Using Trio – Juniper Networks' Programmable Chipset – for Emerging In-Network Applications

Mingran Yang[†], Alex Baban[*], Valery Kugel[*], Jeff Libby[*], Scott Mackie[*],

Swamy Sadashivaiah Renu Kananda[*], Chang-Hong Wu[*], Manya Ghobadi[†]

[†]Massachusetts Institute of Technology [*]Juniper Networks

ABSTRACT

This paper describes Trio, a programmable chipset used in Juniper Networks' MX-series routers and switches. Trio's architecture is based on a multi-threaded programmable packet processing engine and a hierarchy of high-capacity memory systems, making it fundamentally different from pipeline-based architectures. Trio gracefully handles non-homogeneous packet processing rates for a wide range of networking use cases and protocols, making it an ideal platform for emerging in-network applications. We begin by describing the Trio chipset's fundamental building blocks, including its multi-threaded Packet Forwarding and Packet Processing Engines. We then discuss Trio's programming language, called Microcode. To showcase Trio's flexible Microcode-based programming environment, we describe two use cases. First, we demonstrate Trio's ability to perform in-network aggregation for distributed machine learning. Second, we propose and design an in-network straggler mitigation technique using Trio's timer threads. We prototype both use cases on a testbed using three real DNN models (ResNet50, DenseNet161, and VGG11) to demonstrate Trio's ability to mitigate stragglers while performing in-network aggregation. Our evaluations show that when stragglers occur in the cluster, Trio outperforms today's pipeline-based solutions by up to 1.8×.

CCS CONCEPTS

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

KEYWORDS

Network hardware design, Programmable dataplanes, Network support for machine learning

ACM Reference Format:

Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, Manya Ghobadi. 2022. Using Trio – Juniper Networks' Programmable Chipset – for Emerging In-Network Applications. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3544216.3544262

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544262

1 INTRODUCTION

Data-intensive applications are the foundation of today's online services. With the gradual slowdown of Moore's law, hardware accelerators are struggling to meet the performance demands of emerging cloud applications, such as machine learning, databases, storage, and data analytics. Further advances are significantly limited by the amount of computation and memory that can fit in a single server, driving the need for efficient distributed systems for data-intensive applications.

The availability of programmable switches, such as Intel's Tofino [2, 20, 22], has created opportunities to design new packet-processing protocols and compilers [17, 20, 24, 44, 45, 58, 69, 71]. Tofino switches have also paved the way to using *in-network computing* [23, 60, 74] to accelerate applications such as caching [43], database query processing [50, 73], machine learning training [36, 48, 55, 63, 77], inference [76], and consensus protocols [27, 28, 52]. The key idea of in-network computing is to leverage the switches' unique vantage point to perform part of the computation directly inside the network, thereby reducing latency and improving performance.

Although programmable switches have been crucial enablers of this new paradigm, the Protocol Independent Switch Architecture (PISA) [2, 20, 22, 58] is often a poor fit for emerging in-network applications, thus limiting further growth and precluding the widespread adoption of in-network computing applications [35, 37, 67].

This paper presents Trio's programmable architecture for in-network computing. Trio is Juniper Networks' programmable chipset with a multi-billion dollar pre-existing customer base. It has been deployed in hundreds of thousands of routers and switches worldwide in the core, edge, and datacenter environments. The Trio chipset has been used in production devices for over a decade.

Trio is built on a set of customized processor cores, with an instruction set optimized for networking applications. As a result, the chipset has the performance of a traditional ASIC, while enjoying the flexibility of a fully programmable processor by allowing the installation of new features via software. Trio's flexible architecture enables it to support features and protocols developed long after the chipset is released. Trio processor cores have access to a high-performance large memory system to store data and state related to system configuration and packets. The memory system is central to the scalability of emerging applications with large memory footprints.

Trio's architecture is fundamentally different from that of Tofino. Trio has a non-pipelined architecture, so different packets do not

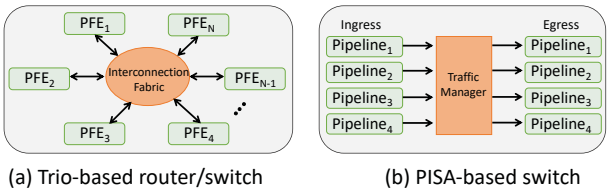

Figure 1: High-level comparison of a Trio-based router/switch and a PISA-based switch.

necessarily flow through the same physical paths on the chip. Incoming packets in Trio are processed independently using thousands of parallel threads (details in §2). These threads use a run-to-completion model [12, 70], in which a thread will execute as many instructions as are needed to complete the processing for the packet it is currently working on. Trio has dedicated logic to ensure packets of the same flow are delivered in order, but packets of different flows can be processed out of order, allowing it to efficiently handle a mix of concurrent applications.

Consequently, Trio can gracefully handle different packet processing rates: it can support lower than line-rate for applications that require rich per-packet processing while maintaining line-rate for applications with simple per-packet processing needs. In contrast, PISA-based switches force all packets to traverse the same set of pipeline stages, independent of the application. P4 programs [19] have an all-or-nothing fate, wherein programmability is sacrificed for line-rate packet processing since PISA-based switches are unable to support flexible packet processing rates.

In this paper, we first describe the fundamental building blocks of the Trio chipset, including details of its packet processing engines and the surrounding memory systems (§2). Next, we describe Trio's programming language, called Microcode (§3). We then use in-network aggregation for machine learning training as the first use case to explain Trio's flexible Microcode design (§4). We introduce in-network straggler mitigation as a second use case to demonstrate Trio's unique ability to launch efficient timer-based threads (§5). We demonstrate that implementing straggler mitigation in Trio is straightforward, while, to the best of our knowledge, enabling *efficient* straggler mitigation inside PISA-based devices is challenging, if not impossible.

We implement both use cases on a testbed with a Juniper MX480 device [10], one 64×100 Gbps Tofino switch, and six ASUS ESC4000A-E10 servers, each with one A100 Nvidia GPU [11] and one 100 Gbps Mellanox ConnectX5 NIC. We train three DNN models (ResNet50 [41], DenseNet161 [42], and VGG11 [68]) to demonstrate Trio's ability to mitigate stragglers while performing in-network aggregation. Our evaluations show that when stragglers occur in the cluster, Trio outperforms SwitchML [63], the state-of-the-art in-network aggregation platform, by up to 1.8× (§6).

Juniper Networks continues to evolve the Trio chipset for higher bandwidth, lower power, and additional functionalities for existing and emerging applications, while also developing software infrastructures to support more use cases. We invite the networking community to identify novel use cases that will leverage Trio's programmable architecture.

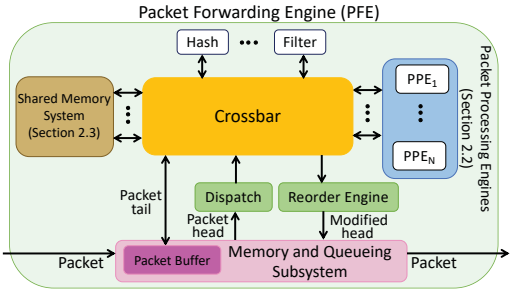

Figure 2: Trio's Packet Forwarding Engine (PFE) architecture. Each PFE has hundreds of multi-threaded Packet Processing Engines (PPEs).

2 TRIO'S ARCHITECTURE

Since its introduction in 2009, the Trio chipset has gone through six generations [16] with various performance points and architectures. This section provides a detailed overview of Trio's recent architecture. First, we give a high-level overview of packet forwarding and processing in a Trio-based router[1] (§2.1). We then turn to the details of Trio's packet processing engines (§2.2). Finally, we explain Trio's various memory types and read-modify-write operations (§2.3).

2.1 Trio-based Router Architecture

Figure 1 illustrates the high-level differences between a Trio-based router (or switch) and a PISA-based switch. There are two important components of every Trio-based device: (*i*) Packet Forwarding Engines and (*ii*) Packet Processing Engines, described below.

Packet Forwarding Engine (PFE). PFEs are the central processing elements of Trio's forwarding plane and are used to systematically move packets in and out of the device. A Trio-based device consists of one or more PFE. Depending on the generation, each Trio chipset supports a different packet processing bandwidth. Trio's first-generation PFEs supported 40 Gbps of network bandwidth with multiple chips. Today, Trio's sixth-generation PFE supports 1.6 Tbps in a single chip. A small router may have only a single PFE, while larger routers have multiple PFEs connected by an *interconnection fabric*, as shown in Figure 1(a). By providing any-to-any connection between the PFEs, the interconnection fabric expands the bandwidth of a device much farther than a single chip could support. Each PPE handles packets in both the ingress and egress directions. Packets arrive at the system through an ingress PFE and exit through an egress PFE.

Packet Processing Engine (PPE). Each PFE has hundreds of multi-threaded *Packet Processing Engines* (*PPEs*), as shown in Figure 2. Each PPE supports tens of threads working on different packets at the same time. Unlike Tofino's architecture, where pipelines cannot access each other's registers, PPE threads within one PFE can share state efficiently via shared memory. Section 2.2 explains the PPE's thread-based design in more detail.

Parallel packet processing. PFE's hardware logic automatically divides each incoming packet into *head* and *tail* parts (analogous to PISA's header and payload). The packet head is the first

[1] In this paper, we use the term router and switch interchangeably. Historically, Juniper Networks' devices are called routers.

part of a packet and is usually large enough to hold all of the packet headers needed to process the packet (the size of the packet head is different for each generation of the Trio devices but is typically around 200 bytes). The tail consists of the remaining bytes of the packet (if any). When a new packet arrives, a hardware module inside PFE, called the *Dispatch* module, sends the packet head to a PPE for processing based on availability, and the PPE spawns a new thread for this packet head. Packet tails are held in the PFE's Packet Buffer in the Memory and Queueing Subsystem to avoid storing a large number of bytes in the PPE threads. By default, each thread works on a single packet. Many PPE threads work in *parallel* to provide the required processing bandwidth.

