be simplified. In the future, we will improve the placement algorithms on the foundation presented in this paper to support any multi-path topology with relaxed assumptions on devices.

Supported architectures. Currently, ClickINC only considers FPGA as a pipeline-based device which can provide more features than switch ASICs. More potential can be explored in this space. Programmable chips with different architectures (e.g., Silicon One [3], Spectrum [27], and Trio [34]) and target DSLs (e.g., DOCA [26], Microcode [34]) can also be modeled and supported.

9 RELATED WORK

INC Applications. Recent INC acceleration solutions only provide a monolithic program that couples the application functions (e.g., key-value store, application data aggregation), the network functions (e.g., reliability, packetization), and the programming abstraction and runtime environment of a specific platform. §5.1 lists the examples of key-value store [17], synchronous aggregation [20, 30], and database query [21, 33]. Besides, ASK [10] proposes a solution for asynchronous key-value stream aggregation.

INC Frameworks on a Single Platform. A class of works aim to improve the INC program development on a single platform. Click [23] supports modular policy configuration on the control plane for traditional routers. μP4 [31] allows modular programming in data plane on PISA switches by composing reusable libraries. P4all [11] advances modular programming by introducing elastic parameters to be configured by the compiler based on an objective function. NetRPC [36] proposes INC-enabled RPC system for simplifying INC adoption; it pre-defines several operation primitives on the switch and supports limited use cases. These three works target on a single device. Flightplan [32] supports the partition and distribution of a *single P4 program* on heterogeneous devices. Its program needs to be manually partitioned based on empirical decisions.

INC Frameworks on Multiple Platforms. The existing cross-platform frameworks target different scenarios or users, and provide different abstractions. Lyra [8] is a unified language for heterogeneous devices to hide hardware differences. It helps "network operators" but not as much for the application developers: (1) Lyra applies to programmable switches with pipeline-based ASICs; (2) Lyra's programming abstraction couples the network operations, and multi-tenant application offloading, leading to a cumbersome development; (3) Lyra only searches for a feasible solution based on SMT solver that is time-inefficient for a large-scale network with many devices.

10 CONCLUSION

ClickINC is the first work of its kind that truly decouples the INC application development and deployment process from the network and device details. The heavy lifting of ClickINC presents a simple programming interface to users and allows users to focus on the application logic only. The clear split of duties ensures agile development and quality deployment for new applications, helping accelerate the adoption of the INC paradigm and enjoy the benefits it offers. Extensive experiments show ClickINC is superior to existing tools.

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We thank our SIGCOMM reviewers for their insightful comments, and publication chairs Richard Ma and Xia Zhou for useful feedback. This work is supported by NSFC (62032013, 62272258), and NSFC-RGC (62061160489).

REFERENCES

[1] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: a versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[2] Pietro Bressana, Noa Zilberman, Dejan Vucinic, and Robert Soulé. 2020. Trading latency for compute in the network. In *Proceedings of the Workshop on Network Application Integration/CoDesign*. 35–40.

[3] Cisco. 2023. Silicon One. https://www.cisco.com/c/en/us/solutions/silicon-one.html.

[4] Alibaba Cloud. 2023. SNA*: a hyper-converged programmable gateway. https://opennetworking.org/wp-content/uploads/2022/05/Dennis-Cai-Final-Slide-Deck.pdf.

[5] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review* 46, 2 (2016), 18–24.

[6] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 1–7.

[7] Python Software Foundation. 2020. In-band Network Telemetry (INT) Dataplane Specification. https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[8] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*. 435–450.

[9] Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. 2017. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications* 88 (2017), 50–71.

[10] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 33–47.

[11] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association.

[12] Intel. 2021. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[13] Intel. 2023. Introducing IPDK. https://opennetworking.org/wp-content/uploads/2022/05/Deb-Chatterjee-Final-Slide-Deck.pdf.

[14] Intel. 2023. P4 Studio SDE. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html.

[15] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the Fast Lane: A Line-Rate Linear Road. In *Proceedings of the Symposium on SDN Research (SOSR)*.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

[17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching (SOSP). In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*.

[18] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562.

[19] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*. 90–106.

[20] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *NSDI*. 741–761.

[21] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. 2019. The Case for Network Accelerated Query Processing. In *CIDR*.

[22] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. 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 (ASPLOS)*. 795–809.

[23] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 1–13.

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[25] NPL. 2023. Network programming language. https://nplang.org.

[26] NVIDIA. 2023. DOCA SDK Early Access. https://developer.nvidia.com/nvidia-doca-sdk-early-access.

[27] NVIDIA. 2023. Spectrum SN4000 Open Ethernet Switches. https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/.

