FARM: Comprehensive Data Center Network Monitoring and Management[1]

Jérôme Graf*†, Pavel Chuprikov*, Patrick Eugster*, Patrick Jahnke†
*Università della Svizzera italiana (USI)
†SAP SE

***Abstract*— Modern data centers face growing workloads, putting accrued pressure on network monitoring solutions necessary for ensuring correct and efficient operation. Advances in network programmability have meanwhile led to yet more monitoring data being straightforwardly collected from switches, exacerbating bottlenecks in corresponding collection-centric approaches. This limits scalability and responsiveness, especially when several monitoring tasks are deployed side-by-side, as is common for network management. We present a novel and comprehensive selection-centric solution for network monitoring and management (M&M) called FARM that significantly simplifies the development and deployment of network M&M tasks while being effective and scalable. FARM's main novelty lies in its comprehensive design. Instead of focusing solely on individual parts of network monitoring, FARM takes a global perspective on the problem and aligns all of its components correspondingly: a strongly decentralized software architecture, a specifically designed programming model, and an integrated performance optimization framework. In short, FARM performs monitoring (re)actions locally on switches to the extent possible, using centralized components only if and when needed, and globally optimizes placement, considering placement constraints intrinsically expressed through its programming model as well as commonalities among tasks. Deployed in a production data center, FARM shows significant gains in responsiveness (up to 3427× faster over recent generic approaches and 4× faster over highly specialized solutions), and savings in network bandwidth (10000×) and computational effort. Placement optimization shows excellent scalability up to 10200 seeds across 1040 switches.**

I. Introduction

To manage networks, administrators need to continuously monitor them to detect and mitigate exceptional behavior. Existing monitoring systems, however, exhibit many limitations affecting their semantics, scalability, and responsiveness (cf. *resource efficiency and full accuracy dilemma* [1]).

a) Limitations of existing systems: Based on early constrained switch designs, monitoring approaches are traditionally *collection-centric*: collecting all information possible (raw samples, simple statistics) through simple agents executing on switches, forwarding it all to a logically centralized collector that computes a global picture of the network state by filtering and analyzing data sent by all agents (e.g., sFlow [2], IPFIX [3]). More recent approaches exploit the increasing programmability of network devices [4] to obtain yet more information, in particular from packet payloads. However, sending raw packets and statistics from hundreds or thousands of switches to a collector can quickly congest network links and overwhelm the collector, even if implemented in a streaming fashion, as in Marple [5], Sonata [6], or Newton [7]. While some of these approaches aim to process raw statistics locally to switches in order to alleviate the bottleneck introduced by a logically centralized collector, they can only capture simple monitoring tasks since their statefulness is confined to aggregates (e.g., minimum, maximum, average, count).

Moreover, such recent approaches lack abstractions to dynamically adapt their behaviour once a query is satisfied. Monitoring tasks can thus only pull information from switches but can not perform *local (re)actions* [8] in response, e.g., adding a ternary content-addressable memory (TCAM) rule or P4 [9] table entries to quench DDoS attacks [10]. Triggering mitigating reactions thus requires additional out-of-band mechanisms, incurring an increased latency that can be prohibitive in many scenarios requiring fast reaction.

In addition, many monitoring tasks need to run simultaneously, such as to detect different types of anomalies, e.g., heavy hitters (HHs), DoS attacks, super-spreaders. Naïvely running several tasks independently side-by-side can lead to transmitting and processing the same data multiple times, exacerbating bottlenecks and multiplying operational costs. Existing solutions provide no opportunities for globally optimizing resource usage across the network and concurrent monitoring tasks. Lastly, many solutions are restricted to specific highly specialized HW or SW platforms [6], [7], [11]–[13] to mitigate performance hurdles induced by design limitations.

b) FARM: We present a novel monitoring and management (M&M) system called FARM (framework for network M&M) for accurate, efficient, scalable, and semantically rich network monitoring as well as management. FARM's main novelty lies in its comprehensive design. Instead of focusing solely on rethinking, innovating, or improving the efficiency of individual parts of network monitoring while relying on existing, non-tailored solutions for the rest, FARM takes a global perspective on the network monitoring problem. It *integrates and aligns hardware and software architecture, a versatile programming model, and an optimization framework correspondingly* to create a tailor-made solution that aims for

[1]Work financially supported by Hasler Foundation grant #21021 and Swiss National Science Foundation grant #192121.

