Query Planning for Robust and Scalable Hybrid Network Telemetry Systems

CHAOFAN SHOU, UC Berkeley, USA
ROHAN BHATIA, Google Inc., USA
ARPIT GUPTA, UC Santa Barbara, USA
ROB HARRISON, U.S. Military Academy, USA
DANIEL LOKSHTANOV, UC Santa Barbara, USA
WALTER WILLINGER, NIKSUN Inc., USA

Network telemetry systems have become hybrid combinations of state-of-the-art stream processors and modern programmable data-plane devices. However, the existing designs of such systems have not focused on ensuring that these systems are also deployable in practice, i.e., able to scale and deal with the dynamics in real-world traffic and query workloads. Unfortunately, efforts to scale these hybrid systems are hampered by severe constraints on available compute resources in the data plane (e.g., memory, ALUs). Similarly, the limited runtime programmability of existing hardware data-plane targets critically affects efforts to make these systems robust. This paper presents the design and implementation of DynaMap, a new hybrid telemetry system that is both robust and scalable. By planning for telemetry queries dynamically, DynaMap allows the remapping of stateful dataflow operators to data-plane registers at runtime. We model the problem of mapping dataflow operators to data-plane targets formally and develop a new heuristic algorithm for solving this problem. We implement our algorithm in prototype and demonstrate its feasibility with existing hardware targets based on Intel Tofino. Using traffic workloads from different real-world production networks, we show that our prototype of DynaMap improves performance on average by 1-2 orders of magnitude over state-of-the-art hybrid systems that use only static query planning.

CCS Concepts: • **Networks** → **Network measurement**; **Network monitoring**.

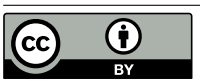
Additional Key Words and Phrases: analytics, programmable switches, stream processing

ACM Reference Format:
Chaofan Shou, Rohan Bhatia, Arpit Gupta, Rob Harrison, Daniel Lokshtanov, and Walter Willinger. 2024. Query Planning for Robust and Scalable Hybrid Network Telemetry Systems. *Proc. ACM Netw.* 2, CoNEXT1, Article 3 (March 2024), 27 pages. https://doi.org/10.1145/3649471

1 INTRODUCTION

In recent years, we have seen the emergence of hybrid network telemetry systems that combine the flexibility of off-the-shelf stream processors, such as Apache Fink [28], with programmable network devices, e.g., Intel Tofino [36]. These hybrid systems enable flexible streaming analytics at scale by offloading a subset of the telemetry processing to the network data plane for execution at line rate. Over time, many of these hybrid systems have converged on a standard format for expressing

Authors' addresses: Chaofan Shou, shou@berkeley.edu, UC Berkeley, California, USA; Rohan Bhatia, brohan52@gmail.com, Google Inc., California, USA; Arpit Gupta, arpitgupta@ucsb.edu, UC Santa Barbara, California, USA; Rob Harrison, rob.harrison@westpoint.edu, U.S. Military Academy, New York, USA; Daniel Lokshtanov, daniello@ucsb.edu, UC Santa Barbara, California, USA; Walter Willinger, wwillinger@niksun.com, NIKSUN Inc., New Jersey, USA.



ACM 2834-5509/2024/3-ART3
https://doi.org/10.1145/3649471

their analytic tasks as queries applied against a stream of incoming packets in a time-windowed fashion. These queries apply a sequence of dataflow operators, e.g., `map`, `reduce`, etc., to the stream of incoming packets and transform them into the desired results for each time window.

Examples of such hybrid systems include Marple [25], Sonata [9], Newton [48], Dynatos [21] and FlyMon [47], but they vary in the types of analytic queries they support and the methods they use to offload query processing to the data plane. Despite the potential benefits of using hybrid telemetry systems in real-world networks, most of these systems are not designed with deployability to production networks in mind. Notably, their designs fail to address how to scale to real-world traffic and query workloads that are dynamic in nature. To achieve the scalability and robustness needed for real-world deployments, these systems should *dynamically* decide which telemetry processing components to offload to the data plane and how to allocate data-plane resources over time. The goal of such dynamic decision making is to maximally benefit from the data plane's fast packet processing capabilities, which reduces the load on the stream processor, while at the same time adhering to the data-plane resource constraints.

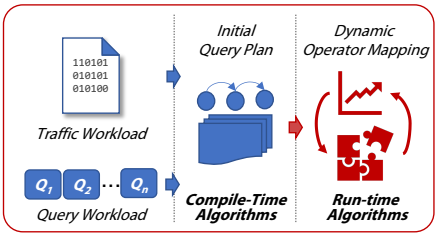


Fig. 1. **Overview of DynaMap.**

Most hybrid telemetry systems do not support such dynamic decision making otherwise known as *dynamic query planning*. Those that support any query planning either make static query-planning decisions at compile time [9], or support a more limited set of input queries [21]. Static decision making for hybrid telemetry systems is bound to perform poorly in real-world networks where traffic and query workloads change over time. Specifically, the inability to reallocate data-plane resources and adapt to changing workloads can negatively impact the system's ability to answer queries accurately and significantly increases the processing load on the stream processor. However, recent works [47, 48] have shown that certain actions critical for dynamic query planning can now be realized with the runtime programmability supported by modern data-plane targets. These developments motivate the design of new hybrid telemetry systems that promise to make dynamic query planning a reality in theory and support it in practice.

In this paper, we argue for and demonstrate the importance of performing dynamic query planning when designing hybrid network telemetry systems for real-world production networks. We also seek algorithmic solutions and designs that are future-proof, i.e., those that will work well for new workloads as well as for future data-plane targets that include the latest runtime-programmable features. To this end, we avoid tailoring our proposed solution to a particular hardware target (e.g., Tofino switch), vendor design (e.g., Tofino vs. Broadcom), or traffic/query workload. We also purposefully discount alternative or simpler solutions that only work well for a specific hardware target or assume a niche traffic or query workload. Specifically, our work offers the following contributions:

Formal treatment of dynamic operator mapping (Section 3): We identify the problem of dynamically mapping stateful operators to data-plane registers at runtime as the key to designing scalable and robust hybrid telemetry systems. We express this problem formally as the General Optimal Mapping (GOM) problem that considers various query- and target-specific constraints. Solutions to this problem find mappings that maximize the efficacy of dynamic query planning. We prove that even for single-stage, data-plane targets, solving this dynamic operator mapping problem is NP-hard and present a new greedy heuristic for solving this dynamic operator mapping problem for multi-stage hardware targets.

Design and implementation of DynaMap (Section 4): We present an implementation of our new greedy heuristic for performing dynamic operator mapping at runtime and complement it with instances of two compile-time algorithms. Figure 1 shows how these runtime and compile-time algorithms work together and form the building blocks of our new hybrid network telemetry system. DynaMap supports dynamic query planning and is, therefore, suitable for deployment in production settings. DynaMap computes an initial query plan at compile time to get started and then, at runtime, implements a learning model for predicting the various dataflow operators' future resource requirements based on their observed resource usage in the past.
Evaluation of a prototype DynaMap (Section 5): We develop a prototype of DynaMap with an Intel Tofino-based hardware switch as the data-plane target and evaluate our prototype using a combination of synthetic and real-world traffic and query workloads. Our results show the benefits of supporting dynamic operator mapping at runtime and demonstrate the efficacy of using purposefully-designed, compile-time and runtime programs in concert to enable dynamic query planning in practice. These empirical results are based on commonly-used, real-world network traffic traces used in several other recent studies. To encourage further research on designing hybrid network telemetry solutions that support dynamic query planning, we open-sourced DynaMap's entire code base and related research artifacts (e.g., query workload model) at `https://github.com/SNL-UCSB/dynamap/`.

This work does **not** raise any **ethical concerns**.

2 HYBRID TELEMETRY SYSTEMS

The emergence of programmable data-plane targets (e.g., programmable switches, NICs, etc.) has motivated new, hybrid designs for network telemetry systems [9, 21, 25, 48]. These systems combine the fast packet-processing speed of hardware switches with the flexible and horizontally scalable data-processing capabilities of modern stream processors, e.g., Apache Spark [35]. As such, these hybrid systems must cope with the disparity in resources and capabilities available at data-plane targets versus general-purpose CPUs. Although modern data-plane targets offer faster packet processing speeds than existing general-purpose CPUs, the former have much more limited compute resources (e.g., memory, stages, storage) and provide much more restrictive reconfigurability and runtime programmability compared to the latter. In this section, we describe the nature of telemetry queries, Protocol Independent Switch Architecture (PISA) switches, and provide case studies of existing hybrid telemetry systems.

2.1 Telemetry queries, data-plane targets, and PISA-pipelines

Telemetry tasks are naturally expressed in the form of queries. Many of the existing hybrid network telemetry systems have converged on expressing their queries as a sequence of dataflow operators (e.g., `map`, `filter`, `reduce`, `distinct`) applied over an incoming packet stream in a windowed fashion. A typical example of such a query is shown in Query 1 that identifies hosts by `sourceIPs` which send traffic to more than Th other distinct hosts within a window of size w seconds.

```
1  packetStream(w)
2  .map(p => (p.sourceIP, p.destIP))
3  .distinct()
4  .map((sourceIP, destIP) => (
       sourceIP, 1))
5  .reduce(keys=(sourceIP,), f=sum)
6  .filter((sourceIP, count) =>
       count > Th)
```

Query 1. **Detecting superspreader hosts**.

Programmable data-plane targets are commonly implemented using ASICs, FPGAs, or NPUs and differ in packet-processing speed and abilities. Among them, existing hybrid telemetry systems frequently incorporate PISA switches implemented using ASICs. In PISA-based data-plane targets, packet processing works by forwarding packets through a pipeline comprised of a sequence of

functional blocks or stages. Each stage contains dedicated resources including a match-action unit (MAU) for matching operations and register memory for stateful operations, such as, calculating a `sum`. Resources in one stage are inaccessible in other stages, but data may be shared across stages through metadata passed from stage to stage. Below, we focus our discussion on PISA targets because they form an entire family of targets with varying capabilities.

Hybrid telemetry systems must ultimately map the high-level operators from telemetry queries to the specific resources available in the PISA pipeline. However, the current generation of PISA-based targets offers limited capabilities to *reconfigure* these pipelines at *runtime*, and assuming no restrictions on the runtime programmability of existing commodity data-plane targets is unrealistic. To illustrate, consider two types of decisions encountered when mapping high-level query operators to data-plane resources: how to size the registers and which high-level operators to map to the registers. While the first decision deals with deciding how much resources (e.g., SRAM) to allocate to a given register, the second type refers to determining which dataflow operator to map to a specific data-plane register. Target- and query-specific considerations play a role in making these two decisions, but while resizing registers at runtime is not feasible with currently available data-plane targets, recent studies (e.g., BeauCoup [3], Newton [48], Flymon [47]) have considered mechanisms to update operator-to-register mappings at runtime without incurring any network downtime due to recompilation. Given these limitations, it is critical to map dataflow operators to stages at compile time that best satisfy the operators' dynamic memory requirements without compromising the queries' accuracy at runtime.