[28] ONF. 2023. P4 Open Source Programming Language. https://p4.org.

[29] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*.

[30] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808.

[31] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. 2020. Composing dataplane programs with μP4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*. 329–343.

[32] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 571–592.

[33] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2407–2422.

[34] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 633–648.

[35] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, et al. 2022. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1345–1358.

[36] Bohan Zhao, Wenfei Wu, and Wei Xu. 2022. NetRPC: Enabling In-Network Computation in Remote Procedure Calls. *arXiv preprint arXiv:2212.08362* (2022).

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A CLICKINC LANGUAGE

This section explains the details of ClickINC language.

A.1 Templates

KVS. For KVS, it mainly contains a cache with exact-match to maintain key-value results, a counter for counting hits of each entry in cache, and a heavy hitter (count-min sketch plus bloom filter) for recording missed queries. The configurable options are: (1) the cache can be realized as using stateful array or stateless matching table, which is decided by application requirements (e.g., the value dimension and size); (2) the cache depth (same as counter), the number of counter-min sketch and bloom filters to compose a heavy hitter; and (3) the type of hash functions, and the triggering threshold of heavy hitter. All of these configurations are decided by compiler, according to profile provided by users or as default.

MLAgg. MLAgg performs aggregation for distributed ML parameters from different works, and the structure contains multiple arrays working as *aggregator* to preserve aggregated parameters, *bitmap* to track workers that have been aggregated, a *counter* to record the number of aggregated parameters, and *sequence* to record the ID of parameter for each stage ML job. The configurable options

```
from Funclib import *
cache=Table(type="exact",keys=hdr.key,vals=hdr.val)
cms=Sketch(type="count-min",keys=hdr.key)
bf = Sketch(type="bloom-filter", keys=hdr.key)
if hdr.op == REQUEST:
  vals = get(cache, hdr.key)
  if vals != None:
    back(hdr={op:REPLY, vals:vals})
  else:
    count(cms, hdr.key, 1)
    if get(cms, hdr.key) > TH:
      write(bf, hdr.key, 1)
      copyto("CPU", hdr.key)
elif hdr.op == UPDATE:
  write(cache, hdr.key, hdr.vals)
  drop
```

Figure 14: Example template of key-value store.

Table 7: ClickINC supported function list

kind	function and operations
Python built-in	min(), max(), sum(), abs(), pow(), round(), range(), len(), dict(), list(), +, -, *, /, %, //, <, >, ==, !=, ≤, ≥, =, &, |, ^, ~, <<, >>, and, or, not, in, not in.
ClickINC extension	ceil(), floor(), sqrt(), randint(), slice()

are: (1) whether convert the floating-point parameter to an integer one, which is decided by the accepted precision value in profile; (2) whether filters sparse parts of parameters according to "is_sparse" in profile; (3) the depth of aggregator (same for bitmap, counter, sequence). The code of MLAgg is described in Fig. 15.

DQAcc. DQAcc provides the SQL DISTINCT in-network acceleration, mainly relying on a hash-based rolling cache, i.e., multiple arrays to store historical value, and a recorder to roll each value to be replaced by new value (to approximate LRU). The configurable options are: (1) the depth and width of the cache; (2) the type of hash algorithms.

A.2 Profiles

A profile includes the following fields, and Fig. 16 shows an example profile for KVS template.

App. App is the dedicated ID corresponding to each template, i.e., "KVS", "MLAgg", "DQAcc". **Performance.** As also dedicated to templates, performance provides an optional interface for users to specify their performance requirements, as illustrated in Table 10. For KVS, it supports an objective function "max_hit_acc" to allow users to specify the performance preference over cache hit ratio and counting accuracy of heavy hitter, and also it allows for specifying demand on cache size; for MLAgg, the precision of parameter aggregation (decides whether the conversion from floating-point number to integer is feasible), the number of aggregators, and whether the parameter is sparse can also be specified.

Traffic distribution. For both template and user-written program, traffic distribution is required to provide the upper limit of the querying frequency (packet per second) of each client, in the format of {"client ID":"*pps", ⋯ }.