https://doi.org/10.1109/ICDCS60910.2024.00055

prime performance. In short, FARM is fully *selection*-centric as opposed to collection-centric by supporting expressive, strongly decentralized, reasoning through so-called *seeds* deployable in a platform-independent ([IND]) way directly on switches. Seeds accurately poll traffic statistics, probe packets, and perform (re)actions *locally* on switches; they execute in a lightweight manner and interact among each other and with their *harvester* (a global analyzer) *only in specific, well-defined states, if at all needed*. FARM's programming model exposes just enough information for joint, dynamic seed placement optimization. Fig. 1 shows the workflow of a M&M task in FARM.

FARM's decentralized architecture ([DEC]) allows to run tasks and *perform actions where they belong*, also management actions. It exploits semantic knowledge from the programming model to efficiently use resources available on network devices. Seeds are executed on switches to get *select* information which is as timely accurate as possible and to *perform required reactions directly*. A seed can nonetheless communicate with a harvester to take global decisions if needed, but does so much more efficiently since information is fully prefiltered locally.

FARM uses a domain specific language (DSL) called automata language for network management and monitoring code (Almanac) to describe M&M tasks in an expressive model ([EXP]) by leveraging the intuitive abstraction of state machines to be executed as seeds. Almanac makes it easy to succinctly describe M&M tasks as executable entities without knowledge of network topology or resources. It is specialized to define communication patterns, resource constraints and utility, placement policies, and local (re)actions. Almanac captures a wide spectrum of use-cases where seeds can analyze switch *statistics*, *packet payloads*, but also TCAM rules. Our DSL abstractions allow FARM's runtime system to *dynamically deploy and relocate* seeds across devices without disruptions which facilitates holistic resource optimization – continuous in time and space – of seed placement for co-existing M&M tasks ([OPT]). To that end, FARM uses a novel, specialized optimization framework that considers network device resources, various overheads (e.g., seed migration), and *beneficial aggregation factors* from (re)using collected data for multiple M&M tasks deployed side-by-side.

c) Contributions and roadmap: In summary, we make the following novel contributions:

- **Decentralized M&M architecture** (§ II) that facilitates switch-local action and reaction, thus enabling network management beyond simple monitoring and leading to improved accuracy, responsiveness, and scalability.
- **Programming model for M&M tasks** (§ III) expressive enough to describe M&M tasks and, at the same time, amenable to static analysis producing dynamic resource constraint and utility for resource optimization.
- **Holistic resource optimization framework** (§ IV) for co-existing M&M tasks, featuring two algorithms: a mixed-integer linear program-based approach and our novel heuristic addressing *scalability* constraints of the former.
- **Platform-independent implementation** (§ V) of FARM on top of open-source Stratum [14] framework supporting *HW and SW solutions from most major vendors*.
- **Evaluation in a production data center** (§ VI) of SAP showing that: (1) FARM's gains significantly in responsiveness (up to 3427× faster over generic approaches and 4× faster over highly specialized solutions), precision, and savings in network bandwidth (up to 10000×) and computational requirements over the state of the art; (2) commodity switches can execute dozens of (even CPU-intensive) seeds with FARM; (3) FARM's global optimizer is scalable and efficient, capable of optimizing up to 10200 seeds across 1040 switches.

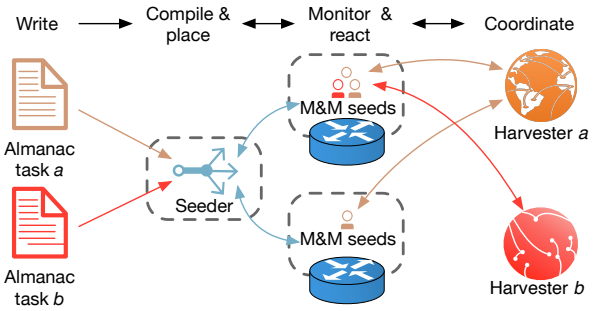


Fig. 1: FARM workflow overview. M&M tasks described in Almanac, possibly by different users, are sent to the seeder. The seeder translates these into executable seeds and deploys them on switches in a network-wide optimized manner. At runtime, seeds (re)act locally and may provide information to their harvester if/when global coordination is needed.