Reorder Engine. When packet processing is completed, the modified packet head is sent to a *Reorder Engine*. The Reorder Engine holds the updated packet head until all earlier arriving packets in the same flow have been processed to ensure in-order delivery. The Reorder Engine then sends the modified packet heads to the Memory and Queueing Subsystem to be enqueued for transmission.

2.2 Packet Processing Engine

Trio's PPEs provide capabilities that are difficult or impossible to achieve with fixed processing pipelines or existing specialized processing units. Each PPE is a VLIW (Very Long Instruction Word) multi-threaded Microcode engine core. Each micro-instruction controls multiple ALUs, operand and result selection, and complex multi-way branching. The complexity of the work needed to execute a micro-instruction means each instruction takes multiple clock cycles. Because each PFE typically serves many packets at the same time, one PPE does not require high single-thread performance. Each thread in Trio has only one datapath instruction at a time. Trio does not dispatch an instruction on the same thread as the previous instruction into the PPE pipeline until the latter exits the pipeline. Hence, there is no need to pass data between instructions on the same thread because subsequent instructions do not depend on the results from the previous ones before the data writebacks are completed.

PPE threads. A PPE thread is usually started when a packet head arrives at the PPE and destroyed when the processing is complete for that packet at this PPE. The thread destruction is automatically handled by the hardware logic in the chip, although the programmer has control as to when to give up the execution of a thread. Threads can also start in response to certain internal events, including statistical collection and timers (more details in §5). External events have the ability to spawn the execution of new threads through similar mechanisms. Together, the PPEs in the ingress and egress PFEs handle all functions needed to process a packet (e.g., packet parsing, route lookup, packet rewriting).

Per-thread local storage. Each PPE has two main forms of internal storage. First, each thread has a dedicated pool of local memory (1.25 KBytes). The local memory can be accessed on any byte boundary, using either pointer registers or an address contained in the micro-instruction. Before a PPE thread is initiated, the packet head is loaded into the local memory of that thread. When an outgoing packet is being sent, the modified packet head is unloaded from the thread's local memory. The use of pointer registers allows efficient access to packet headers, as well as to other types of data structures. Second, each thread has 32 64-bit general-purpose registers that are private to it. The local storage (memory and registers) holds information specific to the packet being processed. Shared state across packets is held in the Shared Memory System accessible to all PPEs.

ALU types. There are two ALU types: (*i*) *Condition* ALUs and (*ii*) *Move* ALUs. Condition ALUs are used for arithmetic or logical operations to produce 32-bit data results and/or for comparative operations to produce 1-bit condition results. Move ALUs produce 32-bit results that can be written into either a register or local memory. The results from the Condition ALUs can be used as inputs to the Move ALUs. This ALU organization allows the resources of each instruction to be flexibly allocated between sequencing control (described next) and generation of logical/arithmetic results to be stored in registers/memory. Importantly, each ALU operand and each Move ALU result can be a bit-field of arbitrary length (up to 32 bits) and an arbitrary bit offset. This has two main benefits. First, it improves the efficiency of accessing fields of varying sizes in a packet header. Second, it improves the utilization of memory and register capacity, allowing each piece of data to use only the bits it needs. Trio has ALUs in both the PPEs and the Shared Memory System. The former are used for operations on registers and local memory, while the latter is used for operations on data stored in the Shared Memory System. Operations on a packet tail is also supported by moving sections of the packet tail to the local memory of the PPE thread.

Sequencing logic. The condition result(s) from one or more Condition ALU can be used by a sequencing logic unit to select the next micro-instruction to be executed. Each micro-instruction includes the address of a target block of one to eight micro-instructions. Any or all of the condition results can be ignored, and the combination of the condition results used is highly flexible. Much of the work done in packet processing involves complex conditional branches in the code, especially during parsing. Trio's ability to perform a complex multi-way branching in a single instruction is well-matched to the needs of packet processing applications. The PPE supports a call-return mechanism to subroutines, which can be nested up to eight levels deep.

Efficient hash calculation. Efficient load-balancing is an important requirement of all routers/switches. In a Trio-based system, a Microcode program is responsible for specifying which packet fields are included in the hash calculation. This allows complete flexibility as to which packet fields contribute to the load balancing decision, including the ability to select fields from packet headers whose protocols have not yet been invented. The hash function in Trio is a high-quality hash function implemented using dedicated logic. As a result, the hash function implementation is more efficient than a comparable hash function implemented in software. The combination of programmable field selection and hardwired hash function gives PPEs an unprecedented balance of flexibility and efficiency.

Flexible programming. There is no fixed limit on the number or type of headers that can be processed by a PPE. Hence, a PPE can easily create new headers or consume/remove existing headers in packets using Trio's Microcode program (§3). As new protocols are developed, the Trio packet processing architecture can adapt by enhancing the software that runs on the PPEs. PPEs can

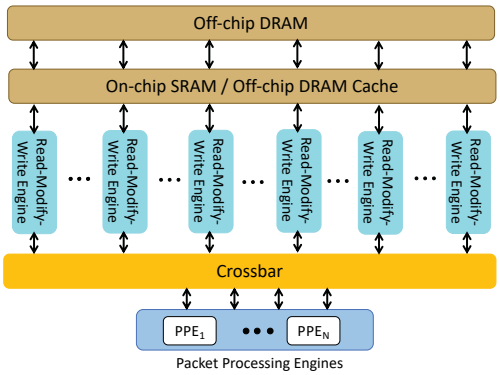

Figure 3: Trio's Shared Memory System.

also create or consume packets to accomplish tasks, such as keep-alive functions, at a much higher rate than can be supported by a control plane CPU because of PPEs' multi-threaded architecture. Importantly, processing cycles are fungible between applications, enabling graceful handling of the packet processing requirements of different applications. As a result, Trio-based systems can provide lower packet rates for applications with richer packet processing and higher packet rates for those with simpler packet processing, or a mix of the two.

2.3 Shared Memory System

Recent Trio chipsets support several GBytes of memory in each PFE. This section gives an overview of Trio's Shared Memory System.

Advantages of shared memory. For switches and routers, some data structures, such as counters and policers, need to be modified at a high rate. To support efficient access to these data structures by hundreds of PPE threads, Trio's Shared Memory System serves as the place for all threads to access and modify the data. All data accesses (read, write, and read-modify-write) to the Shared Memory System are processed by the read-modify-write engines, located close to the Shared Memory System. When multiple threads access the same memory location at around the same time, there is no need to move data from one thread to another. Instead, data modification happens inside the read-modify-write engines. This allows high-speed data updates near the memory and nicely meets the needs of packet processing applications. In contrast, the cache-line-based coherency model used by conventional processors requires data to be moved to the thread during access; this creates longer delays when multiple threads try to modify the same memory location. Although this model can support more complex and general operations on the data, it performs poorly for data structures that can be accessed by hundreds of threads.

Memory types. The Trio memory system is optimized to provide a high access rate for relatively small (8 bytes) requests. To achieve the required combination of bandwidth, latency, and capacity, the memory system uses two types of memory, shown in Figure 3: (*i*) a high-bandwidth on-chip memory, with approximately 70 ns access latency from the PPE; and (*ii*) a large high-bandwidth DRAM-based off-chip memory, with approximately 300 ns to 400 ns access latency from the PPE. The on-chip memory is implemented by a heavily multi-banked SRAM and is typically used for frequently-accessed data structures. The off-chip memory has a multi-megabyte on-chip cache which is similar to the on-chip SRAM and is heavily multi-banked to provide high throughput. The size of the On-chip SRAM and the Off-chip DRAM cache are software configurable (typically 2-8 MBytes and 8-24 MBytes, respectively). The Off-chip DRAM is several GBytes. The on- and off-chip memories are architecturally equivalent and exist in different ranges of a single unified address space. They only differ in capacity, latency, and available bandwidth. This allows data structures to be placed in the type of memory that best matches their capacity and bandwidth requirements.

Memory transactions. The memory system supports read and write operations of varying sizes, from 8 bytes up to 64 bytes (in 8-byte increments). Trio can support full memory system bandwidth with 8-byte accesses. In addition, a rich variety of read-modify-write operations are supported, including Packet/Byte Counters, Policers, Logical Fetch-and-Ops (And/Or/Xor/Clear), Fetch-and-Swap, Masked Write, and 32-bit add. The read-modify-write operations are enabled by *read-modify-write engines*, as specified below.

Read-modify-write engines. Packet processing requires extremely high-rate read-modify-write operations. Processing a single packet may involve updates to multiple counters, operations on one or more policers, and other operations as needed by the application. A naive approach to handle read-modify-write operations is to give one thread *ownership* of a memory location while the operation is carried out. But this approach cannot meet the high efficiency requirements of packet processing. In contrast, Trio offloads the read-modify-write operations to its memory system, where a range of memory locations is handled by a single read-modify-write engine. If multiple requests to the same memory location arrive at around the same time, the engine processes the requests in sequence, guaranteeing consistency of the updates. There is no need to issue explicit coherence commands to a location in memory when mixing read, write, and read-modify-write operations. Each read-modify-write engine processes memory requests at a rate of 8-byte per clock cycle. Hence, a single read-modify-write engine for the entire Shared Memory System cannot provide the memory bandwidth needed to process packets at a sufficiently high rate. To address this challenge, Trio supports several banks of SRAM and off-chip cache with their own read-modify-write engine, enabling the read-modify-write processing bandwidth to scale with the raw memory bandwidth.

Crossbar and shared memory performance. Trio's Crossbar is designed to support all read-modify-write engines, such that the Crossbar itself will never limit the memory performance. If the load offered to a given read-modify-write engine exceeds the 8-bytes per cycle throughput, there will be backpressure through the Crossbar. Juniper Networks increased the number of read-modify-write engines in each generation of Trio chips so that the memory bandwidth increases with the packet processing bandwidth.

3 TRIO'S PROGRAMMING ENVIRONMENT

This section gives an overview of Trio's programming environment. Section 3.1 describes Trio's programming language and the

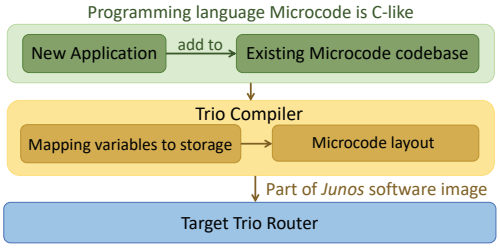

Figure 4: Programmer uses a C-like language called Microcode to program new applications and configure the target Trio routers.

toolchain for programming Trio-based devices. Section 3.2 provides a packet filtering example programmed in Trio Microcode.