Packet format. The packet format also should be provided in the profile, where the traditional network packet header below UDP protocol can be abbreviated as a name, e.g., "ethernet/ipv4/udp", but the application protocol header should be described in detail, e.g., "key":"bit_128".

```
agg_seq_t = Array(row=1,size=Num_agg,w=width(hdr.seq))
bitmap_t = Array(row=1,size=Num_agg,  w=Num_worker)
agg_data_t = Array(row=len(hdr.vals), size=Num_agg, w=
     width(hdr.vals))
valid_t = Array(row=1, size=Num_agg, w=1)
hash_f = Hash(key=hdr.seq, ceil=Num_agg)
index = read(hash_f, hdr.seq)
seq = read(agg_seq_t, index)
isvalid = read(valid_t, index)
delete = 0, overflow = 0
if hdr.op == ACK:
  if isvalid and seq == hdr.seq:
    delete = 1
    forward(hdr)
else:
  if !isvalid and !hdr.overflow:
    write(agg_seq_t, index, hdr.seq)
    write(bitmap_t, index, hdr.bitmap)
    write(agg_data_t, index, hdr.data)
    write(valid_t, index, 1)
  elif seq == hdr.seq:
    bitmap = bitmap_t.read(index)
    if bitmap & hdr.bitmap == 0:
      vals = agg_data_t.read(key=index)
      new_vals = vals + hdr.data
      for i in range(vals):
        if new_vals[i] < 0:
          overflow = 1
          delete = 1
      new_bit = bitmap|hdr.bitmap
      if overflow:
        mirror(hdr={'bitmap':bitmap, 'data':vals,'
            overflow':1})
        forward(hdr)
      elif new_bit = 2^Num_worker-1:
        back(hdr={'op':REQ,'bitmap':new_bit,'data':
            new_vals})
        delete =1
      else:
        write(agg_data_t,index,new_vals)
        write(bitmap_t,index,new_bit)
        drop()
    else:
      forward(hdr)
if delete:
  del(agg_seq_t, index)
  del(bitmap_t, index)
  del(agg_data_t, index)
  del(valid_t, index)
```

Figure 15: Example template of MLAgg

```
{"app" : "KVS",
 "performance":
 {"objective function": max 0.7hit+0.3acc,
  "content": >=1000, ...},
  "traffic frequency": {c1: 10Mpps, c2: 20Mpps, ...},
  "packet_format":
  {"network": "ethernet/ipv4/udp",
   "khdr": {"key": "bit_128"},
   "vhdr": {"value_0": "bit_32"}, ...
} }
```

Figure 16: Configuration for KVS

Table 8: Basic functional unit list for IR

Operation	Explanation	Supported devices
_ram	1D-memory accessed by index	All
_cam	content-addressable memory	FPGA, NFP
_tcam	ternary-content-addressable memory	FPGA, NFP
_emt	stateless exact-match table	All
_semt	stateful exact-match table	FPGA, NFP
_tmt	stateless ternary-match table	All
_stmt	stateful ternary-match table	FPGA, NFP
_lpmt	longest-prefix-match table	All
_randint	achieve an integer random value	All
_crc	CRC series hashing calculation	All
_identity	identity-map hashing	Tofino series
_aes	AES series en(de)-crypto calculation	FPGA
_ecs	ECS series en(de)-crypto calculation	NFP
_checksum	csum16 calculation	All
_mirror	mirroring a packet	All
_multicast	multicasting packet	Tofino series, TD4

Table 9: Device capability abstraction

Classify of instructions	
$\mathcal{B}_{IN}$	Integer addition, subtraction; bit, logical operation; slicing.
$\mathcal{B}_{IC}$	Integer multiplication, division, modulus.
$\mathcal{B}_{CA}$	Floating-point arithmetic and other complex arithmetic.
$\mathcal{B}_{SO}$	Stateful array operations.
$\mathcal{B}_{EM}$	Exact-match table.
$\mathcal{B}_{SEM}$	Stateful exact-match table.
$\mathcal{B}_{NEM}$	(Ternary, LPM)-match table.
$\mathcal{B}_{SNEM}$	Stateful (ternary, LPM)-matching table.
$\mathcal{B}_{DM}$	Direct-match table.
$\mathcal{B}_{BPF}$	Drop, send, copyTo.
$\mathcal{B}_{APF}$	Mirror, multicast.
$\mathcal{B}_{AF}$	Hash functions (CRC8, CRC16, ...), checksum.
$\mathcal{B}_{CF}$	(En, De)-crypto.

A.3 Configuring a Template

The modules and templates usually need to allocate resources on devices, e.g., switch register memory. From the applications' perspective, the resource allocation influences the end users' performance. During the user program development, ClickINC has no idea about the runtime resource requirement; the users may not have the knowledge about how to allocate switch resources and their influences on the performance.