We pinpoint limitations of existing work in § VII, and conclude in § VIII. For ease of presentation and positioning w.r.t. existing work we use the well-known HH detection task – identification of flows beyond a threshold size [11], [15]–[17]. Our technical report [18] gives a variety of other M&M tasks.

II. M&M Architecture

We first present the complete high-level architecture of our framework for network M&M (FARM) and its components, which enable switch-local actions and reactions (cf. [DEC]).

A. Bird's Eye View

FARM builds on the idea of using switch-local support for execution of monitoring tasks (cf. Marple [5]) and extends this idea further to switch-local *management* tasks including *re*actions. Unlike pure monitoring, management decisions often cannot be made without any centralized coordination. One of the key features of the FARM design is that both monitoring and management functionalities can be decomposed into switch-local (distributed) components and centralized components. The former can take advantage of their proximity to the data source to monitor and actuate, while the latter (if needed) use a global view of system state. Communication among the two types of components follows a well-defined pattern expressed through Almanac, a novel DSL.

Most reactions involve the software-defined networking (SDN) control plane, hence, local reactions entail implementing distributed FARM components inside the *switch-local control*

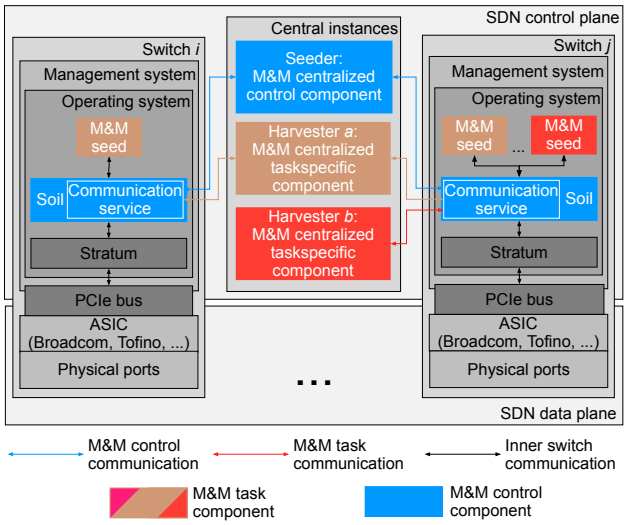


Fig. 2: FARM's architecture overview. Seeds interact via their soil with their harvester and other seeds.

plane. FARM components are designed for generality and seamless deployment across various HW platforms upon a well-defined abstraction, Stratum [14]. For superior performance, switch-local components exploit HW resources of the switch as available, providing most accurate monitoring information with lowest possible delay, crucial for a variety of time-critical measurement tasks (e.g. HH, hierarchical heavy hitter (HHH), DDoS, super-spreaders, QoS). Placement of FARM's decentralized components is then globally optimized through our purpose-built heuristic (see § IV).

FARM's design distinguishes between two types of components, as shown in Fig. 2: (1) M&M *task* components executing the logic of M&M applications (2) M&M *control* components managing deployment and execution of (1).

B. Switch-local Components

State-of-the-art data center (DC) switching devices have two main processing domains: an ASIC optimized for fast packet processing and a management system with a common CPU responsible for communication with control devices, e.g., an SDN controller, and, following either local or global management decisions, updating the forwarding rules of the ASIC (i.e., reacting). FARM components run at the management system, but continuously poll packet processing statistics, including from P4 programs, or sample packets. To optimize usage of shared switch resources, e.g. polling bandwidth, by multiple M&M tasks, we introduce two types of switch-local components – an execution unit and a "hypervisor".

a) Seed: The M&M seeds – seeds for short – of an M&M task collect monitoring data (e.g., sampled packets, statistics), filter it, and analyze it with the goal of performing local management (re)actions (e.g., change its state, update TCAM rules, deploy new P4 table entries) immediately in response without requiring remote intervention. Seeds are stateful and run as lightweight instances (processes or threads) in the switch control plane. They may interact with other seeds and their harvester (a centralized M&M task-specific coordinator, cf. Fig. 1 and § II-C) if and when needed for the M&M task. A seed definition, written in Almanac as detailed shortly in § III, forms the core of an M&M algorithm. It includes an abstract description of where the corresponding seed instance(s) will execute in the network. As the seed behavior itself may need to change as part of local reaction, we introduce explicit states to its definition, where every state may listen to events and perform actions of its own choosing. Because of the high dynamics of FARM, a seed can also change its polling rate dynamically to reduce the switch resources required, which is important to optimize M&M tasks globally (see § IV).