2.2 Query Planning to the Rescue

Given the scarce resources and limited capabilities of existing data-plane targets, hybrid network telemetry systems must carefully decide where in the data-plane pipeline to execute different portions of the input queries' dataflow operators and how much resources to allocate to the different portions. This problem only becomes more challenging in real-world settings where these systems must perform at scale and under inherently dynamic query and traffic workloads. To deal with this problem, some existing systems perform *query planning*. While there exists a large body of literature in the database research community that deals with the problem of query optimization in general (e.g., see [12] and references therein) and adaptive (or dynamic) query processing in particular (e.g., see [6]), a systematic treatment of query planning as part of designing hybrid network telemetry systems is absent from the literature.

In the case of hybrid telemetry systems, query planning entails making a set of domain-specific decisions that include how much memory to allocate to each of the dataflow operators for each telemetry query active during a particular time window (w). If chosen well, these resources will allow the system to accurately identify the *traffic of interest*, or ToI, which is the portion of the total network traffic that satisfies these queries. When the memory needed to execute a specific operator in the data plane is exhausted, hybrid telemetry systems typically delegate the processing of future packets to the stream processor. For example, consider a query planning process that determines a stateful operator can be implemented in the data plane using an array of size 1024 for storing key-value pairs. All future packets corresponding to new keys ≥ 1025 that do not fit in that array must be processed at the stream processor otherwise we will miss some ToI. Consequently, we use the number of packets that require processing at the stream processor rather than in the data plane as our metric to evaluate the efficiency of a chosen query plan. Query planning aims to keep this load as small as possible.

Theoretical considerations. Assuming perfect knowledge of both the considered query and traffic workloads and imposing no restrictions on the data-plane targets' runtime programmability capabilities, query planning boils down to a complex resource scheduling problem. Any feasible

solution to this optimization problem defines a potential query plan. The individual query planning decisions that constitute such a plan specify (i) which operators should be executed during which period in time (*i.e.,* mapping operators in *time*), (ii) which subsets of the ToI identified by some key, e.g., source IP, should be processed by which operators (i.e., mapping operators in *key-space*), and (iii) how much memory should be allocated to each of the operators that are running in a given window (*i.e.,* mapping operators to *data-plane resources*). An optimal solution to this problem will result in a query plan that allocates data-plane resources to accurately identify the ToI and with minimal load on the stream processor.

Challenges at Runtime. In real-world, production networks, the dynamic nature of query and traffic workloads necessarily requires runtime modifications of even an optimal compile-time query plan. While device-level *compile-time programmability* is widely supported in modern network devices and has been extensively studied in prior work (see discussion in [41]), support for device-level *runtime programmability* has only recently been explored, and it remains limited [8, 41]. Any design of an effective hybrid telemetry system that uses currently-available, programmable data-plane devices needs to be informed by the precise nature of the existing restrictions and the impact they have on what type of query planning decisions can and cannot be performed at runtime. For example, the Tofino architecture does not support resizing register memory at runtime, but does support dynamic changes to the match-action tables. In Section 4, we detail how we used both compile-time and runtime capabilities of the Tofino architecture to support dynamic operator mapping at runtime.

2.3 Case Studies of Two Existing Hybrid Telemetry Systems

To our knowledge, only two hybrid telemetry systems have been successfully implemented with query planning. **Sonata** [9] implemented a *static query planning approach* whereby a static query plan is computed and implemented at compile-time and then used throughout the entire duration of system operation. A distinguishing feature of the static query plans considered by Sonata is that they are scalable with respect to the number of different concurrent queries and that this scalability property is attributed to the "iterative query refinement" technique (see [9] for a detailed description of this technique). Using this iterative query refinement mechanism, Sonata's static query plans are optimal solutions of appropriately-formulated ILP problems that describe a combination of the *operator-to-key-space* and *operator-to-register* mapping problems. The resulting static query plans can be expected to perform well for a fixed set of given queries for which the associated ToI (as part of the overall traffic workload) changes only minimally over time. However, lacking any mechanisms to adapt to changes in the query or traffic workloads over time, such static query planning prevents the resulting hybrid telemetry systems from being robust to variability in query and traffic workloads. Figure 2 shows the results of a simple experiment (see Section 5) and illustrates the cost of not adapting to the query and traffic dynamics. In the first few windows, several orders of magnitude fewer packets are sent to the stream processor than the overall number of incoming packets, but that advantage quickly fades with a static query plan. After those first few windows, static query plans result in a workload on the stream processor on par with processing **all** of the packets at the stream processor. Static query planning with Sonata is, therefore, not scalable under query and traffic dynamics.

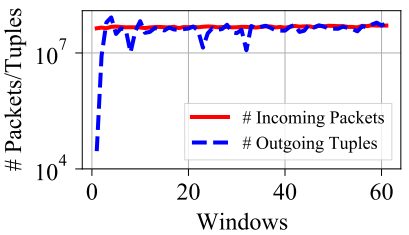


Fig. 2. **Cost of not adapting.**

DynATOS [21] implmented a *dynamic query planning* approach that leverages existing runtime programmability to reduce or eliminate the cost of not adapting to traffic and query workload

dynamics. DynATOS supports dynamic query workloads comprised of multiple concurrent or sequential *approximate queries*, *i.e.,* query responses tolerant of errors within some bound, and is also robust to changes in the traffic workload. DynATOS performs query planning by subdividing each window w into a fixed number of sub-epochs; for each of the sub-epochs, it determines which of the queries are active and then decides how to allocate data-plane resources to each of the active queries' stateful operators. Resource scheduling is thus broken down into a succession of per-window scheduling problems, where each of these problems can be formulated as an ILP that is a combination of the *operator-to-time* (here, time refers to sub-epochs) and *operator-to-resources* mapping problems. To realize this design, DynATOS employs a closed-loop, adaptive approximation and scheduling algorithm rather than explicitly predicting traffic or resource requirements. To support these operations with existing runtime programmability widely supported by current commodity hardware targets, DynATOS assumes a simple data-plane target with only a single computing stage, thus tying the design to a particular hardware architecture while also limiting the expressiveness of the supported telemetry queries.

2.4 Our contribution: DynaMap

Given the strengths and weaknesses of both of the described approaches for designing hybrid telemetry systems, we choose to combine the two approaches. We present DynaMap, a hybrid telemetry system that first computes a suitable query plan at compile time (*i.e.,* before the start of the first window) and subsequently modifies this query plan incrementally, from one window to the next, by leveraging supported runtime programmability capabilities. Determining an initial query plan at compile time takes inspiration from how Sonata computes static query plans, and making runtime modifications at each transition from one window to the next inspired by how DynATOS changes query plans on the fly. In the remainder of this paper, we describe how we realize this hybrid design by developing, implementing, and evaluating a hardware prototype of our system. In particular, we detail our main algorithmic contribution to the design of DynaMap in Section 3, discuss implementation-specific aspects in Section 4, and evaluate our hardware prototype of DynaMap in Section 5.

3 DYNAMIC OPERATOR-MAPPING PROBLEMS

We consider an approach to dynamic query planning that involves creating an initial query plan at compile time and incrementally modifying it at runtime by updating operator-to-register mapping for each new window. Such an approach enables systems to adapt flexibly to variations in traffic or query workload. In this section, we present two data-plane-specific formulations of the dynamic operator mapping problem, prove their NP-hardness, and provide a greedy heuristic to efficiently map operators to registers at runtime.

3.1 A Simplified Optimal Mapping Problem

We first consider a simplified version of the problem of finding an optimal operator mapping at runtime and refer to it as the Simplified Optimal Mapping (SOM) problem.

Problem formulation: The SOM version of the dynamic operator mapping problem assumes that the available hardware targets have only a single computing stage. Denoting by R and O the sets of registers and operators, respectively, the objective of the SOM problem is to compute a mapping or assignment $\alpha : R \rightarrow O$ at the end of each window w that minimizes the SP's load for the next window $w' = w + 1$. We also allow for "partial" assignments that are defined for subsets $R' \subset R$. Letting D denote the resource requirements for window w' (*i.e.,* the per-operator memory requirements (B_o) and the number of input and output tuples (N_o^{in} and N_o^{out}) associated with each

dataflow operators $o \in O$), the SOM problem uses D as input to determine the resource footprint and load on the SP for a feasible operator mapping.

For each operator $o \in O$, we denote by s_o its *size*, and for each $r \in R$ we denote p_r its *capacity*. We also equip each $o \in O$ with two non-negative integer-valued quantities c_o^u and c_o^s that we call o's *unsatisfied* and *satisfied* cost, respectively. We require that for each $o \in O$, $c_o^s \leq c_o^u$. The *satisfaction ratio* $\sigma(o, \alpha)$ of $o \in O$ with respect to a mapping α is defined as

$$\sigma(o, \alpha) = \min\left(\frac{\sum_{r \in R \,:\, \alpha(r)=o} p_r}{s_o}, 1\right).$$

An operator o is said to be *unsatisfied* by the assignment α if $\sigma(o, \alpha) = 0$, *satisfied* if $\sigma(o, \alpha) = 1$, and *partially satisfied* otherwise.

The cost $c(o, \alpha)$ of operator $o \in O$ with respect to the assignment α is given by

$$c(o, \alpha) = c_o^s \cdot \sigma(o, \alpha) + c_o^u \cdot (1 - \sigma(o, \alpha)).$$

Thus the cost of a satisfied operator o is the satisfied cost c_o^s, the cost of an unsatisfied operator is the unsatisfied cost c_o^u, while the cost of a partially satisfied operator is between c_o^s and c_o^u based on the satisfaction ratio $\sigma(o, \alpha)$. Finally, the cost $c(\alpha)$ of an assignment α is defined by

$$c(\alpha) = \sum_{o \in O} c(o, \alpha).$$

Our goal is to compute an assignment α of minimum cost.

Main result: In Appendix A.1 we prove that solving the SOM problem is NP-hard and remains NP-hard even when all input integers are coded in unary.

3.2 The General Optimal Mapping Problem

In contrast to the SOM problem, the General Optimal Mapping (GOM) problem is concerned with data-plane targets that have multiple stages (and registers).