For certain applications, ClickINC can derive the resource requirements directly from the performance metric; for example, an MLAgg switch memory should equal to its bandwidth-delay product [30]. There are applications without the mathematical models to derive the resource requirements from the performance metric. ClickINC provides a learning-based approach: it maintains historical records of given parameter $\mathbf{x}$ and the performance $\mathbf{y}$, and learns the performance estimation function $\mathbf{y} = f(\mathbf{x})$ (e.g., $f(\cdot)$ could be a neural network and the learning method can be SGD). When a user submits a configuration with application performance metric, ClickINC searches for the parameter $\mathbf{x}$ with minimum resource allocation that satisfies the performance requirements $\mathbf{y}$.

$$\min_{\mathbf{y}=f(\mathbf{x})} g(\mathbf{x},\mathbf{y}), \quad s.t. \bigwedge_{i\in[1,k]} h_i(\mathbf{x},\mathbf{y}) \le 0, \tag{4}$$

```
1  Prog ::== Declare | Operation
2  Declare ::== header | parse | data | instance
3  header ::== h_type string {hBody}
4  hBody ::== struct {hFields}
5  hFields ::== type<length> string
6  type ::==  int | float | bit | bool
7  length ::== 1,2,...,1024
8  parse ::== cond? extract(hBody)
9  data ::== type string
10 instance ::== emt | semt| tmt | stmt | lpmt | cam |
       tcam | ram
11 Operation ::== cond? statement | statement
12 statement ::== data = operand | operand
13 operand ::== data calc | instance action
14 action ::== write | get | drop | mirror | multicast |
       randint | crc
15         | aes | ecs| calc
16 calc ::== + | - | * | / | % | bit operation | >>const
       | <<const
17 condition ::== state | state&&state | state||state
18 state ::== data compare
19 compare ::== > | >= | == | <= | <
```

Figure 17: IR instruction syntax

Table 10: INC profile

Template	KVS	MLAgg	DISAcc	OPSketch	DDoSAD
Performance	"max_hit_acc" : [0.7, 0.3], "depth" : >= 1000	"precision_dec" : 3 "is_sparse": 0, "depth": >= 500	"c_depth" : >= 1500 "c_len" : >=8	"c_depth" : >= 5 "c_len" : >=800	"c_depth" : >= 10 "c_len" : >=2000

where k is the number of performance metric constraints, $g(\mathbf{x},\mathbf{y})$ means the resource consumption, and $h_i(\mathbf{x},\mathbf{y})$ means the i-th dimension of performance metric is satisfied. The optimization problem can be solved using gradient descent.

A.4 Intermediate Representation

The syntax of IR is described in Fig. 17, where the *instance* and *action* are the basic functional units listed in Table 8. These units can be further utilized by network operator to write a new *object* and *primitive* module in Fig. 5 to update the library, which are provided to developers for programming with frontend language. Although the devices in the same architecture share some common constraints, they exhibit their exclusive features as well due to particular resource requirements, e.g., Trident4 supports the en(de)-cryption while Tofino does not. Therefore, to map instructions to the correct devices, we abstract the device capability in form of atomic operations (e.g., CRC calculation) that are listed in Table 8, and classify them into different types as shown in Table 9, which helps to rule out impossible mappings during allocation.

B THEORIES ON PLACEMENT

B.1 Analysis of Program Partitioning

To ensure the correctness of process on program partitioning and instruction block construction, we provide the following theory. First, we define the *partitioning legality* as:

Definition B.1. Given the partitions of IR program $\mathcal{P}$, $\forall p_1, p_2 \in \mathcal{P}$, there is no bidirectional traffic flow, i.e., $p_1 \nLeftrightarrow p_2$.

The partitioning legality ensures that any two partitions can be separately placed on different devices.

Program partitioning. The data in program is two kinds: (1) stateless data which is new for each round program execution and the data change will not affect the next packet, e.g., an intermediate variable; (2) stateful data, which is same for all packets and the data change affects the next packet, e.g., a cache table. To ensure the data consistency and correctness, stateful data cannot be duplicated. Thus, the instructions with operations on the same stateful data (we call them *state-sharing* instructions) cannot be partitioned on different devices, i.e.,:

Lemma B.2. *$\forall$ two instructions p_1 and p_2, if they are state-sharing, the partitioning legality is unsatisfied.*

Proof. Assume p_1, p_2 are placed on upstream device and downstream device respectively, if the stateful data is located on upstream device, then after p_1 is executed, the traffic flow to downstream device to execute p_2 which however needs to return to upstream device for accessing the stateful data, causing bidirectional traffic flow and violates partitioning legality; if the stateful data is located on downstream device, then traffic will flow to downstream device to access stateful data and return upstream device to complete p_1, and also violates partitioning legality. □

Thus, we need to group all state-sharing instructions together as an inseparable partition. Following this, we construct a directed graph for IR program as G, where we the vertex is inseparable state-sharing instruction partition or each other nornal instruction, and the edge to describe instruction dependency.