b) Soil: The M&M seed foundation layer (soil) manages the execution of the seeds, tracks their switch resource usage, and optimizes/aggregates communication with the ASIC over a PCIe bus, thus serving as abstraction layer between seeds and the switch. Resources tracked include three ASIC-specific types – *bus bandwidth* for probing, statistics *polling capacity*, and *TCAM space* for tracking specific flows and/or implementing various forms of local reaction. It employs its own communication service to establish and optimize its communication with remote components (i.e., centralized components or seeds or soil on other switches). The soil can aggregate polling when multiple seeds that execute different M&M tasks poll the *same data* from the switch. In such case, it is possible to poll the data only once for all seeds to minimize communication to the ASIC and avoid contention. Such opportunities are statically analyzed and leveraged for aggregation benefits (see § IV) by the M&M centralized control instance (seeder). Note that the soil carefully divides the ASICs' TCAM between monitoring and packet forwarding such that the switching behavior is not affected when rearranging the TCAM due to FARM operation. This approach draws inspiration from iSTAMP's [16] TCAM division, used for fine-grained monitoring, and extends it with an accurate polling mechanism between TCAM and seeds.

We detail the seed programming abstraction in § III, but we would like to highlight here that this abstraction is more generic than query operations used in several prior approaches (e.g., Sonata [6], Marple [5]). This allows FARM to support altering local behavior, e.g., *re*acting to some stimuli, quickly, without the need to involve a centralized entity for decision making and/or seed redeployment (e.g., Newton [7]).

C. Centralized Components

Switch-local components may still require centralized components to partake in some M&M tasks by taking centralized decisions based on data received from distributed seeds, and coordinate the placement and maximize joint utility of deployed seeds. FARM thus uses, respectively:

a) Harvester: Each M&M task can use its own specific centralized component, called a harvester, that collects (or *harvests*) and takes global management actions for it when seed-local decision-making is insufficient.

b) Seeder: The M&M centralized control instance, called a seeder, optimizes the resource utilization of all co-deployed M&M tasks. It dynamically (un)installs and (re)positions the

```
program  pg  ::= strdec* fundec* ma*
machine  ma  ::= machine mname [extends mname] md
m. def.  md  ::= {pl; xd*; st*}
var. dec. xd ::= [external] typ x [= ex] | tty y [= ex]
state    st  ::= state sname {xd*; [ut] ev*}
type     typ ::= bool | int | long | string | list | packet
                 | action | filter | ...
trig t.  tty ::= time | probe | poll | ...
utility  ut  ::= util (x) {ac*}
place    pl  ::= place (all | any) [ex* | ra]
range    ra  ::= [sender | receiver] [midpoint] [ex]
                 range op ex
filter   fil ::= dstIP ex | srcIP ex | port ex | ...
event    ev  ::= when (trg) do {ac*;}
trigger  trg ::= rec | y [as x] | enter | exit | realloc
receipt  rec ::= recv pat from (mname [@dst] | harvester)
operator op  ::= and | or | + | - | * | / | <= | >= | == | <>
expr.    ex  ::= v | y | x | fil | not ex | ex op ex | ...
action   ac  ::= y = ex; | x = ex; | transit ex;
                 | if(ex) then {ac*} [else {ac*}]
                 | while(ex){ac*} |return ex;
                 | send ex to (mname [@dst] | harvester);
```

Fig. 3: Core Almanac syntax. $\bar{z}$ denotes several instances of z, $[z]$ that z is optional. Blue highlights keywords representing state machine constructs, orange seed-specific primitives. Background marks constructs with a common purpose: placement (red), resource allocation (green), event handling (purple).

seeds following a global placement optimization algorithm (see § IV). The seeder also establishes the interface for seeds to communicate with each other either via the seeder or directly by requesting a seed's network location.

III. M&M Seed Programming Model

This section introduces our expressive ([EXP]) automata language for network management and monitoring code (Almanac)[2] and illustrates it through HH detection.