3.1 Trio's Programming Language and Toolchain

The programming language for Trio-based devices is a C-like language called Microcode. The programmer implements all packet processing operations in Microcode, including packet parsing, route lookup, packet rewriting, and in-network computations (if any). Figure 4 shows the tools needed to program new applications on Trio. To program a new application on Trio, the programmer uses the Microcode language to write new applications and adds the new Microcode program to the existing codebase. Then the programmer uses Trio's compiler to generate the software image and configures the target device.

Expression syntax. Microcode supports C-style expressions. The supported variable types include scalar (label, bool, and integers in various sizes) and compound (struct and union). Microcode also supports pointers and arrays, conditions, function calls and gotos, and switch statements.

Instruction boundary. A Microcode program has multiple instructions. A single Microcode instruction can perform limited operations, and the programmer needs to explicitly specify the instruction boundaries. Typically, a single Microcode instruction can perform four registers or two local memory reads, and two registers or two local memory writes.

Variable storage classes. When defining a new variable in Microcode, the programmer needs to specify the location to store the variable. There are three types of variable storage classes: memory (PPE's local memory and registers), bus (indicating that the variable serves as input to the ALUs), and virtual (indicating constant values). Access to data stored in the Shared Memory System, such as forwarding tables, is achieved via the external transactions specified below.

External transaction. PPEs can issue *external transaction* (XTXN) to other modules, such as the Shared Memory System, Hash lookup/insert/delete, high-performance Filters, and counter/policer blocks, over the Crossbar. These XTXNs can be either synchronous or asynchronous. In synchronous XTXNs, the PPE thread is suspended until the XTXN reply is received; in asynchronous XTXNs, the PPE thread continues running normally. PPEs can also fetch data from the packet tails through XTXNs. In this case, packet tails are sent from the Memory and Queueing Subsystem, pass through the Crossbar, and then arrive at the local memory of the PPE. An XTXN consists of a request by a PPE to a target and a reply sent by the target back to the PPE. The format of the XTXN depends on the target block. For instance, read requests sent to the Shared Memory System take memory address as the parameter, and the data is returned in the XTXN response register.

Compiler. To compile Microcode programs, the programmer uses a tool called *Trio Compiler* (TC). TC maps the source code for an instruction to the various resources the instruction can control, including mapping variables to their underlying storage and assigning instructions to Microcode memory inside PPEs. TC has characteristics of both compilers and assemblers. On the compiler side, TC supports the translation of high-level C-style expressions into hardware instructions. On the assembler side, TC source code must contain instruction delineation, whereby the programmer marks the beginning and end of blocks of code representing a single instruction. If the code designated to one instruction does not fit, TC fails the compilation because it cannot implement the requested actions across multiple instructions. TC does not have a separate compilation and linking phase. It requires the complete source code instead of individual modules to generate the binary. This binary contains data to initialize PPE resources such as Microcode memory and local memory. It also defines required symbols, such as the address in local memory where the packet header starts. This binary file serves as part of the *Junos*[2] software image used by Trio's ASIC driver for device initialization.

vMX Virtual Router. Juniper Networks is making a concerted effort to enable third-party access to programming Trio-based devices. As a first step to enable third-party access to Trio's functionalities, Juniper Networks developed the vMX Virtual Router [5]. vMX is a virtualized Universal Routing Platform and consists of a virtual control plane (VCP) and a virtual forwarding plane (VFP). The VCP is powered by the *Junos* operating system, and the VFP runs the Microcode engine optimized for x86 environments. vMX is available as licensed software for deployment on x86-based servers and cloud services, such as Amazon Web Services.

Advanced Forwarding Interface. In Trio, packet forwarding is a sequence of operations executed by a PFE. Each operation can be represented by a node on a graph of potential packet forwarding operations. The PFE executes a series of operations for an individual packet based on its type/fields. Juniper Networks' Advanced Forwarding Interface (AFI) [3] provides partial programmability by allowing third-party developers to control and manage a section of this forwarding path graph via a small virtual container called a *sandbox*. The sandbox enables developers to add, remove and change the order of operations for specific packets.

3.2 Microcode Program Example

We illustrate the usage of Trio Microcode by showing an example of a filtering application whose function is to forward all incoming IP packets with no optional headers and drop all non-IP packets and IP packets with options.

Microcode program workflow. Figure 5 shows the Microcode program workflow. Each incoming packet is processed by one PPE

[2]Juniper Networks' *Junos* is the operating system that powers Trio-based devices.

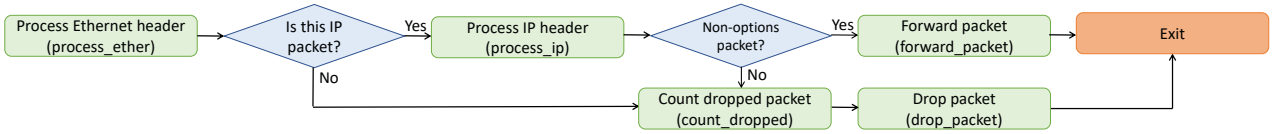

Figure 5: Microcode program workflow of the filtering application. Labels in brackets are the corresponding instruction names.

thread. The thread first looks at the packet's Ethernet header. If EtherType is equal to 0x0800, the packet is an IP packet, and the next step is to process the IP header; otherwise, it is a non-IP packet. In this case, the thread drops the packet and increments the Packet/Byte Counter for non-IP packets. For an IP packet, the thread further examines whether it has any IP option fields. The thread forwards all non-option IP packets, but drops IP packets with options and increments the Packet/Byte Counter for IP-option packets. After completing all required operations, the thread exits.

Packet header formats. Programmers need to define packet header structures in the Microcode program. The format of the packet header definition is similar to that of P4 [19], where each header is defined by an ordered list of field names with the corresponding field widths. Here, we show the definition of the standard Ethernet header as an example.

```
struct ether_t {
    dmac     : 48;
    smac     : 48;
    etype    : 16;
};
```

Ethernet header processing. This instruction decides whether the incoming packet is an IP packet by looking at the EtherType field in the Ethernet header. If the incoming packet is an IP packet, the program continues to instruction `process_ip`; otherwise, it goes to instruction `count_dropped`. In this case, we set the intermediate register `ir0` to 0 to indicate the current packet is a non-IP packet, and `count_dropped` will use `ir0` to calculate the starting address of the corresponding Packet/Byte Counter for non-IP packets.

```
process_ether:
begin
    ir0 = 0;
    if (ether_ptr->etype == 0x0800) {
        goto process_ip;
    }
    goto count_dropped;
end
```

IP header processing. This instruction looks at the Version and Internet Header Length (IHL) fields of the IP header. A Version value equal to 4 and IHL value equal to 5 indicate the current packet is a non-option IP packet. For non-option IP packets, the program continues to instruction `forward_packet`; otherwise, it goes to instruction `count_dropped`. In this case, we set the intermediate register `ir0` to 1 to indicate the current packet is an IP-options packet, and `count_dropped` will use `ir0` to calculate the starting address of the corresponding Packet/Byte Counter for IP-options packets.

```
process_ip:
begin
    const ipv4_t *ipv4_addr = ether_ptr +
        sizeof(ether_t);
    ir0 = 1;
    if (ipv4_addr->ver == 4 &&
        ipv4_addr->ihl == 5) {
        goto forward_packet;
    }
    goto count_dropped;
end
```

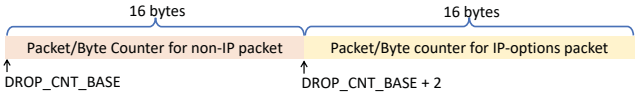

Figure 6: Packet/Byte Counter layout in our example. Pointer indicates the starting address of each counter.

Dropped packet counting. This instruction increments the Packet/Byte Counter for dropped packets. Packet/Byte Counter is a special counter stored in the Shared Memory System. Each Packet/Byte Counter is 16 bytes, and consists of two portions: packet counter portion, which calculates the number of packets; and byte counter portion, which calculates the number of bytes. Figure 6 shows the Packet/Byte Counter layout and the corresponding starting addresses in our example. Instruction `count_dropped` first calculates the address of the Packet/Byte Counter to be incremented based on `DROP_CNT_BASE` and `ir0`. If `ir0` is equal to 0, then the current packet is a non-IP packet, and the corresponding counter address is `DROP_CNT_BASE`; otherwise the current packet is an IP-options packet, and the corresponding counter address is `DROP_CNT_BASE+2`. This instruction then issues an external transaction (XTXN) called `CounterIncPhys`, which is specific for incrementing Packet/Byte Counter and takes two parameters: the counter address and the packet length. This XTXN increments the packet counter portion and the byte counter portion in Packet/Byte Counter separately: the packet counter portion is incremented by 1, and the byte counter portion is incremented by `pkt_len`.

```
count_dropped:
begin
    const : addr = DROP_CNT_BASE + ir0 * 2;
    CounterIncPhys(addr, r_work.pkt_len);
    goto drop_packet;
end
```

Packet forwarding and dropping. For completeness, we show brief definitions of instructions `forward_packet` and `drop_packet` as referenced by prior instructions. In our implementation, both packet forwarding and packet dropping are completed by multiple instructions.

```
forward_packet:
// code to forward the packet
// based on the destination address

drop_packet:
// code to drop the packet
```

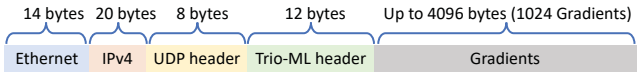

Figure 7: Trio-ML packet format.

4 USE CASE 1: IN-NETWORK AGGREGATION IN TRIO

The previous sections described Trio's flexible packet processing architecture, the memory system, and the programming environment. In this section, we discuss in-network aggregation for distributed ML training as a concrete use case.