As long as two instruction has direct dependency (i.e., the next instruction uses the value generated by the previous instruction), we use an directed edge to connect them from previous instruction to the next one. For example, $p_1 \rightarrow p_2$ indicates the instruction p_2 directly depends on p_1. Obviously, instructions with direct dependency represents there exists data flow (the left value of p_1 flows to one of the right values of p_2), i.e., $p_1 \rightarrow p_2$ can infer that $p_1 \Rightarrow p_2$, based on which we have:

Lemma B.3. *The instruction p_2 depends on p_1 is equaling to $p_1 \Rightarrow p_2$.*

Proof. We first prove that p_2 depending on p_1 can infer $p_1 \Rightarrow p_2$. If p_1 has direct dependency with p_2, i.e., $p_1 \rightarrow p_2$, obviously it equals to $p_1 \Rightarrow p_2$; if p_2 indirectly depends on p_1, we assume there exists an instruction p_a that has direct dependency with p_1 and p_2, i.e., $p_1 \rightarrow p_a$ and $p_a \rightarrow p_2$. Then, we have $p_1 \Rightarrow p_a \Rightarrow p_2$, thus $p_1 \Rightarrow p_2$ and the statement is proved. Last, we should prove that $p_1 \Rightarrow p_2$ can infer that p_2 depends on p_1. If the left value of p_1 flows to p_2, obviously $p_1 \rightarrow p_2$; otherwise, we similarly assume an instruction p_a, and the left value of p_1 flows to p_a and p_a's left value flows to p_2, and thus we have $p_1 \rightarrow p_a$ and $p_a \rightarrow p_2$, i.e., p_2 indirectly depends on p_1, the Lemma B.3 is proved. □

Then, we have the following lemma:

Lemma B.4. *Directed acyclic IR dependency graph satisfies the partitioning legality.*

Proof. Assume that the acyclic dependency graph violates the partitioning legality, i.e., $\exists$ instructions p_1, p_2, $p_1 \Rightarrow p_2$ and $p_2 \Leftarrow p_1$, thus we have p_2 depends on p_1 and p_1 depends on p_2 respectively according to Lemma B.3. It means that p_1 and p_2 are cyclic in dependency, which is impossible for Directed acyclic graph (DAG). Therefore, the assumption is wrong and Lemma B.4 is proved. □

According to the above theory, we need to group the cyclic instruction on dependency graph as a hybrid vertex, so that becoming a IR DAG and partition legality can be always satisfied.

Instruction block. The instruction block construction process should also maintain the partition legality. In detail, given the IR DAG $G = (V, E)$, we define the predecessor set for each vertex $v \in V$ as $\mathcal{P}(v) = \{x \in V | < x, v >\in E\}$. We apply the Kahn's algorithm [18], a variant of Topological sorting on G to generate a series of the Kahn's partitions $\mathcal{K} = \{K_i\}_{i=1}^{N_K}$, where $V = \bigcup_{i=1}^{N_K}\{v | v \in K_i\}$ and $K_i \cap K_j = \emptyset (i \neq j)$. According to the Kahn's algorithm, for $\forall v \in K_i, i \in \{2, 3, \cdots, N_K\}$, $\mathcal{P}(v) \subset \bigcup_{l=1}^{i-1} K_l$ holds. That is, any predecessor vertex of a partition K must belong to a partition before K, which leads to the following lemmas.

Lemma B.5. *Given the Kahn's partitions $\mathcal{K} = \{K_i\}_{i=1}^{N_K}$ for the DAG $G(V, E)$, $\forall v_m \in K_i, v_n \in K_j$, if $i > j$, then $v_m \not\Rightarrow v_n$, where $\not\Rightarrow$ means a node cannot reach another node on the graph.*

Proof. Assume $\exists\, v_m \in K_i, v_n \in K_j, i > j$ to make $v_m \Rightarrow v_n$ hold. Then $v_m \in K_i$ is a predecessor vertex of $v_n \in K_j$ (i.e., $v_m \in \mathcal{P}(v_n)$), which means it is impossible for $K_i \subset \bigcup_{l=1}^{i-1} K_l$. Therefore, the assumption is wrong and Lemma B.5 is proved. □

Lemma B.6. *Given Kahn's partitions $\mathcal{K} = \{K_i\}_{i=1}^{N_K}$ for the DAG $G(V, E)$, $\forall v_m, v_n \in K_i$, if $m \neq n$, then $v_m \not\Rightarrow v_n$.*

Proof. Assume that $\exists v_m, v_n \in K_i, i \neq j$ makes $v_m \Rightarrow v_n$. Then $v_m \in K_i$ is the predecessor vertex of $v_n \in K_i$ (i.e., $v_m \in \bigcup_{l=1}^{i-1} K_l$), which contradicts with the assumption of $v_m \in K_i$. Therefore, Lemma B.6 is proved. □

The following theorem is derived from the lemmas:

Theorem B.7. *Given Kahn's partitions $\mathcal{K} = \{K_i\}_{i=1}^{N_K}$ for the DAG $G(V, E)$, $\forall v_m \in K_{i-1}, v_n \in K_i$, if $< v_m, v_n >\in E$, then no $v_l \in V (l \neq m, n)$ can make $v_m \Rightarrow v_l$ and $v_l \Rightarrow v_n$.*

Proof. Assume $\exists v_l \in K_j (l \neq m, n; j \in [1, N_K])$ that makes $v_m \Rightarrow v_l$ and $v_l \Rightarrow v_n$ hold. We know $j \neq i - 1$ and $j \neq i$ from Lemma B.6. Then if $j < i - 1$, Lemma B.5 tells us that $v_m \in K_{i-1} \not\Rightarrow v_l \in K_j$, which violates the assumption. Similarly, if $j > i$, $v_l \in K_j \not\Rightarrow v_n \in K_i$ also contradicts with the assumption. Hence, j does not exist and Theorem B.7 is proved. □

B.2 Analysis of Device Equality

Starting from the initial device status that devices at a layer in the same pod (we call them peer devices subsequently) are exactly equal in resources, we prove that these peer devices can maintain equality under our allocation algorithm.

Spine-leaf topology. Each leaf is connected with all the same spine switches, and any path is the leaf-spine-leaf structure sharing the common spines. Thus, it's straightforward that all spines should be allocated with the same part of an INC program and regarded as the same device.

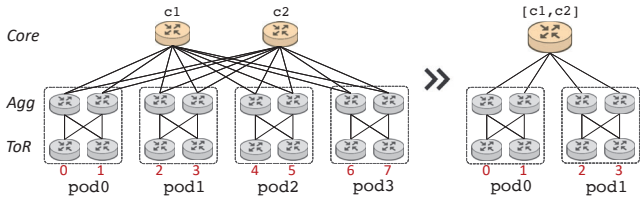


Figure 18: Example of full-clos fat-tree topology.

Full-clos Fat-tree topology. For a full-clos fat-tree topology, each switch in a pod is fully connected with each of upper-layer switches which should have a higher throughput capacity, as illustrated in Fig. 18. In this case, the core switches are also fully shared by all Agg switches, which is similar to spine-leaf topology and thus can also be reduced as the same device, as illustrated in the right sub-figure of Fig. 18.

Then, we should infer the equality of Agg switches in a pod. First, we denote the INC program as instruction set $[0, n]$ which should be allocated along path pod0-pod1, and we assume program placed on core switches are $[i, j], 0 \leq i \leq j \leq n$. Then $[0, i)$ should be placed on switches in pod0, and $(j, n]$ needs to be placed on switches in pod1. Supposing the ToR0 switch in pod0 is allocated with instructions $[0, p]$, as ToR0 connects with all Agg switches in pod0, these Agg switches must be placed the same instructions $(p, n]$, making other ToR switches e allocated with $[0, p]$ correspondingly. Thus, the equality of switches at the same layer in a pod is proved.

Device-equal Fat-tree topology. As illustrated in Fig. 19, this topology targets that device of each layer has the same throughput capacity. A k-fat-tree has k pods and $(\frac{k}{2})^2$ core switches, and each layer in a pod has $\frac{k}{2}$ switches. In this topology, each Agg switch in a pod fully connects with the $\frac{k}{2}$ core switches, which means these core switches are shared by the current pod and can be reduced as a device. Supposing the traffic is from pod0 to pod1, then we can derive the topology as the right sub-figure shows in Fig. 19.

Thereafter, we need to prove the equality of Agg devices and ToR devices in a pod. First, we still assume a instruction set $[0, n]$ to be placed along path pod0-pod1. For switches in pod0, we suppose the placement is $[0, p0]$ on ToR0, $[0, p1]$ on ToR1, $[q0, k0]$ for Agg0, and $[q1, k1]$ for Agg1. As ToR0 and ToR1 are both fully connected with Agg0 and Agg1, we have $p0 = p1 = q0 = q1$, and the case for switches in pod1 is similar. Thus ToR switches in the same pod can also be reduced as a device. Then we can derive the topology shown as the left-below sub-figure in Fig. 19, i.e., multiple paths diverge from the same ToR device in pod0 and converge at pod1. Fortunately, we can notice that the multiple paths are exactly the same regardless of device type, available resources. Thus, for any non-random allocation algorithm, the instructions placements on these paths are absolutely same, i.e., the allocated instructions on the Agg switches in pod0 are exactly same, and so are core switches and Agg switches in pod1. That means these switches can be reduced to a single device respectively, and the topology shown as the left-below sub-figure in Fig. 19 converts to a chain. Thus, the equality of switches at the same layer in a pod is also proved.