A. Language Overview

Almanac is centered around the concept of seeds, which are patterned after the well-known state machine abstraction. It draws inspiration from a variety of more generic languages and models (e.g., Esterel [19], IO-Automata [20]) based on state machines due to programmers' familiarity with that abstraction in the space of networking (cf. [21]), adding features and actions specific to M&M (e.g., packet filter expressions). Fig. 3 presents a subset of the C-like syntax for expressing state machines in Almanac. Following, $\bar{z}$ represents several instances of z, and $[z]$ means that z is optional. Blue highlighted keywords represent common state machine constructs; orange highlights seed-specific primitives.

a) Machines: A seed state machine **machine** has a name *mname*, representing a seed type rather than a single seed instance. A state machine includes a set of variable declarations xd, a set of placement constraints pl, and a set of declarations of explicit states for the machine. Variables marked as **external** are customizable at deployment, providing a way to customize seed behavior. Optionally, a machine **extends** another machine with Almanac currently implementing a simple form of single inheritance: states can be overridden in child machines, but variables cannot be overridden or shadowed. More advanced mechanisms, e.g. [22], are under investigation. Note that *trigger variables* y are special kinds of variables used for triggering events. They are assigned one of the trigger types tty – **time**, **poll**, and **probe**. Variables of type **time** or **poll** both denote events strictly periodic in time, but the latter represent polling data from the ASIC and contain filter information *fil* needed for seed placement optimization (see § IV). Type **probe** variables carry similar information to **poll** but are used to set up packet probing (sampling). The period provided for **probe** is only a lower bound and actual rate depends on the traffic. Semantics of placement constraints are explained in § III-B.

b) States: Each discrete state **state** of a machine has a name *sname* and a definition including a set of local variables xd (**external** is disallowed), a set of events ev that can affect the machine in the given state, and a callback function ut estimating the seed's utility in a given state when supplied with resource allocation bound to variable x. Note that as syntactic sugar (not shown for brevity), events can also be described at the machine level, applying to all states, with the possibility of overriding such global definitions.

c) Events: Each **event** is defined by a trigger trg for executing it and a set of actions acs performed in response. The trigger can be entering (**enter**) or exiting (**exit**) the state, the reception (**recv**) of a message from another machine (instance) or the harvester, a trigger variable reaching its triggering condition possibly assigning event data to a variable with **as** x (e.g., polled statistics, packet sample), or a resource reallocation (**realloc**) event due to placement (re)optimization (see § IV). Receptions use pattern matching on messages, constraining the source (*mname*) at a given destination (*dst*), which can be a seed, a group of seeds, other switches, or a harvester. A simple and common pattern is a formal argument; if the received message has the same type as the argument, the corresponding value will be assigned implicitly.

d) Actions: The body of an event handler includes a sequence of actions — assignments of expressions to trigger variables (e.g., to modify polling rates) or regular variables, common control structures (**if**, **while**), sending (**send**) of messages to another machine *mname* at a given host *dst* or broadcast to all hosts (no *dst*), and explicit transitions (**transit**) to states. Logic *without* state machine-related operations can be modularized into common auxiliary functions (*fundec*, omitted in the syntax for brevity), e.g. to filter TCAM rules, match regular expressions. These can **return** values.

e) Runtime library: The soil' runtime support library (see List. 1 excerpt), includes definitions for types `Probe` and `Poll` expected by respective trigger variables. The library also allows for querying resource information, e.g., `res()` returns a `Resources` structure containing amounts of allocated resources of each type. In addition, the forwarding rules in TCAM can be modified through the API using `addTCAMRule()`, `deleteTCAMRule()`, and `getTCAMRule()`. Finally, `exec()` runs

[2] An almanac is a calendar with agricultural and seasonal advice.

```
// Triggers
struct Probe { int ival; filter what; }
struct Poll { int ival; filter what; }
// Resource monitoring
struct Resources { float vCPU; int TCAM; ...}
Resources res() { ... }
// Dataplane
struct Rule { filter pattern; action act; }
void addTCAMRule(Rule rule) { ... }
void removeTCAMRule(filter pattern) { ... }
Rule getTCAMRule(filter pattern) { ... }
// Running external code
void exec(string command) { ... }
```

List. 1: Excerpt of runtime library's API.

external code (cf. ML example in § VI).

f) Utility callback function: Every state comes with a *partial* callback function `util` which takes resource amounts as argument and returns a floating point value. To keep placement efficient, we impose a range of syntactic restrictions on `util`'s body: 1. the allowed acs are `if-then-else` and `return`; 2. the allowed ops are `and`, `or`, `==`, `<=`, `>=`, `+`, `-`, `*`, and `/`; 3. function calls are forbidden except for `min` and `max`.