Data-parallel training. One of the common approaches to distributed Machine Learning (ML) training is *data-parallel* training. In this approach, the neural network is replicated across N workers (or replicas), with each worker processing a small subset of the training data (mini-batch) to compute its local model gradients. At every iteration, workers must synchronize their model parameters by exchanging and aggregating their gradients to ensure convergence [39]. This step is called *allreduce*, and it is most commonly implemented using a parameter server [53] or ring-allreduce [9, 65].

Overview of in-network aggregation. The allreduce step puts significant pressure on the network fabric because the entire set of model gradients must be exchanged many times throughout the training process [39, 40, 65]. Recent work proposed *in-network aggregation* to improve the performance of distributed ML training workloads [18, 36, 47, 48, 54, 63, 77]. By aggregating gradients *inside* network switches, rather than at end-hosts, in-network aggregation accelerates training jobs with heavy communication overheads.

Trio-based in-network aggregation. We now describe Trio-ML, our Microcode implementation to perform in-network aggregation inside Trio-based devices. Section 5 extends Trio-ML to handle straggling workers and describes the challenges faced by Tofino-based in-network aggregation solutions [48, 63] in the presence of stragglers.

Trio-ML packet format. Figure 7 illustrates Trio-ML's aggregation packet format. Following previous proposals [48, 63], we use UDP packets to carry the gradients. Packets are addressed to the router with a pre-defined destination port (e.g., 12000). After the UDP header, we define a Trio-ML header that describes the block of gradients carried in each packet. A block is a subset of DNN model gradients that fits in one packet. The gradients are 32-bit integers converted from floating-point using the scaling approach proposed by ATP [48].

Trio-ML header structure. Figure 8 shows the Trio-ML header structure, as defined in our Microcode program. `job_id` and `block_id` uniquely identify the block of gradients for each training job. All servers participating in the same `job_id` send blocks of gradients using the same sequence of `block_ids`. `src_id` identifies the sender of each packet, enabling the aggregator to keep track of which servers have contributed their data to the block and to recognize retransmissions by the servers. Generation number `gen_id` is used to distinguish blocks in consecutive iterations of the model aggregation. The header structure has three additional variables (`age_op`, `degraded`, and `src_cnt`) that are most meaningful in the context of stragglers (details in §5).

```
struct trio_ml_hdr_t { // 12 bytes
    job_id    :  8; // aggregation job id
    block_id  : 32; // aggregation block id
    age_op    :  4; // if the block has aged out
    final     :  1; // if the block is final block
    degraded  :  1; // aggregation is partial
              :  2; // unused for byte alignment
    src_id    :  8; // source id of the packet
    src_cnt   :  8; // number of sources contributing
    gen_id    : 16; // generation id
              :  4; // room to expand grad_cnt
    grad_cnt  : 12; // number of gradients
};
```

Figure 8: Trio-ML packet header structure.

Job records. We use a hash table (with key = `job_id, block_id`) to keep track of ongoing aggregations in the device. Figure 17 in Appendix A.1 shows the structure definition of job records. Job records are created at job configuration time and persist until the job is complete. They contain the current number of active blocks being aggregated for each job (`block_curr_cnt`). They also control memory sharing across jobs by capping the maximum number of concurrent aggregation blocks (`block_cnt_max`) and the maximum number of gradients per block (`block_grad_max`) for a given job. In addition, job records hold the parameters required for generating and forwarding the response, as well as the block expiration timeout. Finally, job records contain bitmasks indicating which sources (workers) are participating in the job.

Block records. Trio-ML creates a block record when it receives a packet with a new `block_id` from a server. Figure 18 in Appendix A.1 shows the structure definition of block records. The Microcode program removes the block record when the block aggregation is complete and the block's result has been generated. Block records hold the block's aggregation state, including the count and bitmask of sources that must still deliver their packets, pointers to the aggregation buffer and the parent job record, and the block's start time and expiration interval. Figure 9 illustrates the Trio-ML aggregation Microcode program's data structure operations when multiple aggregation jobs are present concurrently, and each job directs multiple blocks to aggregate in parallel.

Window-based streaming aggregation. Following prior work [48, 63], we assume servers do not send the entire model to the aggregator at once. Instead, they stream the gradients using a `window` parameter. Each server has a parameter called `window` that controls the number of outstanding gradients waiting to be aggregated. Section 6 evaluates the impact of `window` size on performance.

Trio-ML aggregation Microcode program workflow. Figure 10 shows Trio-ML's aggregation Microcode program workflow. Each aggregation packet is processed by one thread. Each thread starts by extracting `job_id` and `block_id` from the packet and using them to look up the block aggregation record. If the block record does not exist (i.e., this is the first packet with a certain `block_id`), the thread proceeds to create the block record (provided that a job record with the specified `job_id` exists). The block record points to the aggregation buffer in the Shared Memory System (DMEM), where gradients are summed up. In Trio, packets consist of a head, which holds the first 192 bytes of the packet, and a tail, which

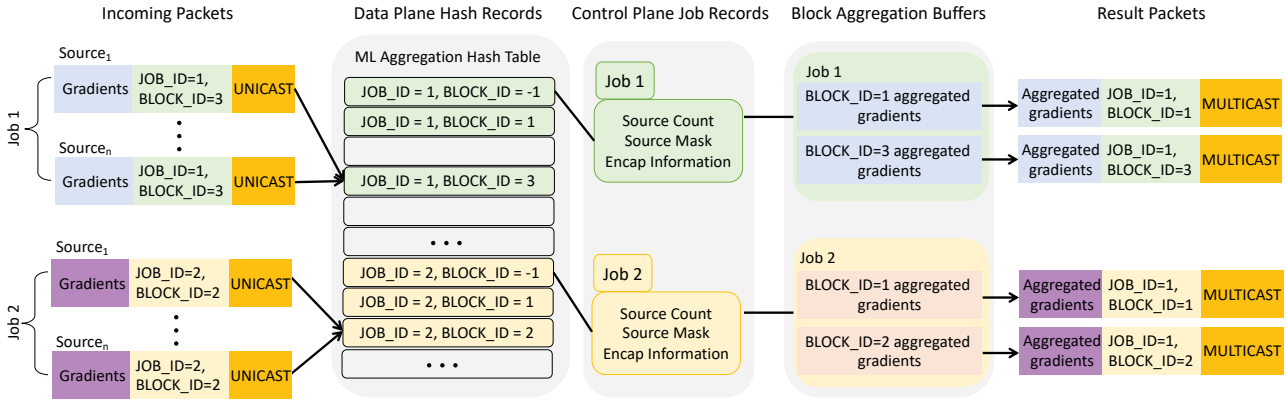

Figure 9: Data structure operations in Trio-ML aggregation Microcode program.

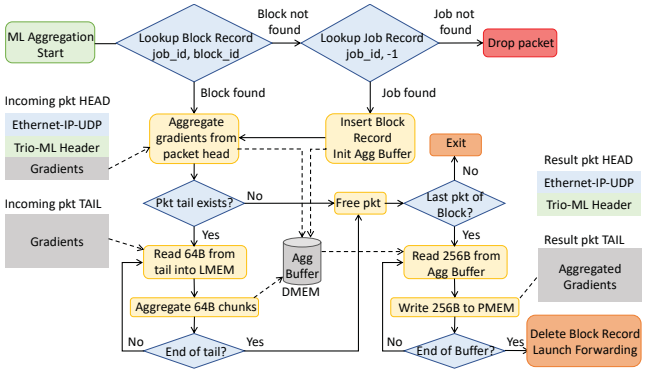

Figure 10: Trio-ML aggregation Microcode program workflow.

holds the rest of the packet. Packet head data is readily available in the thread's Local Memory (LMEM), whereas tail data resides in the Memory and Queueing Subsystem and must be read into LMEM before they can be used. Hence, aggregation proceeds in two phases. Phase one aggregates gradients from the packet head, while phase two is structured as a loop that aggregates gradients from the packet tail in 64-byte chunks (16 32-bit gradients). When all gradients in a packet have been aggregated, the block context is checked for completeness. If all sources have been accounted for, the block context is complete, and the result generation phase starts.

Result packet. Every block produces one aggregation Result packet. The Result packet is a new packet, whose IP/UDP and Trio-ML headers are reconstructed from the block and job records, and whose data (gradients) come from the aggregation buffer. The new packet must have a head and tail, just as incoming packets do. Accordingly, the tail is constructed in a loop: each iteration pulls a 256-byte chunk from the DMEM aggregation buffer into LMEM and then writes it out to the new tail in the Packet Buffer (PMEM). Finally, the Result packet is passed to the standard forwarding code using the nexthop address from the job record (`out_nh_addr` field).

Hierarchical aggregation. Trio-ML uses single-level aggregation when all ML sources are connected to interfaces hosted on the same PFE. In this case, the destination IP address of the Result packet is the address of the multicast group, which spans all sources participating in the job. Server membership in the multicast group is achieved by allowing each server to issue an IGMP registration or by including server interfaces in a Static Multicast configuration on the router. Standard IP forwarding then takes care of delivering the Result packet to all servers. Trio-ML uses hierarchical aggregation when ML sources span multiple PFEs. In hierarchical aggregation, one of the PFEs is configured as a top-level aggregator to which the other (first-level aggregator) PFEs feed their results. With hierarchical aggregation inside a multi-PFE chassis, first-level PFEs send their packets to the designated top-level PFE directly, without relying on IP forwarding. The top-level PFE sees lower-level PFEs as individual sources and aggregates them in the same way as a single-level aggregator does. Hierarchical aggregation can be extended to work across multiple devices by setting the destination IP of the Result packet to the IP address of next-level aggregator and relying on IP forwarding to unicast the packet. The top-level aggregator will, of course, multicast the final result back to the servers. A desirable property of hierarchical aggregation is that the amount of data is reduced as the aggregated gradients move up the hierarchy, in a manner opposite to multicast replication. Note that when hierarchical aggregation is being set up, all configurations are done via the control-plane, and no Microcode changes are needed.

5 USE CASE 2: IN-NETWORK STRAGGLER MITIGATION

As a first use case, the previous section walked through Trio-ML's Microcode program, explaining how it implements in-network aggregation for ML training jobs. This section describes in-network straggler mitigation as a second important use case.