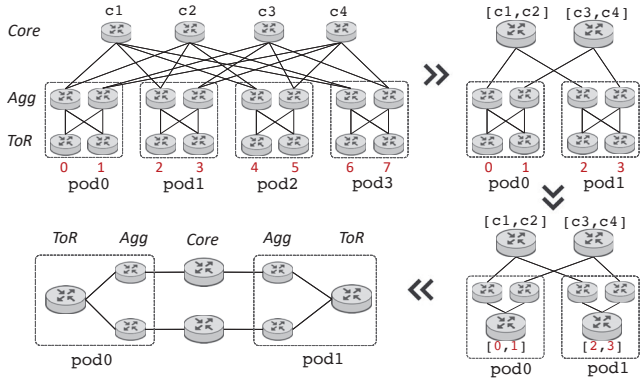


Figure 19: Example of device-equal fat-tree topology.

C PSEUDO ALGORITHMS

This section describes the core algorithms for program placement.

C.1 Block construction

Block construction is described in Algorithm 3.

C.2 Program merging

The program merging process is described in Algorithm 4.

D DEVICE MODELING AND CHIP RESOURCE CONSTRAINTS

The architectures of programmable network devices are mainly pipeline and run-to-complete (RTC). Some devices, e.g., Netronome smartNIC and FPGA, can implements both pipeline and RTC, and we call it hybrid device. ClickINC covers the resource constraints of four major kinds of programmable chips: Tofino series ASIC, Trident 4 switch ASIC, Netronome Network Processor, and Xilinx FPGA. The constraints for other programmable chips can be modeled similarly. Please refer to the material: http://arxiv.org/abs/2307.11359 for the detailed chip resource constraints.

E DEPLOYMENT CONSTRAINTS

Each block v should be allocated only once, and each instruction in the block should be deployed:

$$\bigwedge_{v \in V} \Big[\sum_{d \in D} x_{v,d} \bigwedge_{p \in v} \big(\bigvee_{s \in S_d} a_{p,s} \big) = 1 \Big] \tag{5}$$

where $x_{v,d}$ indicates whether primitive block v is deployed on device d, and $a_{p,s}$ denotes whether primitive p is deployed on stage s.

Since the application throughput is bottlenecked at the device with the minimal bandwidth, given the throughput requirement H, we have the constraint:

$$\bigvee_{l \in L} \Big[\bigwedge_{d \in D[l]} (h(d) \geq H[l]) \Big] \tag{6}$$

where $h(d)$ is the bandwidth of device d.

Typically the application flow has a fixed forwarding path, which raises two topology constraints: deployment scope T_s (i.e., the

Algorithm 3: Instruction block construction

Input: Primitives IR DAG $G = (V, E)$.
Output: the results of blocks G_{out}.

$\mathcal{K} \leftarrow$ Kahn_partition(G);
$G_1 \leftarrow$call intra_partition(G);
$G_{out} \leftarrow$call inter_partition(G_1);
return G_{out};
Function intra_partition(G):
  $V_1 \leftarrow \emptyset, V \leftarrow G.V; E_1 \leftarrow G.E;$
  for $v_1 \in V$ **do**
    P $\leftarrow \{v_1\}$; remove v_1 from V;
    for $v_2 \in V$ **do**
      if $v_2.type = P.type$ **then**
        if $\mathcal{K}(v_2) = \mathcal{K}(P)$ **then**
          if $\mathcal{P}(P) \cap \mathcal{P}(v_2) \neq \emptyset$ **then**
            add v_2 to P; remove v_2 from V;
            combine in-edges of P, v_2 in E_1;
    add P to V_1;
  return $G_1 = (V_1, E_1)$;
Function inter_partition(G_1):
  $\mathcal{K} \leftarrow Kahn\_partition(G_1)$;
  $V_2 \leftarrow G1.V, E_2 \leftarrow G_1.E, V \leftarrow \emptyset$;
  while $|V| < |V_2|$ **do**
    $V_2 \leftarrow V, V \leftarrow \emptyset$;
    for *i from 0 to* $|\mathcal{K}| - 1$ **do**
      for $v_1 \in \mathcal{K}[i]$ **do**
        P $\leftarrow \{v_1\}$, remove v_1 from $\mathcal{K}[i]$;
        S$\leftarrow successor(v_1) \cap \mathcal{K}[i+1]$;
        for $v_2 \in S$ **do**
          if $\mathcal{K}[i+1]$ **then**
            add v_2 to P;
            remove v_2 from $\mathcal{K}[i+1]$;
            remove $< v_2, P >$ from E_2;
        $addPtoV$;
    $\mathcal{K} \leftarrow Kahn\_partition(G_2 = (V, E_2))$;
  return $G_2 = (V_2, E_2)$;

Algorithm 4: Program merging

Input: the parsing graph of INC program and main program T_{inc}, T_{main}; the dependency graph of INC program and main program G_{inc}, G_{main}.
Output: the whole parser and program T_w, G_w.