Problem formulation: This generalization imposes additional constraints on the problem. For example, any solution to the GOM problem has to satisfy certain ordering constraints for each query. To illustrate, if $o_i \rightarrow o_j$ are two consecutive stateful operators of a query and if $stages(o_i)$ denotes the set of stages to which o_i is mapped, then we require $max(stages(o_i)) < min(stages(o_j))$. For example, in the case of Query 1, we can only map the `reduce` operator to stages that are greater than the ones assigned to the `distinct` operator.

To account for these additional constraints, GOM's input differs from SOM's input in that it also includes conditions on the order in which the stateful operators for each query have to be processed. The conditions are specified by means of a partition $O_1, \ldots O_\ell$ of O into ℓ different *dependency chains* and an ordering $\pi_i : O_i \rightarrow \{1, \ldots, |O_i|\}$ for every dependency chain O_i. Here, a dependency chain represents a sequence of stateful operators for the input queries. For example, for Query 1, the set of stateful operators `distinct` $\rightarrow$ `reduce` is an example of a dependency chain.

Main result: Given that the GOM problem is a more complex version of the SOM problem, our proof of the NP-hardness of the SOM problem implies that the GOM problem is also NP-hard.

3.3 A greedy heuristic for solving GOM

To describe the key aspects of our proposed greedy heuristic, we call a mapping or assignment $\alpha_2 : R'' \rightarrow O$ an *extension* of $\alpha_1 : R' \rightarrow O$ if it satisfies the properties of $R' \subset R'' \subset R$ and $\alpha_1(r) = \alpha_2(r)$ for every $r \in R'$. Thus, α_2 assigns all the registers that α_1 does in precisely the same way and assigns some additional registers. A feasible extension e of α is an extension α' of α that is a feasible assignment, where an assignment is feasible when it satisfies the defining properties. We define E as the set of feasible extensions of α and note that each possible way to compute the

Algorithm 1: Greedy Heuristic Algorithm

```
Input: O, R, D;
α ← α_0;
while True do
    Compute a set E of feasible extensions of α;
    E ← extend(α);
    if E is empty then
        return α;
    e' ← max_{e∈E}([(c(α)−c(e)) / (cap(e)−cap(α))]);
    α ← e';
```

candidate set E and selecting e from E leads to a new heuristic. Thus, the objective of a greedy heuristic for the GOM problem is to compute a feasible assignment α of minimum cost.

Our greedy solution (Algorithm 1): At a high level, Algorithm 1 follows a natural greedy strategy and starts with the partial mapping or assignment α_0, which leaves all registers unmapped. This is a feasible assignment. Then the algorithm iteratively extends the assignment, adding more registers while satisfying the ordering constraints at every intermediate step. The algorithm halts when it is no longer possible to add more registers and maintain feasibility. Since there exists an infinite number of possible rules for selecting $e \in E$, our goal is to design simple rules that avoid selecting high-cost mappings. To this end, we divide our proposed greedy heuristic into two sub-tasks: computing a set of feasible extensions (E) and selecting the extension e in E that minimizes the mapping cost.

For the first sub-task, the algorithm focuses on dependency chains rather than individual unsatisfied operators. This approach speeds up computation by making decisions at a higher level and avoiding the need to explore extensions for each operator separately. Note that making decisions at the granularity of dependency chains rather than individual operators improves the computational efficiency of the algorithm. For a given unsatisfied dependency chain O_i, the algorithm finds an unsatisfied operator and returns the set of extensions (E) where this unsatisfied operator can be mapped to one of the unassigned registers.

For the second sub-task, we apply the greedy rule that selects the extension e in E that gives the most “bang per the buck", where “bang" refers to the cost (derived from the set of resource requirements that define the input D) and “buck" refers to the total capacity of the mapping. More formally, we define the total capacity of a mapping α with domain R' by $\text{cap}(\alpha) = \sum_{r \in R'} p_r$. Our algorithm selects e from E that maximizes

$$u(e) = \frac{c(\alpha) - c(e)}{\text{cap}(e) - \text{cap}(\alpha)}.$$

Such an approach prioritizes selecting extensions that help reduce the stream processor’s load and have smaller memory overhead in the data plane.

4 DESIGN AND IMPLEMENTATION OF DYNAMAP

Designing a new query planning algorithm that leverages the dynamic operator mapping approach presented in Section 3 at runtime requires developing a complementary set of auxiliary compile-time algorithms. In this section, we first describe three simple examples of the type of auxiliary algorithms needed to build a fully-functioning hybrid network telemetry system that supports

dynamic query planning. We then present the design and implementation of our new prototype of such a system that we call DynaMap.

4.1 Auxiliary compile-time algorithms

Our dynamic operator mapping algorithm assumes that an initial query plan consisting of an initial query refinement plan and an initial set of register sizes has been selected at compile time. To this end, we compute, for a given set of queries, both an initial query refinement plan and an initial set of register sizes. Ensuring that the runtime mapping of operators to registers for future windows can be performed in practice also requires an algorithm that uses operator memory requirements from past windows to predict the per-operator memory requirements for a new window.

Some of our design decisions described below regarding the proposed compile time algorithms as part of the design of our end-to-end system DynaMap are empirical in nature. These empirical findings are based on traffic workloads that consist of commonly-used real-world network traffic traces and query workloads that have been used in several recent studies (see Section 5 for details). These particular workload datasets have also been widely used in the past for evaluating the performance of existing network telemetry systems, and irrespective of their use, they inevitably invite questions about their representativeness and general applicability. However, like others have done in the past, we leave answering these questions for future work. At the same time, we emphasize that in our case, the described compile-time algorithms that leverage some of the empirical findings derived from these workloads only serve as simple examples of the type of supplementary efforts needed to build a fully-functioning hybrid network telemetry system. In fact, we designed our solutions to work for any workload, make no claims regarding their optimality for specific workloads, and invite others to develop better-performing solutions where possible. Some of the results in Section 6 indicate if and where there is room for such improvements.

Computing initial query refinement plans. For a given refinement plan, we define the metric *total operator memory (TOM)* as the sum of the memory required for each of the plan's stateful operators. TOM thus quantifies a refinement plan's memory footprint. Performing experiments with real-world traffic workloads, we observed that different queries have different memory footprints; the footprint of each query decreases as the number of refinement levels increases and the rate of decrease is generally only marginal for larger refinement levels. Together with the fact that the number of refinement levels of a query's refinement plan determines the number of ALUs needed to execute the required stateful operators in the data plane, these observations suggest that we can frame the refinement plan selection problem as finding, for each query, the number of refinement levels that minimize the load at the SP. To this end, we consider a heuristic that, instead of finding the optimal number of refinement levels for each query, computes a threshold value that minimizes the SP's load. Note that this threshold value implicitly determines the number of refinement levels for each query. We call this method the *TOM-based heuristic* and refer to Appendix A.2 for a more detailed description of the algorithm that implements this heuristic.

Determining initial register sizes. In general, determining an optimal register sizing policy is challenging. However, hardware-specific memory constraints that impose bounds on both the total memory per stage and the maximum memory per ALU suggest that intra-stage memory allocation matters more than inter-stage allocation and motivate our heuristic that uses the same configuration across all stages to compute appropriate register sizes for each ALU in a stage. For deciding how to allocate memory within a stage, we selected an allocation with high variability, mainly because having registers of different sizes can be expected to provide more flexibility when making operator mapping decisions at runtime for future windows. For simplicity, we consider variability scenarios that result from allocating memory proportional to the register's order. More concretely, if $\{1, 2, ..., A\}$ denotes the ordered set of registers in a given stage, then the allocated

register sizes will be $\{S, 2S, ..., AS\}$, where S is a scaling factor. Since we slice our decisions into different stages and repeat the same configuration across stages, we refer to this method as the *Slice-and-Repeat (SnR) heuristic*.

Predicting the input workload. To work in practice, we require DynaMap to use operator memory requirements from past windows to predict the memory requirements for a new window and rely on these predicted values to compute new operator mappings. Given the limited amount of historical data about measured operator memories, we consider only simple learning models that do not require a large training dataset. Among various off-the-shelf, time-series prediction tools, we chose the well-known Winter-Holt's double exponential smoothing prediction (DESP) method [40], mainly for its simplicity and ability to capture aspects of the short-term trends in memory requirements for the different operators over time. We observe that for most dataflow operators, the median prediction error with this method is less than 10 %, and to handle prediction errors in practice, especially for large operators, we simply scale the predicted memory requirements by a factor proportional to its prediction interval.

4.2 Prototyping with a Commodity Switch

We built DynaMap by integrating our newly proposed query-planning algorithm into an existing hybrid telemetry system, Sonata [9]. Our choice to extend Sonata for this purpose is motivated by its publicly available codebase and modular design and requires making only minimal changes. In the process, we also discuss system-specific optimizations we considered to run DynaMap with an off-the-shelf, Tofino-based hardware target. While updating query plans at runtime adds additional runtime complexity to hybrid telemetry systems such as DynaMap, our goal here is not to optimize DynaMap specifically for Tofino but to demonstrate the feasibility of dynamic query planning on existing programmable data plane targets and illustrate the challenges of implementing DynaMap with a Tofino-based switch as our data plane target. To this end, we develop DynaMap as a fork of Sonata's implementation with augmented query planner, runtime, and data-plane driver modules to support dynamic query planning. DynaMap retains Sonata's general mode of operation. The data plane processes incoming packets in a windowed fashion and through different stages; if a packet satisfies at least one of the queries, then the data plane adds some metadata to the packet header and sends it to the stream processor for further processing. At the end of each window, the stream processor reports the results to DynaMap's runtime, which then updates the match-action tables and resets the register values in the data plane.

Enabling dynamic operator mapping at runtime. DynaMap uses match-action table entries to update the mappings between stateful operators and data-plane registers/ALUs at runtime. This task also entails specifying the nature of the stateful operators (e.g., `distinct` or `reduce`) and their keys (e.g., `(sIP)` or `(sIP, dIP)`, etc.) for each ALU. At the ingress, DynaMap encodes this information into the packet's header vector (PHV) as metadata (see Appendix B for more details).

Encoding all the necessary information at ingress into the PHV is prohibitively expensive. For example, for a data plane target with sixteen ALUs, eight queries with an average of two refinement levels will need around 5k bits of metadata per packet. To deal with this issue, we leverage the observation that many of these metadata fields are reusable across processing stages. More concretely, we create two categories of metadata fields, namely persistent and ephemeral fields, and DynaMap reuses the *ephemeral* fields after every stateful operation. For example, the metadata fields carrying the information about which ALU programs (e.g., `distinct` or `reduce`) to use are reusable across different stages and are thus ephemeral (for more details, see Appendix B).This optimization enabled us to successfully compile all eight queries running concurrently on the Tofino-based Edgecore WEDGE100BF-32X [7]. Each query comprised 1-3 stateful operators and 2-3 refinement levels. The open-sourced code base at `https://github.com/SNL-UCSB/dynamap/`

details how DynaMap synthesizes the P4 program for configuring this data plane target. Scaling the current prototype for more queries (or refinement levels) will require further optimizations to reduce the metadata overhead.