The straggler problem. In shared clusters hosting several jobs, different servers often experience uncorrelated performance jitter due to congestion, load imbalance, resource contention, garbage collection, background OS activities, or storage delays [13, 26, 33, 34, 38, 51, 56, 79]. As a result, servers that are collectively working on the same job must wait for the slowest job, hence, decreasing the overall application performance. This problem is known as the straggler problem and has been studied extensively for several

distributed applications, including MapReduce [14, 15, 31, 75, 78], Spark [57, 79], database queries [59], key-value stores [46], and machine learning [25, 30, 38, 61, 72]. Google cites the straggler problem as one of the main causes of poor performance in its data processing jobs [46], and production traces from Facebook and Microsoft indicate that jobs can be slowed down by a factor of eight due to stragglers [14].

The case for in-network straggler mitigation. Existing systems address the straggler problem in a number of ways, including cloning [13, 14], speculative execution [15, 31, 79], and rapid reassignment [38]. Cloning approaches are prohibitively expensive for large jobs, such as ML training. Speculatively executing duplicate work and using rapid reassignment approaches require servers to coordinate updates via message passing, which delays detection and mitigation. In addition, today's straggler mitigation techniques are all server-based, thereby potentially creating a circular dependency wherein stragglers may need to be mitigated using other servers that could be straggling themselves. We argue that decoupling straggler mitigation from servers is a more robust approach. Hence, we propose *in-network* straggler mitigation as a more suitable solution for latency-sensitive applications. In-network straggler mitigation leverages the network devices' vantage point to keep track of active workers and promptly react to stragglers. Importantly, in-network straggler mitigation avoids the need for extra communication time across servers to detect straggling workers in distributed systems.

Trio to the rescue. We now explain our approach to implementing in-network straggler mitigation in Trio-based devices. To the best of our knowledge, it is challenging, if not impossible, to realize *efficient* in-network straggler mitigation in PISA-based devices mainly because performing timer-based operations (such as sending notification packet to servers when the timer for checking straggler events expires) in P4 requires coordination with the switch control plane. In comparison, Trio has the ability to perform timer-based processing and can spawn multiple threads in PPEs. We use our Trio-ML application (§4) as a running example to explain the use of Trio's timer threads for straggler detection and mitigation.

Straggler detection with timer threads. Trio's architecture contains tens of high-resolution timers, which can be used to launch Microcode threads that execute periodically. To detect straggling sources in our Trio-ML application, we leverage these threads to trigger periodic scanning of the aggregation hash table. Another advantage of Trio is that its hash hardware supports a per-record 'Recently Referenced' (REF) flag. REF flags are set when a record is created and whenever it is referenced by a lookup. To detect stragglers, we program Trio's timer threads to periodically visit all hash records to check and clear their REF flags. The timer threads determine whether the records have aged out by checking each record's REF flag prior to clearing it. If the flag is not set, it means the record has not been accessed for at least the duration of the timer interval. We use this feature to set a timeout interval to detect straggling sources.

Multi-thread scanning of large hash tables. One of the key challenges of in-network straggler detection is that routers and switches inside the network may have to absorb a large amount of data from non-straggling servers while waiting for the timeout to expire. For instance, in our Trio-ML application, the `window` parameter allows servers to send up to 65,535 aggregation packets (each 4 KBytes in size). Trio-ML aggregates these packets as soon as they arrive, but it needs to keep the block records of partially aggregated results in its hash table. Our timer threads need to be able to quickly scan this large hash table to identify aged blocks. We find it is challenging for a single thread to scan a large hash table to locate expired records. To address this challenge, we deploy N periodic threads operating in parallel. This is achieved by initiating the threads such that the interarrival interval between back-to-back threads is $1/N$ of the desired timeout interval. Every triggered thread scans $1/N$ of the aggregation table, thus reducing the amount of processing required by each individual thread by a factor of N. Trio's timer resolution allows hundreds of threads to be deployed in this manner. In this scenario, no PPE is reserved specifically for running timer threads, and every timer thread can be spawned in any of the PPEs based on availability.

Straggler mitigation. Once a straggler is identified, various techniques may be used to mitigate its impact on the application's performance. The complexity of these techniques depends on the application. Following prior work [36], Trio-ML gives up on the straggling source(s) and sends a partial aggregation result to all the workers, including the straggler(s), along with the number of sources participating in the partial result. We use the `age_op` field in the Trio-ML packet header structure (Figure 8) to indicate whether a block has aged out due to stragglers. If it has, the `degraded` field is set on the Result packet to inform the servers that the aggregation result has been calculated using only a partial set of workers, and the `src_cnt` field informs the senders how many non-straggling sources contributed to the aggregation result. Servers that receive partial aggregation results divide the returned aggregated gradient values by the number of aggregated sources extracted from the Trio-ML header.

Advanced straggler mitigation. Although the straggler mitigation approach we use in Trio-ML is simple, the techniques described in this paper are generic and can be used to implement more complex straggler mitigation approaches for other latency-sensitive applications. For instance, service providers can use Trio's timer threads to identify whether a worker is a temporary straggler (slows down temporarily) or a permanent one (is out of service for a very long time), and notify all other workers accordingly. To do so, the Microcode program needs to implement two types of timer threads. One type happens more frequently; it detects straggler events, similar to the timer threads for our ML use case. Another type happens less frequently; this type detects the per-server straggler event count, analyzes whether these are temporary or permanent stragglers, and sends notification to all other workers.

6 EVALUATIONS

This section evaluates the performance of our in-network aggregation and straggler mitigation use cases. First, we explain our testbed setup and methodology (§6.1). Next, we demonstrate Trio-ML's time-to-accuracy and iteration time speedups in the presence of stragglers, as well as the efficiency of timer threads in Trio (§6.2). Finally, we benchmark the latency and throughput of the Trio-ML Microcode program without stragglers (§6.3).

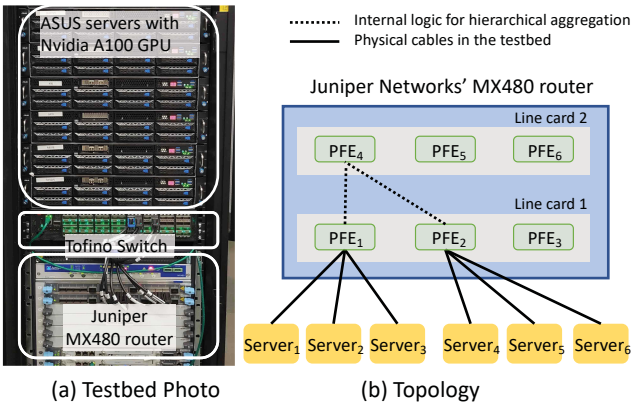

Figure 11: Our 100 Gbps testbed with a Juniper MX480 router, a Tofino switch, and six GPU servers.

6.1 Methodology and Setup

Testbed. Our testbed includes six ASUS ESC4000A-E10 servers, one 64×100 Gbps Tofino switch, and one Juniper Networks' MX480 router [10]. Each server is equipped with one A100 Nvidia GPU [11] (40 GBytes of HBM2 memory) and one 100 Gbps Mellanox ConnectX5 NIC. The Juniper router is populated with two MPC10E-15C-MRATE line cards [4]. Each line card hosts 12 × 100 Gbps ports distributed across three PFEs based on Trio's 5th generation chipset. Figure 11(a) shows a photo of our testbed. To demonstrate the power of hierarchical aggregation in Trio, we connect three servers to PFE_1 and another three servers to PFE_2, as shown in Figure 11(b). All PFEs are internally connected; hence, we enable Trio-ML's hierarchical aggregation by configuring PFE_4 as the top-level aggregator. The dotted lines in Figure 11(b) illustrate the internal path of our hierarchical aggregation setup. For Tofino experiments, we connect all six servers to a single pipeline because SwitchML's open-source code does not yet support hierarchical aggregation across pipelines. Note that connecting all servers to a single pipeline of the Tofino switch guarantees the best performance of SwitchML. If servers are connected to multiple pipelines, recirculation is required and will result in performance degradation.

DNN Workloads. We evaluate three real-world DNN models: ResNet50 [41], DenseNet161 [42], and VGG11 [68]. Table 1 summarizes the models and batch sizes used in our experiments. Following prior work [7, 63, 66], we select batch sizes that achieve the best possible time-to-accuracy. We train all three models with the ImageNet dataset [32].

SwitchML setup. For our baseline, we use SwitchML's open-source code [64]. SwitchML provides RDMA-based and DPDK-based implementations. However, its RDMA-based implementation is not yet integrated with training frameworks. Hence, we use SwitchML's DPDK-based implementation integrated with the PyTorch [49] training framework. In addition, SwitchML provides two packet size designs: (*i*) SwitchML-64, where each packet carries 64 gradients and uses a single pipeline on the Tofino switch to perform aggregation, and (*ii*) SwitchML-256, where each packet carries 256 gradients requiring all four pipelines to perform the aggregation at line rate. SwitchML-256 performs better than SwitchML-64; therefore, in our evaluations, we use SwitchML-256 with pool size 512, even though it consumes the resources of *all four pipelines* on our Tofino switch. Finally, SwitchML end-hosts retransmit their gradients after 1 ms to tolerate packet loss. But this feature creates spurious retransmissions during straggling periods, reducing SwitchML's performance. Therefore, we disable this feature in our experiments.

DNN	Model Size	Batch size/GPU	Dataset
ResNet50	98 MB	64	ImageNet
VGG11	507 MB	128	ImageNet
DensNet161	109 MB	64	ImageNet

Table 1: DNN models used in our experiments.

Trio-ML setup. To make an apples-to-apples comparison with SwitchML, we configure Trio-ML servers to use DPDK integrated with PyTorch. Unless otherwise stated, we configure each server to send 1024 gradients per packet and stream the gradients using a `window` size of 4096 packets. Section 6.3 evaluates the impact of varying the number of gradients and the `window` size on performance. For straggler detection, we launch $N = 100$ timer threads on Trio, each with a 10 ms timeout period, unless otherwise stated.