$T_w \leftarrow T_{main}, G_w \leftarrow G_{main}$;
call Parsing_merger(T_{inc}, T_w);
call Program_merger(G_{inc}, G_w);
Function Parsing_merger(T_{inc}, T_w):
  for s ***in*** $T_{inc}.traversing$ **do**
    $t \leftarrow T_w.find(s), p \leftarrow T_w.find(s.parent)$;
    if $t =$ *None* **then**
      $add\_son(p, s), add\_annotation(s)$;
      $add\_transition(p, s), add\_annotation\_in(p)$;
      $add\_hdr(s.hdr), add\_annotation(s.hdr)$;
    else $add\_annotation(t)$;
Function Program_merger(G_{inc}, G_w):
  if $d \in$ *Pipeline* **then**
    $C_{inc} \leftarrow chain(G_{inc}), C_w \leftarrow chain(G_w)$;
    for s ***in*** C_{inc} **do**
      $p \leftarrow get\_ins\_position(s, C_w)$;
      $C_w.insert(p, s), add\_annotation\_before(s)$;
  else
    $G_{whole} \leftarrow merge\_DAG(G_{inc}, G_w)$;
    $L \leftarrow Topological\_sort(G_{whole})$;
    for e ***in*** G_{inc} **do**
      $p \leftarrow get\_level(e, L)$;
      $G_w.insert(p, s), add\_annotation\_before(s)$;

blocks can only be allocated on devices along the path), and deployment direction (i.e., the block execution sequence should match the packet forwarding direction). The scope constraint is:

$$\sum_{v \in, d \notin T_s} x_{v,d} = 0 \tag{7}$$

and the direction constraint is:

$$\bigwedge_{d_i, d_j \in D; v_k, v_l \in V} (F_{d_i,d_j} R_{v_k,v_l} x_{v_k,d_i} x_{v_l,d_j} \geq 0) \tag{8}$$

In the equation, F_{d_i,d_j} denotes the deployment direction: 1 represents the forwarding direction, -1 vice versa, and 0 means no direction needs to be enforced (e.g., for an FPGA-based acceleration card attached to a switch); R_{v_k,v_l} denotes the dependency between two blocks: 1 represents that v_l relies on v_k, -1 vice versa, and 0 means v_k and v_l are independent.

Similarly, dependent blocks on the same device should conform to the pipeline direction:

$$\bigwedge_{s \in S_d; p_i, p_j \in v} (R_{p_i,p_j} a_{p_i,s} a_{p_j,s} = 0) \tag{9}$$

The constraint ensures that no stage overlap occurs in the case that v_j depends on v_i.

Furthermore, the dependent primitives cannot be placed on the same pipeline stage. For Tofino series chips, there is a particular circumstance: a non-matching-table primitive can be placed in the same stage with the matching table that it depends on, to construct a match-action structure as long as they share the same conditional statement. We use R_{p_i,p_j} to denote the dependency between p_i and p_j, where 1 represents p_j depends on p_i, -1 vice versa, and 0 means p_i and p_j are independent. The primitive dependency constraint is:

$$\begin{aligned} &\bigwedge_{s \in S_d; p_i, p_j \in v} \left[(R_{p_i,p_j} a_{p_i,s} a_{p_j,s} = 0) \right] \\ &\bigwedge_{d \in Tofino} R_{p_i,p_j} a_{p_i,s} a_{p_j,s} (1 - case) = 0 \end{aligned} \tag{10}$$

where $case$ is $(R_{p_i,p_j} = 1) \wedge (p_i \in \mathcal{B}_{EM} + \mathcal{B}_{NEM}) \wedge (p_j \notin \mathcal{B}_{EM} + \mathcal{B}_{NEM}) \wedge (cond(p_i) = cond(p_j)))$, and $cond(\cdot)$ denotes the requirement on conditional statement. Through the above mapping, primitives satisfied with the constraint can be constructed as match-action table structure.

Pipeline switch have separate pipelines ξ_{ig} and ξ_{eg} for ingress and egress, respectively (i.e., $S_d = \xi_{ig} + \xi_{eg}$). The instructions related to forwarding decision can only be deployed at ingress. Denoting these instructions as $\mathcal{F}_{fd}$, we have:

$$\sum_{p \in \mathcal{F}_{fd}, s \in \xi_{eg}} a_{p,s} = 0 \tag{11}$$