Predicting future operator memory requirements. To aid the operator-mapping decisions at runtime, DynaMap's query planner needs to estimate the operator memory requirements for the next window. To synthesize these requirements, the query planner needs to extract information for all the input key-value pairs for all queries, irrespective of whether they satisfy the query or not. However, reading these values via the control plane is too slow to support dynamic operator mapping at runtime. Instead, DynaMap's runtime reads all the valid register values in the data plane using packet ferries [38]. These specially-crafted packets can read values from pre-specified data-plane registers.

5 EVALUATION

Our main evaluation results in Section 5.3 show that compared to static query planning approaches, (i) our dynamic solution that implements our proposed runtime and compile-time algorithms in concert and with full knowledge of the future traffic workload or operator memory requirements (i.e., no prediction) achieves on average a **2-3 orders of magnitude** reduction in load at the stream processor (SP) and ensures robustness to changing traffic (Figure 3) and query workloads (Figure 4), and (2) a practical end-to-end version of our solution that includes predicting the future traffic workload or operator memory requirements reduces the SP's load on average by **1-2 orders of magnitude** (Figure 3). We discuss additional findings that evaluate the efficacy of our compile-time algorithms and quantify runtime overhead for DynaMap's Tofino-based prototype in Appendix F.

5.1 Experimental Setup

Real-world traffic workloads. For our evaluations, we use three unsampled packet traces from two different production networks. The packet traces for the CAIDA datasets, i.e., `CAIDA-15` and `CAIDA-16`, were collected from a commercial backbone link on Dec 12, 2015, and Jan 21, 2016, respectively. The `CAMPUS-21` dataset was collected from the border router of a medium-sized campus network on Oct 10, 2021. To ensure that each dataset simulates an intense 100 Gbps workload, with around 50M packets per window, we apply suitably chosen speed-up factors. For example, for the hour-long `CAIDA-15` trace, we use a speed-up factor of 20 and a window size (w) of three seconds to curate a three-minute-long traffic workload consisting of a total of 60 windows, as considered in previous work [9].

Synthetic traffic workloads. To evaluate DynaMap's performance for traffic workloads that have higher variability in operator memory requirements than the considered production traffic, we curate synthetic traffic workloads in the form of the per-window memories required by each of the operators in a query plan. To this end, we use the operator memory requirements obtained from running eight Sonata queries [37] with the `CAIDA-15` dataset as input workload, computed for each operator the empirical Coefficient of Variation (CoV) of its per-window memory requirements, and synthesize individual operator memory requirements with a prescribed variability by sampling from a suitably-parameterized multivariate Gaussian distribution truncated on a simplex [1].By scaling each operator-specific CoV-value by a fixed factor, we can use this approach to synthesize workloads in the form of operator memory requirements with scaled-up variability while ensuring that the total operator memory remains constant.

Synthetic query workloads. When considering a static query workload, we used the eight queries considered in [9] and listed in Table 1 of Appendix C. This workload has been used in several other recent studies [9], [48], and [21]. However, when our evaluation requires considering dynamic query workloads, we synthesized query workloads that have been considered previously in [21]

and mimic real-world scenarios. To create this workload, we augment these eight queries from the static query workload with their mirrored versions. To mirror a query workload, we swap the keys used for stateful operations, e.g., source and/or destination IPs. The mirrored queries appear in Table 1 in Appendix C as $9 - 13$. We then use a Poisson process with rate λ to model the arrivals of new queries and randomly select each new query from the assembled superset. Each newly arriving query stays active for some random duration (drawn from an exponential distribution with mean μ). Using suitably chosen values of λ and μ, we can control L, the long-term average for the total number of active queries, by appealing to Little's Law [29] that states that $L \approx \lambda\mu$.

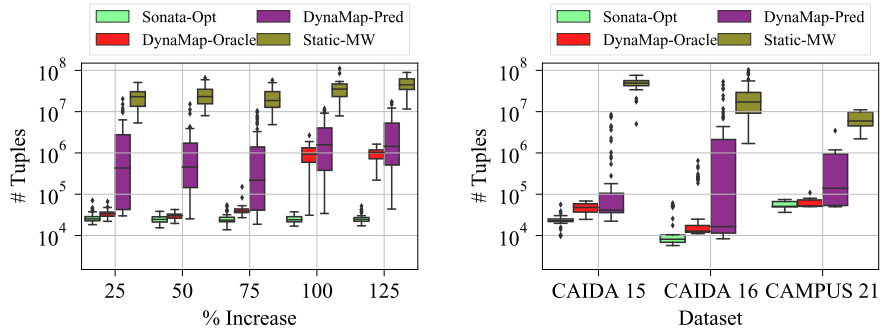
Targets. Since switches have finite resource constraints, we choose to evaluate DynaMap with simulated PISA switches. These simulated switches can vary in terms of their number of stages, the number of stateful ALUs per stage, and memory per ALU and stage.

Comparison with alternative systems. Prior work such as [9] has already demonstrated improved system performance of Sonata's static query plans that utilize iterative query refinement over existing hybrid network telemetry systems such as Marple [25], UnivMon [20], OpenSketch [45] that lack this capability. Thus, we focus here on examining DynaMap's performance against the performance of `Static-MW`, a representative state-of-the-art hybrid network telemetry system that performs static query planning and is an improved version of Sonata in the sense that it uses median values of required per-operator memories computed over multiple past windows rather than single values (from a single window) as input to Sonata's query planning ILP. In addition, our comparisons concern two versions of DynaMap. `DynaMap-Oracle` is a version of DynaMap that assumes oracular knowledge and allows us to decouple the evaluation of the query planning algorithms from the evaluation of the workload prediction algorithm. In contrast, `DynaMap-Pred` is a version that emulates our proposed end-to-end system DynaMap and enables us to evaluate all the proposed algorithms combined, including the workload prediction algorithm. Finally, we also compare against `Sonata-Optimal`, which uses Sonata's query planning ILP to compute optimal query plans for *each* window. Although `Sonata-Optimal` is impractical as it requires resizing register sizes at runtime, we use it here as the baseline because it minimizes the load on the SP. Some of these systems (e.g.,`DynaMap-Pred`, `Sonata-Optimal`) require us to estimate the additional load at the SP when the required operator memory exceeds the allocated memory in the data plane. We provide details about how we approximate this additional load at the SP in Appendix D.

Methodology. To evaluate the different components of our system, we select particular parameter values and operational settings. For example, for each of the systems, we use the per-operator memory requirements for the first $k = 10$ windows for computing initial query plans. We use the remaining 50 windows to evaluate these systems' performance. In this work, we use a simulated PISA switch as our hardware target, featuring twelve stages, eight stateful ALUs per stage, 1.5 Mb of memory per stage, and 0.75 Mb of memory per ALU. While our choices are inspired by previous work such as [9], we impose much stricter memory constraints compared to [9], which assumes that all switch SRAM resources are available for telemetry. In practical scenarios, only a limited subset of available SRAM memory can be allocated for telemetry tasks. Therefore, we considered only a small fraction (i.e., less than 20%) of the total memory for network monitoring. These constrained settings are not only more realistic but also better suited for exploiting the benefits of dynamic query planning. Abundant data-plane resources in certain settings negate the need for sophisticated query planning.

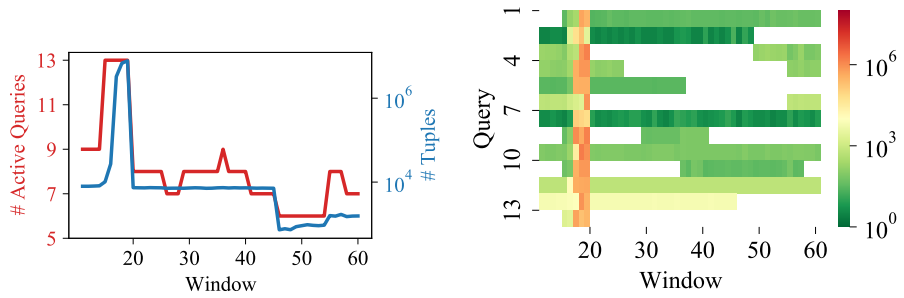
5.2 Main Results

DynaMap-Oracle: Robust to changing traffic workloads. Figure 3a shows `DynaMap-Oracle`'s performance for different synthesized workloads that are given in the form of operator memory requirements with prescribed variability (*i.e.,* scaled-up CoV values) and are generated as described



(a) Synthetic workload. (b) Production workload.

Fig. 3. **Robust to changing input workloads.**



(a) Dynamic workload (b) SP's load breakdown

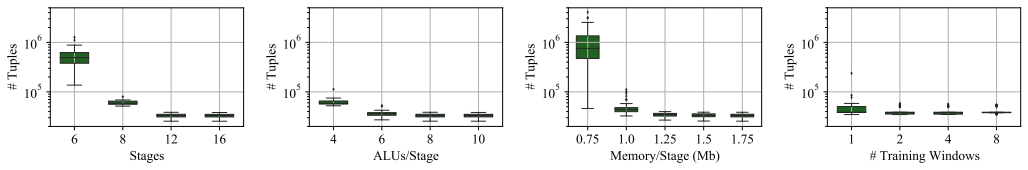
Fig. 4. **Robust to dynamic query workloads.**

in Section 5.1. Here, for each scale-up factor, we report the distribution of the number of tuples—aggregated across all eight queries—sent to the stream processor across fifty windows, represented as box plots. As reported in prior work [9], it is important to note that since different queries focus on varying subsets of the incoming traffic, they contribute differently to this overhead. For smaller scale-up factors (e.g., $25-75\%$), the performance gains over `Static-MW` are consistently some three orders of magnitude, and the difference in performance compared to `Sonata-Optimal` is uniformly less than half an order of magnitude. As the variability increases (*i.e.,* scale-ups of 100% and more), `DynaMap-Oracle`'s performance degrades compared to `Sonata-Optimal`. This degradation stems from the limited opportunities for reallocation when operator memory requirements vary drastically from one window to the next.

Figure 3b shows the performance of `DynaMap-Oracle` and its counterparts on each of our three real-world traffic workloads. We observe that `DynaMap-Oracle` can effectively adapt to input workload dynamics encountered in production traffic as well. In particular, we observe that `DynaMap-Oracle` outperforms `Static-MW` on average by around three orders of magnitude for the two CAIDA datasets. The smaller difference in performance for the `CAMPUS-21` dataset is attributable to the relatively smaller variability in operator memory requirements for this dataset.