Straggler generation pattern. To evaluate the impact of in-network straggler mitigation, we synthetically generate transient worker slowdown by inserting sleep commands into the servers during training. Following prior work [38], we use the "Slow Worker Pattern" to inject stragglers by selecting three possible delay points in each iteration and allowing one of the servers to decide to slow down at each point with a given probability p (straggling probability). If a worker decides to straggle at a particular delay point, it will be slowed for a period uniformly randomly chosen between 0.5 and 2× of the typical iteration time, where typical iteration time refers to the average iteration time of each model when there are no stragglers in the system.

Ideal setup. To compare Trio-ML to an ideal environment where no stragglers exist in the system, we use Pytorch with NCCL [8] and an RDMA backend without adding stragglers.

6.2 Distributed ML Training Speedups

Time-to-accuracy improvements. The ultimate goal of in-network aggregation is to reduce the time-to-accuracy of distributed ML training workloads, even in the presence of stragglers. Figure 12 shows the top-5 validation accuracy results for the three DNN training jobs when the straggling probability p is 16%. This probability emulates an environment with a moderate rate of stragglers. Note that the probability of a straggler event occurring in a single iteration increases with the number of workers participating in a job [56]. Therefore, it is entirely possible that at a large scale, every training iteration observes at least one straggling worker even when it is a different straggler in each iteration due to uncorrelated transient effects [38]. As shown in Figure 12(a), when training ResNet50, Trio-ML reaches 90% target validation accuracy 1.56× faster than SwitchML. Trio-ML recovers from straggler delays via its partial aggregation strategy, whereas SwitchML servers need to wait for the straggling server. Similarly, Figures 12(b) and (c) show that Trio-ML outperforms SwitchML by 1.56× and 1.60× when training DenseNet161 and VGG11, respectively.

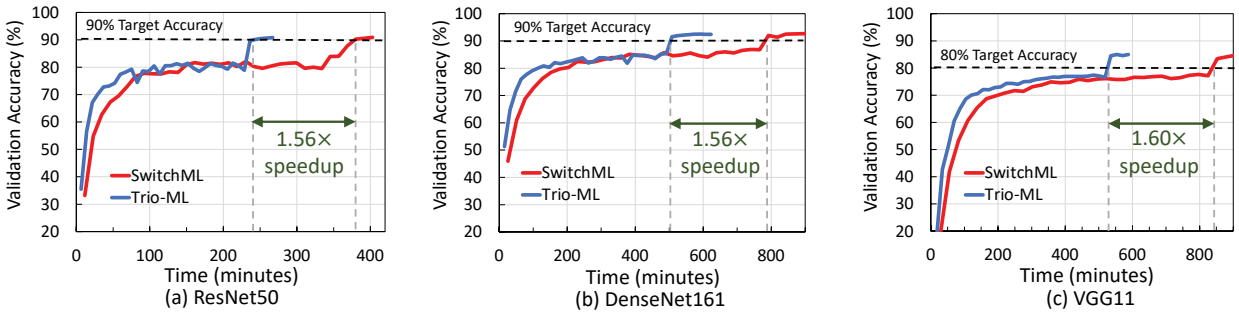

Figure 12: Time-to-accuracy improvements for three DNN models when straggling probability is $p = 16\%$.

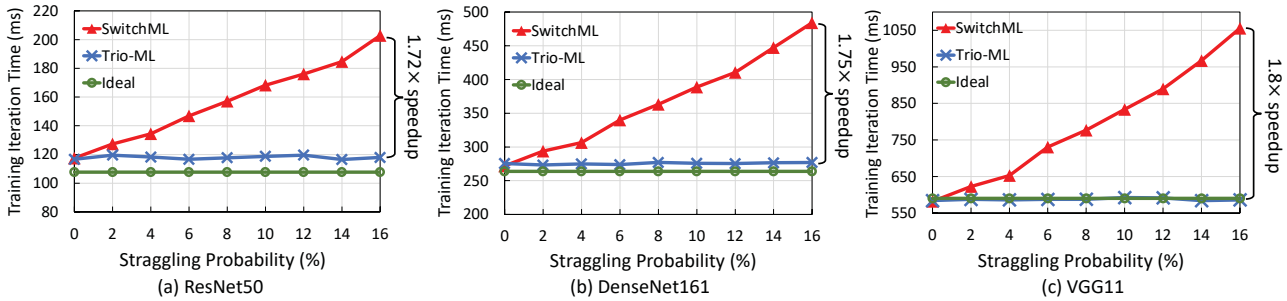

Figure 13: Impact of straggling workers on training iteration time for three DNN models. Trio-ML is able to maintain the training iteration time close to the Ideal case.

Training iteration time. To evaluate the impact of straggling probability on training performance, we sweep through different probabilities and measure the corresponding training iteration times. Figure 13 shows the average iteration time for the first 100 iterations when training three DNN models with Trio-ML, SwitchML, and the Ideal setup. The figure shows that as the straggling probability increases, SwitchML's iteration time increases because the switch needs to wait for the straggling worker(s) before generating the aggregation result. Increasing SwitchML's pool size does not help, as its aggregation logic requires all participating workers to contribute before making progress. In contrast, Trio-ML mitigates the effect of stragglers using its in-network straggler mitigation technique and is able to maintain the training iteration time close to the Ideal case. With straggling probability $p = 16\%$, Trio-ML speeds up average iteration time by 1.72× for ResNet50, 1.75× for DenseNet161, and 1.8× for VGG11, compared to SwitchML.

In-network timer threads' efficiency. Trio's timer threads periodically scan the aggregation records to identify which servers are straggling beyond the specified timeout interval. To evaluate the efficiency of this process, we vary the timeout interval on Trio and measure how long it takes for the non-straggling servers to receive partial aggregation results. For each timeout interval, we send 20 back-to-back packets and report the time between sending one aggregation packet and receiving the corresponding result packet on each server. Figure 14 shows Trio-ML servers are able to recover from stragglers within 2× the timeout interval.

6.3 Trio-ML Microcode Program Performance

The previous section established Trio-ML's performance in the presence of stragglers. This section benchmarks the performance of our in-network aggregation Microcode program without stragglers (i.e., $p = 0$). In these benchmarks, we use four servers connected to the same PFE.

Microcode program analysis. The Trio-ML Microcode program is quite compact, using ≈60 instructions. It uses a single thread per packet where most of the cycles are spent on a loop that reads gradients from the packet tail into the thread's local memory and adds them into the aggregation buffer. This loop's efficiency is ≈1.2 run-time instructions per gradient, and it is executed for every packet from every source. Another loop copies aggregated gradients into the Packet Buffer when building the aggregation result packet; it uses less processing time, because it is executed once per block, i.e., once for the whole set of sources. Trio's read-modify-write engines perform the summation of gradients entering the aggregation buffer. Trio-ML uses 12 such engines, and each add operation takes two cycles. With a 1 GHz clock speed, the current Trio generation supports 6 billion operations per second per PFE.

Aggregation latency. We quantify the PFE aggregation latency by instrumenting the Trio-ML Microcode program to keep track of the amount of time each aggregation packet spends in Trio. To compute a faithful estimation of a single thread's aggregation latency, we enforce each server to send only one aggregation packet at a time by setting the `window` parameter to one. Each experiment consists of sending 10,000 back-to-back packets, and we repeat the experiments 20 times. The left y-axis of Figure 15 reports the average aggregation latency as the number of gradients per packet is increased. With 64 gradients per packet, the aggregation latency is 30 μs. Larger packets incur a larger aggregation latency, but this increase is not always linear. For instance, increasing the packet size by a factor of 16 (1024 on the x-axis) causes the aggregation

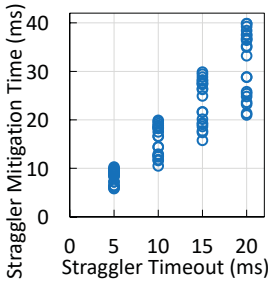

Figure 14: In-network timer threads efficiency.

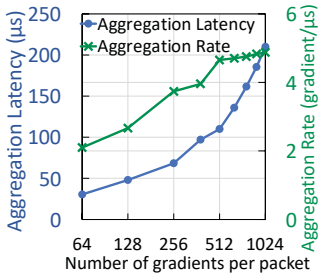

Figure 15: Per PFE aggregation latency and rate.

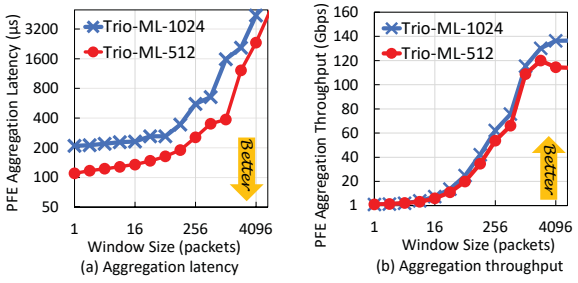

Figure 16: Impact of `window` size on aggregation latency and throughput.

latency to increase to 200 μs (a factor of 6.6 increase). This result suggests Trio is more efficient with larger packets. The right y-axis of Figure 15 confirms this observation by plotting the aggregation rate (the ratio of the number of gradients over the aggregation latency). As shown, the aggregation rate starts to plateau between 512 and 1024 gradients per packet. Next, we evaluate the impact of increasing the `window` size for these two packet sizes.

Impact of aggregation `window` size. Increasing the `window` parameter enables the PPE threads to work on multiple aggregations in parallel. To evaluate the impact of `window` size on Trio-ML's aggregation latency and throughput, we configure the servers to send packets with 512 or 1024 gradients with varying `window` sizes. We refer to these two cases as Trio-ML-512 and Trio-ML-1024. Figure 16 shows the interplay between aggregation latency and throughput as the `window` size increases. Figure 16(a) shows that increasing the `window` size causes the aggregation latency to increase because the Microcode program needs to handle more simultaneous aggregation packets. Figure 16(b) shows that increasing the `window` size improves the aggregation throughput because it pipelines packet arrivals into the router. The best `window` is the one that maximizes the throughput while minimizing latency. We find `window` size 4096 achieves a good balance between latency and throughput.