DynaMap-Oracle: Robust to changing query workloads.

To demonstrate that DynaMap's ability to update the operator-mapping decisions at runtime also makes it robust to changing query workloads, we synthesize dynamic query workload instances using the parameters $L = 8$ and $\lambda = 1/3$ (thus, $\mu \approx 24$ windows of size w). That is, we consider



(a) # stages (b) # ALUs per stage (c) Memory per stage (d) Training windows

Fig. 5. Sensitivity analysis: DynaMap's performance for the different data-plane targets in (a), (b), and (c), and different number of training windows in (d).

a query workload with eight active queries on average and where new queries arrive at a rate $\lambda = 1/3$ (with any newly arriving query starting at the beginning of a window). Assuming that each query has two refinement levels on average, the average length of each query is $\mu + 2 = 26$ windows. Figure 4a depicts one realization of our dynamic query workload model (red line) that we further modify by ensuring that between windows $15 - 19$, all 13 queries are active to mimic a short period of intense data plane resource requirements ("over-subscription"). We observe that during this period, the load at the SP (blue line) increases but recedes to smaller values after the end of this period. This behavior is highlighted in the heatmap plot in Figure 4b that shows the "life" of each active query during the 60 window-long period at the level of individual operators. The color encodes an operator's contribution to the SP's load. The figure visualizes how `DynaMap-Oracle` can find suitable mappings for most operators most of the time. However, during the over-subscription period, it cannot fit many of the operators in the data plane, thus increasing the SP's load.

DynaMap-Pred: DynaMap with prediction. Due to its use of oracular knowledge (i.e., no prediction), `DynaMap-Oracle`'s performance serves as a lower bound for what is achievable with end-to-end systems like ours that have to predict the future input workload, including future per-operator memory requirements, to work in practice. To quantify the "cost of prediction", we use our proposed end-to-end system `DynaMap-Pred` that includes a module for prediction and compare its performance with that of `DynaMap-Oracle`, both for synthetic workloads (Figure 3a) and production traffic workloads (Figure 3b). We observe that using prediction instead of oracular knowledge increases the stream processor's load on average by about an order of magnitude. This increase is attributed to prediction errors that result in over-provisioning or under-provisioning for a subset of operators and contribute to higher loads at the SP.

5.3 Sensitivity Analysis

We next quantify DynaMap's sensitivity to various data-plane constraints and training windows.

Data-plane constraints. In practical scenarios, the hardware targets DynaMap uses for data-plane query execution often differ in their configurations. To understand DynaMap's effectiveness with various data-plane configurations, we analyzed the impact of different numbers of stages and different numbers of ALUs per stage while maintaining constant total memory. Figure 5a and 5b demonstrate that increasing the number of stages or the number of per-stage ALUs makes dynamic query planning more effective. This observation agrees with the intuition that more stages or ALUs afford DynaMap more flexibility for dynamic operator mapping at runtime. However, we observe diminishing returns as we keep increasing these resources.

Given this trend, we expect that DynaMap's performance will degrade further with a decrease in the number of stages. Fewer stages imply fewer ALUs, which implies less flexibility in mapping dataflow operators to data-plane registers at runtime—sending most of the traffic to the stream processor. In such constrained settings, we expect DynATOS to perform better by trading query

accuracy for scalability. We hypothesize that a hybrid approach, integrating DynATOS' intra-window and DynaMap's inter-window query planning techniques, would outperform either method used alone. However, designing such a hybrid system is challenging, especially in the absence of a publicly available codebase for DynATOS, and we leave the task of implementing DynATOS for PISA targets and investigating the performance of such a hybrid approach for future research.

Figure 5c shows that the reductions in the stream processor's load are less pronounced when the total available operator memory is small, indicating that DynaMap's runtime has fewer opportunities to redistribute operator memory when operating in a highly memory-constrained environment. Finally, we observe that the stream processor's load decreases with an increase in memory resources, suggesting that when DynaMap already makes efficient use of available memory for the given traffic and query workload, adding more memory in the data plan has diminishing returns.

Training windows. We now show how the parameter k, i.e., the number of training windows used for selecting the initial refinement plans, affects DynaMap's performance. Figure 5d shows that DynaMap can tolerate using fewer training windows to select the refinement plans at compile-time without incurring any significant performance penalty. This result shows that the initial query plans computed by our compile-time algorithms are effective despite using only 1-2 windows worth of historical data.

6 DISCUSSION

Updating query refinement plans. In its current form, DynaMap does not support changing query refinement plans at runtime. Doing so would require concurrently executing queries corresponding to their new and old refinement plans. However, dealing with the resulting increase in the query workload without compromising query accuracy will lead to a higher load at the SP. In Appendix E,we describe a practical approach to updating query refinement plans at runtime that assumes that updates are only made infrequently (e.g., once an hour or day). We argue that although limited, this capability is of practical interest because it enables DynaMap to handle infrequent but significant shifts in the overall query or traffic workloads over time.

Improving workload predictions. Section 5.2 shows that there is about an order of magnitude gap in performance between `DynaMap-Oracle` and `DynaMap-Pred`. We expect that future versions of DynaMap that implement more advanced learning models for prediction and have access to more training data (i.e., more extended packet traces) will be able to reduce this difference further.

Runtime overhead. In Appendix F, we show that our current prototype takes around 50-60 ms to update the packet-processing pipeline at runtime. While for relatively large window sizes (e.g., three seconds), this overhead has only a marginal impact on query accuracy, its impact is significant for networks and queries requiring sub-second or sub-millisecond windows and highlights the dependence of dynamic query planning on utilized hardware technology. For example, given that the main contributions to the measured runtime overhead result from the transfer of data between the data plane and user space, off-loading user-space tasks to near-switch resources (e.g., executing flexible Spark-streaming tasks on FPGA-based targets [4]) can be expected to result in orders of magnitude reductions in runtime overhead and enable support for running queries at sub-second or sub-millisecond granularities. Such real-time hybrid network telemetry systems will depend critically on dynamic query-planning algorithms to make optimal operator-mapping decisions.

Extension to distributed settings. An ideal network telemetry system should enable the execution of *dynamic query workloads* in *distributed settings* for *dynamic input workloads* at scale [48]. Being both scalable and robust, the proposed DynaMap lays the foundation for system designs that can achieve the stated ideal. However, much work remains, and extending DynaMap to support dynamic query planning beyond the single-switch case to distributed or network-wide settings looms as a promising but challenging problem for future work.

7 RELATED WORK

Modern data plane technologies have had a profound impact on network telemetry [16, 34, 44]. Existing network telemetry systems can be roughly grouped into purely host-based platforms [2, 5, 10, 24, 26, 32, 33, 43, 46]) and data plane-only systems [3, 19, 20, 45, 49]), and include as of late also hybrid designs. Among the latter, we can further distinguish between single-switch (e.g., [9, 21, 47]) and network-wide or distributed solutions (e.g., [11, 17, 18, 23, 25, 30, 31, 42, 48]).

Most existing hybrid telemetry systems do not consider query planning and do not leverage runtime programming capabilities afforded by existing data plane targets. As such, their main focus is on mechanisms for compiling a subset of dataflow operators in the switch by making the best use of readily available compile-time programming capabilities. Noticeable exceptions and most closely related to our work are recently proposed systems such as Sonata [9] and DynATOS [21] that support query planning, and Newton [48] and Flymon [47] that allow for modifying or changing queries on-the-fly (i.e., at runtime, without disrupting normal packet forwarding).

In particular, as a single-switch solution, Sonata relies exclusively on compile-time programming to support static query planning and uses iterative query refinement to achieve scalability. In contrast, while also single-switch, DynATOS leverages runtime programming to support dynamic query planning and considers approximate queries for scalability. While Sonata is unable to deal with the dynamism of traffic and query workloads, DynATOS only considers hardware targets with a single processing stage which limits the types of queries it can run. Similarly, as a single-switch system, Flymon only considers sketch-based queries, does not support query planning, but enables modifying or changing queries at runtime. Newton, on the other hand, is a network-wide solution that also lacks support for query planning but allows for on-the-fly inserting, removing, or updating of Sonata-like queries a scale. In contrast to these existing solutions, DynaMap is a hybrid system for single-switch settings that combines the best of Sonata and DynATOS by supporting dynamic query planning on a commodity switch without compromising flexibility.

8 CONCLUSION

Ensuring that hybrid network telemetry systems are robust to changing traffic and query workloads is critical for enabling network automation in practice. Motivated by the inability of static query planning to achieve this desired robustness, we present the design and implementation of DynaMap, a new hybrid network telemetry system that uses dynamic query planning to ensure robustness. To support making dynamic query planning decisions, it uses runtime programming to update the resource allocation for dataflow operators in the data plane on the fly in response to changing workload conditions; it uses compile-time programming to compute an initial query plan that it configures in the data plane. Our evaluation with both synthetic and production workloads shows that DynaMap's performance is, on average, 1-2 orders of magnitude better than that of its counterparts that can only support static query planning. Our prototype with a Tofino-based switch demonstrates the feasibility of DynaMap.

ACKNOWLEDGMENTS

We thank the anonymous CoNEXT'23 shepherd and reviewers for their comments and suggestions. The views expressed in this paper are those of the authors and do not reflect the official policy or position of the U.S. Military Academy, the Department of the Army, the Department of Defense, or the U.S. Government. Researchers at UCSB, Chaofan Shou, Rohan Bhatia, and Arpit Gupta, were supported by NSF awards CNS-2003257, OAC-2126281, and OAC-2126327, as well as a grant by Verizon Innovation LLC.

REFERENCES

[1] Y. Altmann, S. McLaughlin, and N. Dobigeon. 2014. Sampling from a multivariate Gaussian distribution truncated on a simplex: A review. In *2014 IEEE Workshop on Statistical Signal Processing (SSP)*.

[2] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium*.

[3] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *Proc. ACM SIGCOMM*.

[4] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. 2016. When Apache Spark Meets FPGAs: A Case Study for next-Generation DNA Sequencing Acceleration. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (Denver, CO) *(HotCloud'16)*. USENIX Association, USA, 64–70.

[5] C. Cranor, T. Johnson, O. Spatschek, and V. Shkapenyuk. 2003. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD*.

[6] A. Deshpande, Z. Ives, and V. Raman. 2007. Adaptive Query Planning. In *Foundations and Trends in Databases*.

[7] Edgecore. 2022. Programmable Tofino switches for data centers. https://www.edge-core.com/productsInfo.php?id=335.

[8] Yong Feng, Haoyu Song, Jiahao Li, Zhikang Chen, Wenquan Xu, and Bin Liu. 2021. In-Situ Programmable Switching Using RP4: Towards Runtime Data Plane Programmability. In *Proc. ACM HotNets'21*.