7 DISCUSSION AND FUTURE USE CASES

Trio for in-network telemetry. Most network operators require telemetry or insights into the traffic in their networks for capacity planning, service-level agreement monitoring, security mitigation, and other purposes. Current networking devices usually rely on packet sampling using internal processors embedded in the devices or external monitoring devices for further processing. Because of the high rate of traffic through the devices and the limited amount of processing and bandwidth available for monitoring, only a small percentage of packets (one in tens of thousands or less) is selected to be monitored, and the decision to sample packets is often *blind*, based on a simple time interval [62]. Trio's packet processing flexibility and availability of operational resources make it suitable for in-network telemetry. For instance, service providers can leverage Trio's large memory to keep track of incoming packets to maintain sufficient information for telemetry. Moreover, Trio's timer threads are suitable for periodic monitoring and anomaly analysis. To provide more intelligent telemetry for network operators, machine learning-based classification techniques may be performed on each packet, based on the packet fields already extracted by Trio for routing purposes. Finally, the data structures can be stored more efficiently, thereby reducing the transmission bandwidth and processing cycles of external monitoring devices.

Trio for in-network security. To mitigate DDoS attacks, the MX systems based on Trio support a feature to identify and drop malicious packets, capitalizing on the chipset's high performance and flexible packet filter mechanism. Trio also acts as a fast forwarding-path based on security flows on the SRX security platforms [1]. Trio is capable of performing additional complex in-network security processing on incoming packets, either by aggregating features or by performing inference of ML models installed by service providers, to identify and mitigate anomalies in traffic. Unlike off-device-based solutions, Trio's programmable architecture for anomaly detection on the network datapath enables low-latency threat mitigation.

Packet loss in Trio-ML. Transient traffic spikes may occur in a datacenter running a variety of diverse applications, and this, in turn, may lead to aggregation packets being lost. A practical in-network aggregation system needs a level of resiliency allowing long-running jobs to survive such hiccups. SwitchML [63] suggests how such resiliency can be achieved. Trio-ML implementation has provisions to support this solution, although it is not part of the current code and we leave this to future work.

Future open sourcing plans. We are considering several future open source ideas. First, we plan to add comprehensive support for P4 programming for Trio. Juniper engineering has made an initial effort to achieve this goal [6], but recent changes and enhancements to the P4 core specification should allow greater flexibility and more features to be exposed via the P4 interface. Second, we plan to create a domain-specific language to allow the full scope of forwarding-path features of the Trio chipset to be available to third-party developers. Juniper is exploring development in this area and welcomes feedback from the community.

8 RELATED WORK

In-network computing using programmable switches. Several prior papers proposed in-network computing by leveraging some form of programmability inside the network. These approaches fall into two categories: (1) computation at *line-rate* using PISA-based architectures [18, 48, 63] and (2) computation at *sub line-rates* using on-chip FPGAs [21]. Our in-network ML aggregation use case is closely related to Sharp [18], SwitchML [63], ATP [48], PANAMA [36], and Flare [29]. Sharp [18] is Mellanox's proprietary

design geared towards dedicated ML training clusters; it assumes network bandwidth can be exclusively reserved. In contrast, we consider networks where links are shared across multiple users and applications. SwitchML [63] and ATP [48] use commercially available Tofino switches to perform gradient aggregation. Although Tofino switches can perform line-rate packet processing, their pipelined architecture has more limited programmability, making in-network straggler mitigation extremely challenging. We use SwitchML as a baseline comparison to Trio-ML. SwitchML serves as an apples-to-apples comparison for our use case, making it a more appropriate baseline than ATP. More specifically, ATP's performance improvements are impacted by in-network aggregation and an additional parameter server, while SwitchML and Trio-ML are more similar, as both approaches only use switch/router for aggregation. PANAMA's [36] in-network aggregation hardware can support flexible packet processing, but it is based on FPGAs acting as bumps-in-the-wire, making it impractical for large scale deployments. This paper, however, aims to use Trio's programmable architecture to design new stateful in-network applications from the ground up. Several key features of Trio enable these new applications. First, Trio's large memory and fast access to packet tail data enable efficient in-network computation. Second, Trio's Shared Memory System provides several GBytes of storage; this is sufficient for data storage even in the presence of straggling workers, or when multiple applications are running simultaneously. Finally, Trio has no limits on the number of instructions on a single packet, enabling the Microcode program to launch the computation instructions required by large packets.

Straggler mitigation. There is a plethora of prior work on understanding and mitigating the impact of stragglers in distributed systems [13–15, 25, 26, 30, 31, 33, 34, 38, 46, 51, 56, 57, 59, 61, 72, 75, 78, 79]. In particular, Harlap et al. proposed FlexRR to mitigate the impact of stragglers on distributed learning jobs [38]. FlexRR requires peer-to-peer communication among workers to detect slowed workers and perform work re-assignment. In contrast, we consider mitigating stragglers inside the network without any message passing across workers and without requiring a parameter server. Tandon et al. [72] and Raviv et al. [61] proposed coding theory frameworks for mitigating stragglers in distributed learning by duplicating the training data across workers; however, Trio-ML does not require data duplication.

Alternative switch architectures. The research community has been working on alternative switch architectures to address some of the limitations of PISA-based architectures, such as lack of shared memory and shallow pipeline depths. The most competitive example is dRMT (Disaggregated Programmable Switching) [24]. The dRMT switch architecture implements a centralized, shared memory pool that all match-action stages can access. Instead of executing the match-action stages in a pipeline, dRMT aggregates these stages in a cluster and executes them in a round-robin order. A control logic unit schedules the stages so as to maximize the cluster's throughput while respecting program dependencies. However, the centralized memory pool is gated by a *mux* that connects stages to memory, and only one stage can access memory in a given clock cycle. This can result in the slow down of program execution when an application requires memory access in multiple stages. In Trio, multiple threads can send memory access requests to the same memory location at around the same time, and Trio's read-modify-write engine processes the requests in sequence, guaranteeing consistency of the updates. In addition, dRMT's memory accesses through the crossbar are scheduled at compile time, and this reduces the flexibility of incrementally updating and recompiling the application code. The complexity of the crossbar scheduling algorithm can limit the ability of the architecture to scale to higher numbers of match-action processors. In contrast, Trio's crossbar is scheduled in real-time, thus providing efficient access to memory. This dynamic scheduling mechanism enables Trio to scale from 16 PPEs in the first generation to 160 PPEs in the sixth generation, and it will continue to scale higher in the future. Furthermore, in dRMT, the packet parser and deparser are located outside the match-action processors. Any parsing of the inner headers of the packets that rely on the lookup results (e.g., MPLS encapsulated packets) will have to be recirculated back to the parser for processing. In contrast, Trio's PPEs are fully programmable processors, able to handle packet parsing/deparsing as well as the rest of the packet lookup and processing, in a run-to-completion manner. Trio's multi-threaded PPEs also allow packets to be processed by different Microcode programs depending on their processing requirements.

9 CONCLUSION

This paper describes Juniper Networks' programmable chipset, Trio, and its use in emerging data-intensive in-network applications. Trio has been in production for over a decade and has built a large customer base with billions of dollars in market share. We describe Trio's multi-threaded and programmable packet forwarding and packet processing engines. We then use in-network aggregation for distributed machine learning training and in-network straggler mitigation as two use cases to illustrate Trio's Microcode and programming environment. Our evaluations show that Trio outperforms today's pipeline-based solutions by up to 1.8×. This work does not raise any ethical issues.

10 ACKNOWLEDGMENTS

We would like to thank our shepherd Gábor Rétvári and anonymous reviewers for their valuable feedback. We also acknowledge Juniper Networks for providing resources for this research. In particular, we thank Pradeep Sindhu and the Trio development team for creating the chipsets. We are grateful to Raj Yavatkar and Alex Mallery for valuable discussions of the paper. This work was supported by the Air Force AI Accelerator, ARPA-E ENLITENED PINE, DARPA FastNICs, NSF grants CNS-2008624, SHF-2107244, ASCENT-2023468, CAREER-2144766, and Sloan fellowship. This research was partially sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. The nature of this research is not military-related and does not have direct military implications.

REFERENCES

[1] [n. d.]. Accelerating the Next Generation of Juniper Connected Security with Trio. ([n. d.]). https://blogs.juniper.net/en-us/security/accelerating-the-next-generation-of-juniper-connected-security-with-trio.

[2] [n. d.]. Barefoot Tofino. ([n. d.]). https://www.barefootnetworks.com/products/brief-tofino/.

[3] [n. d.]. Juniper Networks Advanced Forwarding Interface (AFI). https://github.com/Juniper/AFI. ([n. d.]).

[4] [n. d.]. Juniper Networks' MX Series Universal Routing Platform Interface Module Reference. ([n. d.]). https://www.juniper.net/documentation/us/en/hardware/mx-module-reference/topics/concept/mpc10e-15c-mrate.html.

[5] [n. d.]. Juniper Networks vMX Series Universal Routing Platform. https://www.juniper.net/us/en/products/routers/mx-series/vmx-virtual-router-software.html. ([n. d.]).

[6] [n. d.]. Juniper P4 Agent. https://github.com/Juniper/JP4Agent. ([n. d.]).

[7] [n. d.]. MLPerf: A broad ML benchmark suite. ([n. d.]). https://mlperf.org/.

[8] [n. d.]. NVIDIA Collective Communication Library (NCCL). ([n. d.]). https://developer.nvidia.com/nccl.

[9] 2017. Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow. (2017). https://eng.uber.com/horovod.

[10] 2022. Juniper Networks' MX480 Universal Routing Platform. (2022). https://www.juniper.net/us/en/products/routers/mx-series/mx480-universal-routing-platform.html.

[11] 2022. NVIDIA A100 Tensor Core GPU. (2022). https://www.nvidia.com/en-us/data-center/a100/.