[9] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. 2018. Sonata: Query-driven network telemetry. In *Proc. ACM SIGCOMM*.

[10] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. 2014. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. NSDI*.

[11] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. 2020. Carpe Elephants: Seize the Global Heavy Hitters. In *Proc. of ACM SIGCOMM SPIN Workshop (SPIN '20)*.

[12] J.M. Hellerstein. 2017. Query optimization. In *In P. Bailis, J.M. Hellerstein, and M. Stonebraker, editors, Readings in Database Systems (Chapter 7)*.

[13] Heather Hulett, Todd G. Will, and Gerhard J. Woeginger. 2008. Multigraph realizations of degree sequences: Maximization is easy, minimization is hard. *Oper. Res. Lett.* 36, 5 (2008), 594–596.

[14] Mobin Javed and Vern Paxson. 2013. Detecting stealthy, distributed SSH brute-forcing. In *ACM SIGSAC Conference on Computer & Communications Security*. 85–96.

[15] Jaeyeon Jung, Vern Paxson, Arthur W Berger, and Hari Balakrishnan. 2004. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*. IEEE, 211–225.

[16] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L.J. Wobker. 2015. In-band network telemetry via programmable dataplanes. In *Proc. SOSR*.

[17] Y. Li, K. Gao, X. Jin, and W. Xu. 2020. Concerto: Cooperative Network-Wide Telemetry with Controllable Error Rate. In *Proc. APSys*.

[18] Y. Li, R. Miao, C. Kim, and M. Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Proc. NSDI*.

[19] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *ACM SIGCOMM*.

[20] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM*.

[21] Chris Misa, Walt O'Connor, Durairajan Ramakrishnan, Reza Rejaie, and Walter Willinger. 2022. Dynamic Scheduling of Approximate Telemetry Queries. In *Proc. USENIX NSDI'22*.

[22] MoonGen. 2021. MoonGen Packet Generator. http://scholzd.github.io/MoonGen/.

[23] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. 2014. DREAM: dynamic resource allocation for software-defined measurement. In *Proc. ACM SIGCOMM*.

[24] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 129–143.

[25] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. ACM SIGCOCMM*.

[26] opensoc 2015. OpenSOC. http://opensoc.github.io/.

[27] P4. 2022. P4Runtime Specification. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html.

[28] report [n. d.]. Apache Flink. http://flink.apache.org/.

[29] Richard Serfozo. 1999. Introduction to Stochastic Networks.

[30] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*. 1–16.

[31] J. Sonchack, O. Michel, A.J. Aviv, E. Keller, and Smith J.M. 2018. Scaling Hardware Accelerated Monitoring to Concurrent and Dynamic Queries With ∗Flow. In *Proc. ATC*.

[32] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 233–248.
[33] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2018. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 453–456.
[34] Y. Tokusashi, H.T. Dang, F. Pedone, R. Soule, and N. Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proc. EuroSys*.
[35] url [n. d.]. Apache Spark. http://spark.apache.org/.
[36] url [n. d.]. Barefoot's Tofino. https://www.barefootnetworks.com/technology/.
[37] url [n. d.]. Sonata Queries. https://github.com/sonata-queries/sonata-queries.
[38] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and TS Eugene Ng. 2022. Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon. In *USENIX NSDI*.
[39] Wikipedia. 2022. F-score. https://en.wikipedia.org/wiki/F-score.
[40] Peter R Winters. 1960. Forecasting sales by exponentially weighted moving averages. *Management science* 6, 3 (1960), 324–342.
[41] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Hongyi Liu, Matty Kadosh, Alan Lo, Aditya Akella, Thomas Anderson, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. 2021. A Vision for Runtime Programmable Networks. In *Proc. ACM HotNets'21*.
[42] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 561–575.
[43] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 207–220.
[44] M. Yu. 2019. Network Telemetry: Towards A Top-Down Approach. In *ACM SIGCOMM Computer Communication Review*.
[45] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*.
[46] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. 2017. Quantitative Network Monitoring with NetQRE. In *Proc. ACM SIGCOMM*.
[47] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. 2022. FlyMon: enabling on-the-fly task reconfiguration for network measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 486–502.
[48] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. 2020. Newton: intent-driven network traffic monitoring. In *ACM CoNEXT*. 295–308.
[49] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*.

APPENDIX A.1: NP HARDNESS PROOF FOR SOM

We will prove that a variant of the Simplified Optimal Mapping (SOM) problem is NP-hard by reduction from the 3-Partition problem. First, we define a decision-problem variant of SOM starting with all of the inputs to the original SOM problem. The question is to determine if there exists an assignment (α) with cost of at most some non-negative real number (K). The 3-Partition problem consists of an input set $X = \{x_1, \dots x_n\}$ of positive integers, together with a positive integer T; here, n is a multiple of 3. The question is to determine whether there exists a partition of X into disjoint subsets $\{X_1, \dots, X_{n/3}\}$ of X so that for every $1 \le j \le n/3$ it holds that (1) $|X_j| = 3$ and (2) $\sum_{x_i \in X_j} x_i = T$. We refer to such a partition as a *solution partition.* It is known (see for example [13]) that 3-Partition remains NP-hard even when every $x_i \in X$ satisfies $T/4 < x_i < T/2$. This guarantees that any partition of X into $X_1, \dots, X_\ell$ that satisfies requirement (2) also automatically satisfies requirement (1) and therefore is a solution partition.

Proof. We give a reduction from the variant of the 3-Partition problem where every $x_i \in X$ satisfies $T/4 < x_i < T/2$. Given an instance $(X = \{x_1, \dots, x_n\}, T)$ of 3-Partition, we create an input to SOM as follows. The set R has n registers $\{r_1, \dots, r_n\}$, and for every $i \le n$, the register r_i has capacity p_{r_i}. The set O has $n/3$ operators $\{o_1, o_2, \dots, o_{n/3}\}$. Every operator o_j has size $s_{o_j} = T$, satisfied cost $c^s_{o_j} = 0$, and an unsatisfied cost $c^u_{o_j} = 1$. This concludes the construction. We now prove that the instance (X, T) of 3-Partition has a solution partition if and only if the constructed instance T, R has an assignment α of cost at most $c(\alpha) = \sum_{o_j \in O} c^s_{o_j} = K = 0$.

For the forward direction, suppose that there exists a solution partition $X_1, \dots, X_{n/3}$ of X. Define the assignment $\alpha : R \to O$ as follows: for every $r_i \in R$, determine the unique j for which $r_i \in X_j$ and set $\alpha(r_i) = o_j$. To show that $c(\alpha) = \sum_{o_j \in O} c^s_{o_j}$, it is sufficient to verify that $\sigma(o_j, \alpha) = 1$ for every operator $o_j \in O$. In particular, we have that

$$\sigma(o_j, \alpha) = \frac{\sum_{r_i \in R \,:\, \alpha(r_i) = o_j} p_{r_i}}{s_{o_j}} = \frac{\sum_{x_i \in X \,:\, x_i \in X_j} x_i}{T} = 1$$

We conclude that $c(\alpha) = \sum_{o_j \in O} c^s_{o_j}$, as claimed.

For the reverse direction, suppose there exists an assignment α of cost at most $c(\alpha) = \sum_{o_j \in O} c^s_{o_j}$. Since

$$c(\alpha) = \sum_{o_j \in O} c(o_j, \alpha) = \sum_{o_j \in O} c^s_{o_j},$$

it holds that $c(o_j, \alpha) = c^s_{o_j}$ for every $o_j \in O$. Hence, for every $o_j \in O$, we have $\sum_{r_i \in R \,:\, \alpha(r_i) = o_j} p_{r_i} \ge s_{o_j} = T$. Furthermore, since

$$\begin{aligned} T \cdot \frac{n}{3} &= \sum_{r_i \in R} c_i \\ &= \sum_{o_j \in O} \sum_{\substack{r_i \in R \text{ s.t.} \\ \alpha(r_i) = o_j}} c_i \\ &\ge \frac{n}{3} \cdot T \end{aligned}$$

and the first term of the chain is equal to the last, the last inequality must hold with equality. It follows that $\sum_{r_i \in R \,:\, \alpha(r_i) = o_j} p_{r_i} = T$ for every $o_j \in O$.

For every $j \le n/3$, we set $X_j = \{x_i \;:\; \alpha(r_i) = o_j\}$. We have that for every j,

$$\sum_{x_i \in X_j} x_i = \sum_{r_i \in R \,:\, \alpha(r_i) = o_j} c_i = T.$$

Thus the partition $X_1, \ldots, X_{n/3}$ of X satisfies requirement (2) of solution partitions. Since every x_i is in the range $(T/4, T/2)$ the partition also satisfies the requirement (1), that $|X_j| = 3$, of solution partitions. We conclude that $X_1, \ldots, X_{n/3}$ is a solution partition, completing the proof. □

Remarks: Our proof shows that it is in fact NP-hard to determine whether a given instance admits an assignment of cost 0. Thus, assuming P ≠ NP, the Optimal Assignment problem can not admit an approximation algorithm with *any* factor.

Moreover, the choice of a satisfied cost of 0 and an unsatisfied cost of 1 for every operator in our proof might suggest that the NP-hardness result only kicks in for instances with unrealistic choices of costs. However it is easily verified that the proof does not rely on the precise choice of costs and goes through for any selection of operator costs, as long as $c^s_{o_j} < c^u_{o_j}$ for every operator $o_j \in O$.

Finally, our proof also shows that the SOM problem is strongly NP-hard; *i.e.,* it remains NP-hard even when all input integers are coded in unary.

APPENDIX A.2: TOM-BASED HEURISTIC ALGORITHM

Algorithm 2: TOM-based Heuristic Algorithm

```
1 Input: Q, D;
2 l_min ← ∞;
3 for Th ∈ [0, M] do
4     p ← get-refinement-plan(Q, Th, D);
5     l ← get-load(p, D);
6     if l ≤ l_min then
7         l_min ← l; p_min ← p;
8 return p_min, l_min
```

Given a set of input queries (Q), the proposed algorithm, referred to as TOM-based heuristic (see Algorithm 2), uses a set of operator memory requirements (D) as input and iterates over a range of threshold values Th, starting with $Th = 0$ (*i.e.,*, selecting plans that use all refinement levels). For each threshold value, the algorithm considers the refinement plan obtained by applying the function `get-refinement-plan`. This function first determines for each query the refinement plan with the smallest TOM among all refinement levels. The function then chooses the smallest refinement level (and the corresponding refinement plan) for which Δ-TOM ≤ Th (line 4). Here, Δ-TOM denotes the rate of decrease of a query's memory footprint as the number of refinement levels increases. For the chosen refinement plan, the function then computes the average SP load after applying the proposed dynamic operator mapping algorithm at runtime for the given input (*i.e.,* operator memory requirements) (line 5). At the end of this iterative process, the algorithm returns the refinement plan with the smallest average SP load (line 8).

Remark: This heuristic readily generalizes to the dynamic query workload case by considering different samples of subsets of queries from the pool of all input queries. For each such sample, we use Algorithm 2 to compute the refinement plan for each query in this sample and finally select the refinement plan for each query that is the most popular one across all the samples.

APPENDIX B: TOFINO-BASED IMPLEMENTATION

Below, we detail how we implemented different functionalities in a Tofino-based switch to enable dynamic operator mappings. As described in Section 4.2, enabling dynamic operator mappings

```
action do_execute_reduce() {
    reduce_program.execute_stateful_alu(meta_op.index);
}

action do_execute_distinct() {
    distinct_program.execute_stateful_alu(meta_op.index);
}

action drop_exec_op(){
    modify_field(
        meta_app_data.drop_exec_op, 1);
 }

table execute_op {
    reads {
        meta_op.select_prog : exact;
    }
    actions {
        do_execute_reduce;
        do_execute_distinct;
        drop_exec_op;
    }
    size : 2;
}
```

Fig. 6. Using packet's metadata field and match-action table to select the SALU program for stateful operations.

in the data plane entails dynamically changing (1) the program for each stateful arithmetic logic unit (SALU) for each register in the data plane, (2) the set of keys used for stateful operations, and (3) the packet-processing pipeline in the data plane.

Updating SALU programs. Figure 6 shows how DynaMap uses a table, `execute_op`, to dynamically change a SALU's program. For this table, it defines actions executing the `reduce` and the `distinct` program, respectively. This table reads a packet's `select_prog` metadata field to decide which action (*i.e.,* SALU program) to select for stateful operations.

Updating the set of keys for stateful operations. To understand how DynaMap dynamically updates the keys for stateful operations, consider the case where the two keys are, say, `dIP` and `sIP`, and DynaMap needs to dynamically decide which of the two to use for computing the index value for the stateful operation. Figure 7 shows how DynaMap uses the `init_hash_field_data` table to apply two different actions. Each of these actions reads the packet field value from the packet, applies the mask (for iterative refinement), and writes the masked value to the packet's metadata field. Figure 8 shows how DynaMap uses this metadata field to compute the hash index for stateful operation. Extending this code block to enable hashing on multiple packet fields is straightforward. Here, instead of creating a single metadata field, DynaMap creates multiple metadata fields and then changes their values using tables specific to each of these fields.

Updating the packet-processing pipeline. Dynamically mapping stateful operators to different registers in the data plane requires updating the packet processing pipeline to ensure the sequential composition of these operators. Sonata used a combination of the query-specific metadata field `qid_drop` and `IF` statements to compose dataflow operators in the data plane. Such an approach is only suited for static operator-to-register mappings. DynaMap's control program applies all the match-action tables in sequence and uses a packet's metadata fields to decide whether or not to

```
action do_init_src_ip(dynamic_mask) {
    bit_and(meta_op.field_value, ipv4.srcIP, dynamic_mask);
}

action do_init_dst_ip(dynamic_mask) {
    bit_and(meta_op.field_value, ipv4.dstIP, dynamic_mask);
}

table init_hash_field_data {
    actions {
        do_init_src_ip;
        do_init_dst_ip;
    }
    size : 1;
}
```

Fig. 7. Match-action tables to select the key for stateful operation.

apply particular actions in the match-action table to it. Such an approach enables DynaMap to dynamically reconfigure the packet-processing pipeline at runtime.

To enable such flexible packet processing, DynaMap narrows the scope of the `drop` metadata field to match-action tables for stateful operators. The decision of whether the remaining downstream operators should be applied to this packet is encoded in this `drop` field. Such a change ensures that DynaMap can compose these operators without requiring `IF` statements.

DynaMap also enables support for multiple compile-time dependencies. For example, if there are two operators a and b in the first stage and c and d in the second stage, it enumerates all possible dependencies between them, *i.e.,* $a \rightarrow c$, $a \rightarrow d$, $b \rightarrow c$ and $b \rightarrow d$. To realize these dependencies, it makes the operators in the second stage (*i.e.,* c and d) match on both a and b's 'drop' metadata. At runtime, DynaMap adds match-action table entries such that only one of the two possible dependencies actually occurs for these operators.

Note that although making an exact match on N drop fields (where N denotes the number of operators in the last stage) requires 2^N match-action table entries at runtime, an operator in the second stage only follows a single operator in the first stage, which in turn enables DynaMap to use a single ternary match-action rule.

Metadata fields. So far, we described how DynaMap uses different metadata fields to dynamically update the packet processing pipeline at runtime. We now summarize the type of metadata fields DynaMap uses and provide an estimation of their memory overhead.

- **Ephemeral fields.** For each data-plane register, DynaMap needs the following metadata fields: (1) one bit `select_program` field to change the SALU program at runtime; (2) one bit `drop` field to mark if the packet needs further processing; (3) a `index` field of size log_2(register size) bits to store the index for executing a stateful operation; (4) a `keys` field for storing the packet fields it uses as keys for the stateful operation. All these fields are narrowly scoped, can be reused across stages, and are therefore termed *ephemeral*. A single register requires around 114 bits (1 for SALU program + 1 for drop + 96 for keys + 16 for index). Thus, for the target considered in this paper (8 stages), DynaMap requires around 1.8 Kb for a packet's metadata for ephemeral fields.
- **Persistent fields.** For each query, DynaMap needs the following metadata fields: (1) a 16-bit `qid` field for each query that gets executed in the data plane; (2) `keys` to store the packet fields that are used as keys for computing the index value for query's last stateful operator in the data plane; and (3) `index` field to store the hash value for the key fields. The index field enables DynaMap's

```
field_list hash_op_fields {
    meta_op.field_value;
}

field_list_calculation hash_op_calc {
    input {
        hash_op_fields;
    }
    algorithm: hash_algo;
    output_width: hash_width;
}

action do_init_hash() {
    modify_field_with_hash_based_offset(
        meta_op.index, 0,
        hash_op_calc, reg_1_size);
}

table init_hash {
    actions {
        do_init_hash;
    }
    default_action : do_init_hash;
    size : 1;
}
```

Fig. 8. Match-action tables to compute the hash value for the selected the key for stateful operation.

runtime to read the aggregated `value` from the data plane, which in turn enables DynaMap to emit the output (key, value) pair to the stream processor for further processing. For each query, DynaMap requires 128 bits in a packet's metadata for these persistent fields, which is similar to Sonata's metadata footprint. This number will increase as the number of input queries increases.

DynaMap introduces an additional optimization to avoid wasting register memory for `distinct` operation. More concretely, for `reduce` operation, DynaMap configures a two-dimensional register memory specifying the number of rows, where each row is of size 32 bits. For the match-action table applying the `reduce` operator, it updates the value in the row identified by the index field. Naively, using the same approach for `distinct` operator will waste 31 bits for each row as it only requires one bit to store the state. To ensure that the SALU programs for `distinct` operator can make use of all available bits in the register, DynaMap uses five additional bits for computing the hash. These five bits enable the match-action table to select a bit among the 32 bits in each row. This optimization further requires an additional 80 bits in the packet's metadata for the target considered in the paper.

APPENDIX C: TELEMETRY QUERIES

In addition to the eight queries considered in previous work [9] (*i.e.,* queries 1-8 in Table 1, we also synthesized five additional queries (*i.e.,* queries 9-13 in Table 1). These five queries are the mirrored versions (*i.e.,* flipping the keys used for stateful operations) of a subset of the original eight queries. Here, different queries have varying resource requirements. For instance, Query 1 involves only one stateful operation, while Queries 2-5 and 7 involve two, and Queries 6 and 8 require three stateful operations [37] Similarly, the memory requirement for the stateful operation in Query 1 is an order of magnitude greater than that for Query 2.

#	Query	Description
1	Newly opened TCP Conns. [46]	dstIPs for which the number of newly opened TCP connections exceeds the threshold.
2	SSH Brute Force [14]	dstIPs that receive similar-sized packets from more than threshold unique senders.
3	Superspreader [45]	srcIPs that contact more than threshold unique destinations.
4	Port Scan [15]	srcIPs that send traffic over more than threshold destination ports.
5	DDoS [45]	dstIPs that receive traffic from more than threshold unique sources.
6	TCP SYN Flood [46]	dstIPs for which the number of half-open TCP connections exceeds threshold Th.
7	TCP Incomplete Flows [46]	dstIPs for which the number of incomplete TCP connections exceeds the threshold.
8	Slowloris Attacks [46]	dstIPs for which the average transfer rate per flow is below the threshold.
	Additional Queries	
9	Mirrored 1	srcIPs for which the number of newly opened TCP connections exceeds the threshold.
10	Mirrored 2	srcIPs that send similar-sized packets to more than threshold unique receivers.
11	Mirrored 4	dstIPs that send traffic to more than threshold destination ports.
12	Mirrored 6	srcIPs for which the number of half-open TCP connections exceeds threshold Th.
13	Mirrored 7	srcIPs for which the number of incomplete TCP connections exceeds the threshold.

Table 1. Telemetry queries for generating query workloads.

APPENDIX D: LOAD AT STREAM PROCESSOR

As part of our evaluation, we extended Sonata's simulator to emulate each of the alternative query-planning techniques. Some of these techniques (e.g., DynaMap-Pred, Sonata-Optimal) require us to estimate the additional load at the stream processor when the required operator memory exceeds the allocated memory in the data plane. While we can compute this load accurately using a packet-level simulation, such an approach is prohibitively slow. Instead of packet-level simulation, Sonata uses the stream processor to aggregate operator memory requirements for each window and uses these values to estimate the stream processor's load. However, such aggregation has the disadvantage that it loses all information about the order in which packets are processed in the data plane.

Estimating the load at the stream processor. To address this issue, we approximate the additional load using the *average* value. Specifically, we report the product of the average number of tuples-per-key and the number of keys that cannot fit in the data plane register. For example, if the average number of tuples-per-key for an operator is 10 and given the register size, five keys cannot fit in, then our estimate of the additional load at the stream processor will be 50.

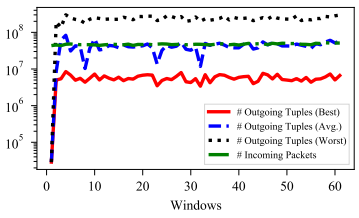


Fig. 9. Output workload because of a Sonata's static query plan compared to the input workload. The output workload is estimated, and the figure shows the best, worst, and average cases.