[12] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, A. Kind, J. F. Logan, M. Peyravian, M. A. Rinaldi, R. K. Sabhikhi, M. S. Siegel, and M. Waldvogel. 2003. IBM PowerNP network processor: Hardware, software, and applications. *IBM Journal of Research and Development* 47, 2.3 (2003), 177–193. https://doi.org/10.1147/rd.472.0177

[13] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2012. Why let resources idle? Aggressive Cloning of Jobs with Dolly. In *USENIX HotCloud* (usenix hotcloud ed.). https://www.microsoft.com/en-us/research/publication/let-resources-idle-aggressive-cloning-jobs-dolly/

[14] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 185–198. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan

[15] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/reining-outliers-map-reduce-clusters-using-mantri

[16] Sally Bament. 2022. Juniper Introduces New Trio 6-based MX Portfolio. (2022). https://blogs.juniper.net/en-us/service-provider-transformation/juniper-introduces-new-trio-6-based-mx-portfolio.

[17] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-band Network Telemetry. In *ACM SIGCOMM*.

[18] Gil Bloch. 2019. Accelerating Distributed Deep Learning with In-Network Computing Technology. (Aug. 2019). https://conferences.sigcomm.org/events/apnet2019/slides/Industrial_1_3.pdf

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review (CCR)* (2014).

[20] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[21] Pietro Bressana, Noa Zilberman, Dejan Vucinic, and Robert Soulé. 2020. Trading Latency for Compute in the Network. In *ACM NAI*.

[22] Broadcom. [n. d.]. BCM56870 Series. ([n. d.]). https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[23] A. Caulfield, P. Costa, and M. Ghobadi. 2018. Beyond SmartNICs: Towards a Fully Programmable Cloud: Invited Paper. In *IEEE HPRS*.

[24] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *ACM SIGCOMM*.

[25] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. 2013. Solving the Straggler Problem with Bounded Staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX Association, Santa Ana Pueblo, NM. https://www.usenix.org/conference/hotos13/session/cipar

[26] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[27] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soule. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking* 28, 4 (2020).

[28] 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.

[29] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: Flexible in-Network Allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 35, 16 pages. https://doi.org/10.1145/3458817.3476178

[30] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[31] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[33] Celestine Dünner, Thomas Parnell, Dimitrios Sarigiannis, Nikolas Ioannou, Andreea Anghel, Gummadi Ravi, Madhusudanan Kandasamy, and Haralampos Pozidis. 2018. Snap ML: A Hierarchical Framework for Machine Learning. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 252–262. http://papers.nips.cc/paper/7309-snap-ml-a-hierarchical-framework-for-machine-learning.pdf

[34] Farshid Farhat, Diman Zad Tootaghaj, Yuxiong He, Anand Sivasubramaniam, Mahmut Kandemir, and Chita R. Das. 2018. Stochastic Modeling and Optimization of Stragglers. *IEEE Transactions on Cloud Computing* 6, 4 (oct 2018), 1164–1177. https://doi.org/10.1109/tcc.2016.2552516

[35] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. 2022. Enabling In-situ Programmability in Network Data Plane: From Architecture to Language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 635–649.

[36] N. Gebara, P. Costa, and M. Ghobadi. 2021. PANAMA: In-network Aggregation for Shared Machine Learning Clusters. In *Proc. Conference on Machine Learning and Systems (MLSys)*. 1–16.

[37] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. 2020. Challenging the Stateless Quo of Programmable Switches. In *ACM Workshop on Hot Topics in Networks (HotNets)*. ACM. https://www.microsoft.com/en-us/research/publication/challenging-the-stateless-quo-of-programmable-switches/

[38] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2016. Addressing the Straggler Problem for Iterative Convergent Parallel ML. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 98–111. https://doi.org/10.1145/2987550.2987554

[39] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *CoRR* abs/1806.03377 (2018).

[40] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H. Campbell. 2018. Communication Scheduling as a First-Class Citizen in Distributed Machine Learning Systems. *CoRR* abs/1803.03288 (2018). arXiv:1803.03288 http://arxiv.org/abs/1803.03288

[41] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[42] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '17)*. IEEE Computer Society, Honolulu, HI, 2261–2269. https://arxiv.org/abs/1608.06993v5

[43] 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. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[44] 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 '20)*. Association for Computing Machinery, New York, NY, USA, 90–106. https:

//doi.org/10.1145/3387514.3405855

[45] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic External Memory for Switch Data Planes. In *ACM HotNets*.

[46] Eugene Kirpichov and Malo Denielou. 2016. No shard left behind: dynamic work rebalancing in Google Cloud Dataflow. (May 2016). https://cloud.google.com/blog/products/gcp/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow

[47] Benjamin Klenk, Nan Jiang, G. Thorson, and L. Dennison. 2020. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), 996–1009.

[48] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761.

[49] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alexander Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. *CoRR* abs/1903.12287 (2019). arXiv:1903.12287 http://arxiv.org/abs/1903.12287

[50] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. 2019. The Case for Network Accelerated Query Processing. In *Proceedings of the Innovative Data Systems Research Conference (CIDR '19)*.

[51] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[52] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[53] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[54] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement learning with In-Switch Computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 279–291.

[55] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM '17)*.

[56] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[57] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[58] P4.org Architecture Working Group. [n. d.]. P416 Portable Switch Architecture (PSA). ([n. d.]). https://p4.org/p4-spec/docs/PSA.html.

[59] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[60] Dan R. K. Ports and Jacob Nelson. 2019. When Should The Network Be The Computer?. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*.

[61] Netanel Raviv, Itzhak Tamo, Rashish Tandon, and Alexandros G. Dimakis. 2020. Gradient Coding From Cyclic MDS Codes and Expander Graphs. *IEEE Transactions on Information Theory* 66, 12 (2020), 7475–7489. https://doi.org/10.1109/TIT.2020.3029396

[62] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*.

[63] 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. https://www.usenix.org/conference/nsdi21/presentation/sapio

[64] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2022. SwitchML open source code. (2022). https://github.com/p4lang/p4app-switchML.

[65] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. (2018). arXiv:cs.LG/1802.05799

[66] Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2018. Measuring the Effects of Data Parallelism on Neural Network Training. *CoRR* abs/1811.03600 (2018). arXiv:1811.03600 http://arxiv.org/abs/1811.03600

[67] Vishal Shrivastav. 2022. Stateful Multi-Pipelined Programmable Switches. In *Proceedings of the 2022 ACM SIGCOMM Conference (SIGCOMM '22)*.

[68] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. (2015). arXiv:cs.CV/1409.1556

[69] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM*.

[70] Erich Strohmaier, Jack J. Dongarra, Hans W. Meuer, and Horst D. Simon. 1999. The Marketplace of High-Performance Computing. *Parallel Comput.* 25, 13–14 (dec 1999), 1517–1544. https://doi.org/10.1016/S0167-8191(99)00067-8

[71] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. 2022. Taurus: A Data Plane Architecture for Per-Packet ML. *ASPLOS* (2022).

[72] Rashish Tandon, Qi Lei, Alexandros G. Dimakis, and Nikos Karampatziakis. 2017. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 3368–3376. http://proceedings.mlr.press/v70/tandon17a.html

[73] 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 Conference (SIGMOD '20)*.

[74] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *EuroSys*.

[75] Da Wang, Gauri Joshi, and Gregory W. Wornell. 2019. Efficient Straggler Replication in Large-Scale Parallel Computing. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 4, 2, Article 7 (April 2019), 23 pages. https://doi.org/10.1145/3310336

[76] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets'19)*.

[77] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan RK Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. 2022. Unlocking the Power of Inline {Floating-Point} Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 683–700.

[78] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 265–278. https://doi.org/10.1145/1755913.1755940

[79] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 29–42.

A APPENDICES

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

A.1 Trio-ML Record Structure

Figure 17 shows the structure definition of the job records. The field without a field name represents unused bits for byte alignment. The information contained in other fields are listed as follow:

`block_curr_cnt`: current number of active blocks
`block_cnt_max`: maximum number of concurrent blocks
`block_grad_max`: maximum number of gradients per block
`block_exp`: block timeout interval in millisecond
`block_total_cnt`: job's cumulative blocks count
`out_src_addr`: Result packet source IP
`out_dst_addr`: Result packet destination IP
`out_nh_addr`: pointer to egress forward chain
`src_cnt`: number of ML sources in the job
`src_mask_0`: bitmask field for job's sources
`src_mask_1`: additional bitmask field for job's sources
`src_mask_2`: additional bitmask field for job's sources
`src_mask_3`: additional bitmask field for job's sources

```
struct trio_ml_job_ctx_t { // 58 bytes
    block_curr_cnt  : 16;
    block_cnt_max   : 12;
    block_grad_max  : 12;
    block_exp       :  8;
    block_total_cnt : 32;
    out_src_addr    : 32;
    out_dst_addr    : 32;
    out_nh_addr     : 32;
                    : 24;
    src_cnt         :  8;
    src_mask_0      : 64;
    src_mask_1      : 64;
    src_mask_2      : 64;
    src_mask_3      : 64;
};
```

Figure 17: Trio-ML job record structure.

Figure 18 shows the structure definition of the block records. The field without a field name represents unused bits for byte alignment. The information contained in other fields are listed as follow:

`block_exp`: block timeout interval in millisecond
`block_age`: age of the current block
`block_start_time`: start time of the current block
`job_ctx_paddr`: pointer to the job record
`aggr_paddr`: pointer to the aggregation buffer
`grad_cnt`: number of gradients in the block
`rcvd_cnt`: number of received ML sources
`rcvd_mask_0`: bitmask field for received sources
`rcvd_mask_1`: additional bitmask field for received sources
`rcvd_mask_2`: additional bitmask field for received sources
`rcvd_mask_3`: additional bitmask field for received sources

```
struct trio_ml_block_ctx_t { // 58 bytes
    block_exp        :  8;
    block_age        :  8;
    block_start_time : 64;
    job_ctx_paddr    : 32;
    aggr_paddr       : 32;
                     : 20;
    grad_cnt         : 12;
                     : 24;
    rcvd_cnt         :  8;
    rcvd_mask_0      : 64;
    rcvd_mask_1      : 64;
    rcvd_mask_2      : 64;
    rcvd_mask_3      : 64;
};
```

Figure 18: Trio-ML block record structure.