Note that for a given set of queries and an input workload, for each of the queries' operators, we can compute the three metrics: $(N_{in}, N_{out}, B_{req})$. Here, for a given operator, N_{in} denotes the number of input tuples for the operator, N_{out} the number of output tuples after applying the stateful operation, and B_{req} is the required operator memory. The memory allocated for this operator in the data plane is denoted by B_{alloc}. If B_{alloc} is greater than B_{req}, the data-plane register is over-provisioned. In this case, the number of tuples sent to the stream processor (N_{sp}) is given by $N_{sp} = N_{out}$. For the cases where the data-plane register is under-provisioned (*i.e.*, $B_{alloc} < B_{req}$), we estimate the load due to keys that find an entry in the data plane to be $N_{out}\frac{B_{alloc}}{B_{req}}$ and the additional load due to hash collision to be $N_{in}\frac{B_{req}-B_{alloc}}{B_{req}}$. Therefore, the total load due to under-provisioning is estimated as $N_{sp}^{\text{avg}} = N_{out}\frac{B_{alloc}}{B_{req}} + N_{in}\frac{B_{req}-B_{alloc}}{B_{req}}$.

```
1  packetStream(W)
2  .filter(p => p.proto == 17)
3  .map(p => (p.dIP, p.sIP))
4  .distinct()
5  .map((sIP, dIP) => (dIP,1))
6  .reduce(keys=(dIP,), f=sum)
7  .filter((dIP, count) => count > Th)
```

Query 2. Detect DDoS Attacks.

Regarding the calculations used for estimating the best and the worst case load estimates, recall that in the best case, the keys with only a single tuple arrive last. We can express this load as $N_{sp}^{\text{best}} = N_{out} + \frac{B_{req} - B_{alloc}}{B_{key}}$. Here, B_{key} denotes the memory footprint of a single key. In the worst case, all the single-tuple keys arrive first and use all the allocated memory in the data plane. We can express this load as $N_{sp}^{\text{worst}} = N_{in} - \frac{B_{alloc}}{B_{key}}$.

Figure 9 shows the load at the stream processor for each of these three cases. The experiment is similar to the one we conducted for Figure 2. In particular, we used the first window's operator memory requirements to compute a static query plan using Sonata's query planner. We then apply this plan to all 60 windows. We used the three different expressions described above to compute the load at the stream processor for average, best, and worst case, respectively.

APPENDIX E: ACCURATE QUERY EVALUATION

Handling under-provisioned data-plane registers. Under-provisioned data-plane targets can result in an increase in hash collisions. If a packet's key generates a collision, we send it to the stream processor. We then perform the aggregation operation for all the packets belonging to this key at the stream processor and join it with the result of the ones aggregated in the data plane to report the final output. This partial execution approach works well if we observe hash collisions for the query's last operator that gets executed at the switch. However, if the collision occurs for an intermediate aggregation operation, sending the packet to the stream processor affects accuracy. For example, consider Query 2 to detect victims of DNS reflection attacks. If the query plan is such that we execute up to line 6 in the data plane, then handling hash collisions for the `distinct` operator is non-trivial. More concretely, if we apply the remaining operators for the keys that observe collision for the `distinct` operator and join it with the output of the `reduce` operator for the other keys, the output will be different. To illustrate, consider the set of input tuples as $\{(a, b), (a, b), (a, c), (c, d)\}$, and threshold (line 7) of 2. Due to collision, we send the tuple (a, c) to the stream processor. Here, with partial query execution, the output is a null set, different from the actual output $(a, 2)$.

To ensure accurate query evaluation, one option is to send the output of the `distinct` operator in the case of a collision to the stream processor, bypassing the remaining operators in the data plane. The other option is to send the output of the `distinct` operator back to the data plane. Given the complexity of realizing this second option, we chose to use the first option.

Changing refinement plans. To explain why changing the refinement plan at runtime affects a query's accuracy, we consider the strawman approach that abruptly changes the refinement plan from one plan to the other. To recall, iterative refinement works by filtering the input workload of an operator A based on the previous window's output of another operator B. The operator B executes at the previous refinement level relative to that of the operator A. Thus, if DynaMap changes the refinement plan in a window, the filtering mechanism is severely impaired. Consider a concrete example – refinement plan for window w_1 is $0 \rightarrow 8 \rightarrow 24 \rightarrow 32$ (call it R_1) and the refinement plan for the successive window w_2 is $0 \rightarrow 16 \rightarrow 28 \rightarrow 32$ (call it R_2). Note that the output from

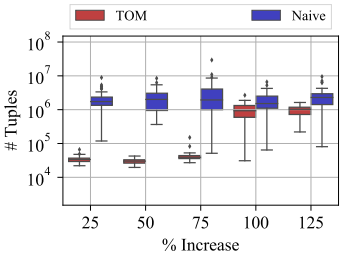
the operator executing at /8 in window w_1 would be used for filtering the input workload for the operator executing at /24 in window w_2. However, when abruptly changing from R_1 to R_2, there is no operator executing at /24 in w_2. Moreover, the new operators introduced in w_2 (e.g., /28) lack the information to apply the filter operation for their input workload. In essence, changing refinement plans at runtime requires making these abrupt changes to refinement plans, but making such changes affects the query's accuracy.

To preserve query accuracy, we can iteratively change the refinement plan over multiple consecutive windows. In the example described above, instead of adding all the operators for refinement plan R_2 in w_2, we only add operator /16 in w_2, /28 in w_3, and so on. This approach never introduces any operators that lack the information to create an input workload filter in any window. Since this approach takes l windows to modify the plan (here l is the number of refinement levels), it is not suited for making changes at runtime. However, the technique can be used in practice when the frequency of changing refinement plans is of the order of hours or days. In this case, the changes are intended to handle significant shifts in the overall traffic patterns. A related problem concerns predicting the memory requirements for new operators. In particular, the proposed approach requires estimating the memory requirements for new operators at different granularities for which we have no historical data. Estimating the memory requirements of such operators will require a more sophisticated learning model that leverages the spatial relationship between operators at different levels of granularity. We leave this exploration for future work.

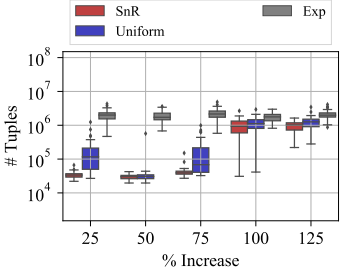
APPENDIX F: ADDITIONAL RESULTS

Role of the compile-time algorithms. We perform two experiments using our controllable synthesized input workloads to show how well the two proposed compile-time algorithms aid our runtime algorithm in making efficient operator mapping decisions. In the first experiment, we replace our TOM-based heuristic (for refinement plan selection) with Naive, a simple alternative that selects a refinement plan at random from the set of all feasible refinement plans, but we retain the described SnR heuristic (for register sizing).

For the second experiment, we retain our TOM-based heuristic as the compile-time algorithm for refinement plan selection but replace the described SnR heuristic for register sizing with two naive alternatives, Exp and Uniform. Here, Exp is a heuristic that allocates register sizes in increasing powers of two (high variability), and Uniform is a heuristic that creates registers that are all of roughly the same size (low variability), and both heuristics use register sizes that are powers of two so that the available data-plane memory is utilized effectively. Figure 10a shows that for workloads with smaller prescribed scaled-up variability in operator memory requirements, selecting refinement plans with our TOM-based algorithm results in significant performance gains over using Naive and aids the runtime algorithm to make effective operator mapping decisions. At the same time, Figure 10b demonstrates that while the difference in performance between using the SnR vs. Uniform is small, the Exp heuristic performs poorly. These differences indicate that low-variability register sizes are generally less helpful in enabling effective operator mappings at runtime than high-variability register sizes. Together, Figure 10a and Figure 10b show that as the variability in operator memory requirements increases by more than 100%,



(a) Refinement plans.



(b) Register sizes.

Fig. 10. **DynaMap's compile-time algorithms are effective.**

all algorithms struggle, because in these scenarios, remapping
operator memories at runtime is challenging, irrespective of how the initial registers have been sized.

Runtime overhead. Compared to existing telemetry systems, DynaMap adds runtime complexity that materializes in runtime overhead. In particular, at the end of each window, DynaMap performs the following tasks: (1) read values of all registers in the data plane through packet ferries (t_1); (2) predict per-operator memory requirements for the next window and then compute new operator-register mappings (t_2); (3) update match-action table entries to apply new operator-register mappings for the next window (t_3), and (4) clear all register values for the next window (t_4);. Here, Task 3 has to wait for the completion of Task 2, which is dependent on Task 1's completion. Task 4 is only dependent on Task 1. Thus, we can express the runtime overhead as $\max\{(t_1+t_2+t_3), (t_1+t_4)\}$.

To evaluate this runtime overhead, we conduct micro-benchmarking experiments with a Tofino-based switch. More precisely, we use a simple testbed that consists of two servers (S1 and S2), each with Intel Xeon E5-2650L V4 14-core processors and Intel's X710-DA4 10 Gbps network adapter, and a Tofino-based switch (Edgecore WEDGE100BF-32X [7]). We use `scapy` to create specially crafted packets and rely on MoonGen [22] to send them to the switch at scale ($\approx$ 11 *Mpps*). We use S2 to send the runtime commands (e.g., reset registers) via the BFRuntime gRPC API [27]. We configure the switch to add timestamps to the outgoing packets' header before sending them to S2. We use `tcpdump` to capture packets at S2 and extract their timestamp values to quantify each of the task's completion times. The results of these experiments show that the average completion time for each of the four tasks is 11 ms, 15 ms, 25 ms, and 25 ms, respectively. Overall, our current prototype takes around 50-60 ms on average to complete all these tasks at runtime.

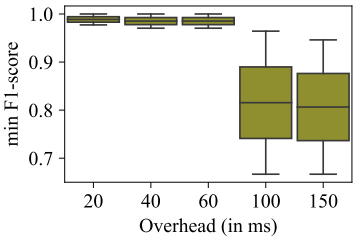


Fig. 11. **Runtime overhead vs. accuracy.**

This runtime overhead results in a delay in updating the packet processing pipeline at the start of a window, and this delay can affect the queries' accuracy. Figure 11 quantifies this relationship between delay and query accuracy. For a given runtime overhead (*i.e.,* period of delay) on the x-axis, we quantify the accuracy of each query by discarding all the packets received during that period at the start of each window. For each data point on the x-axis, we report the distribution of minimum accuracy, quantified as minimum F1-score [39] (*i.e.,* worst accuracy) among all queries over 50 windows. We observe that for the given query workload, as long as the total runtime overhead is less than 60 ms, the impact on query accuracy is marginal. We leave efforts to further reduce this runtime overhead for future work.

Received June 2023; revised December 2023; accepted January 2024