B. From Almanac to Tasks and Seeds

A network operator creates a task t by supplying the seeder with a set M^t of machines and, for each $m \in M^t$, the assignment of values to m's `external` variables. The seeder's *first step* for that task is to use the SDN controller to resolve `place` directives for each machine m, producing: the set of seeds S^m, and for each seed $s \in S^m$ the non-empty set of switches N^s, at exactly one of which s must be placed. The seeder's *second step* is to analyze `util` to determine each s's resource constraints $C^s(\overline{r_i})$ and utility function $u^s(\overline{r_i})$, where $\overline{r_i}$ is a sequence of variables representing allocated resource amounts. In essence, $C^s(\overline{r_i})$ reflects `util`'s domain, and $u^s(\overline{r_i})$ `util`'s return value; both are represented as explicit polynomials making them suitable for placement optimization (see § IV). Finally, to infer aggregation opportunities, as the *third step*, for every seed, the seeder derives the set of `poll` variables Y^s, and for each $y \in Y^s$: the polling subject y`.what`, and interval function y`.ival`$(\overline{r_i})$, which can depend on allocated resources. Formal definitions and further details are given in [18].

a) Placement constraints: To find S^m and then N^s for each $s \in S^m$, the seeder considers a sequence of pl directives $\Pi_1, \ldots, \Pi_k$ from m's Almanac description with each ex inside Π_i fully evaluated to constants. So $\Pi_i =$ `place` q_i pc_i, where q_i is either `all` or `any` quantifier, and pc_i is an optional placement constraint. Then, S^m is simply a union of seed sets $\pi[\![q_i\ pc_i]\!]$ corresponding to individual Π_i, where $\pi[\![\cdot]\!]$ is the placement interpretation function. $\pi[\![q_i\ pc_i]\!]$ has three main cases: (a) *pc is empty:* q_i refers to all switches (e.g., $\pi[\![$`all`$]\!] = N$ and $\pi[\![$`all`$]\!] = \{\{n\} : n \in N\}$; (b) *$pc$ is $\overline{ex}$, ex_is evaluate to switch ids:* q_i is restricted to those specific switches, otherwise similar to (a); (c) *pc is ra:* q_i refers to paths encoded by $[ex]$ (all paths if omitted), the remaining part of pc specifies a set of nodes on each path. The last case relies on two auxiliary helper functions: $\varphi^s[\![\cdot]\!]$ and $\varphi_{\text{path}}(\cdot)$. For a `bool` expression ex, $\varphi^s[\![ex]\!]$ evaluates all variables referenced in ex based on seed s's state and returns a closed boolean formula with *fil* as atomic propositions. Then, for such a closed formula ex_c, $\varphi_{\text{path}}(ex_c)$ represents the result of seeder retrieving a set of paths matching ex_c from the SDN controller. As an example, consider $ex_c =$ `srcIP "10.1.1.4" and dstIP "10.0.1.0/24"`. If

$$\varphi_{\text{path}}(\texttt{srcIP "10.1.1.4" and dstIP "10.0.1.0/24"}) = \{(1,2,5,3,4),(1,2,6,3,4),(1,2,7,8,9)\}$$

then

$$\pi[\![\texttt{any receiver } ex \texttt{ range == 1}]\!] = \{\{3,8\}\}$$
$$\pi[\![\texttt{all midpoint } ex \texttt{ range == 0}]\!] = \{\{5\},\{6\},\{7\}\}$$
$$\pi[\![\texttt{any receiver } ex \texttt{ range <= 1}]\!] = \{\{3,4\},\{3,4\},\{8,9\}\}$$

b) Resource constraints and utility: For understanding `util` block, the simplest case is a single `if` statement whose condition does not use `or`. First, the condition is converted by constraint interpretation function $\kappa^s[\![\cdot]\!]$ to a set C^s of polynomials over $\overline{r_i}$, each of which must be non-negative for the resource constraints to hold. For example,

$$\kappa^s[\![\texttt{res.vCPU >= 1 and res.RAM >= 100}]\!] = \{r_1 - 1, r_2 - 100\}$$

Second, the `return` expression under `if` is converted by an expression interpretation function $\epsilon^s[\![\cdot]\!]$ to a polynomial $u^s(\overline{r_i})$. Supporting `or` operators or several `if` is also possible, but uses instead of a set C^s and a function u^s sets $\{C^s_i\}_{i=1}^k$ and $\{u^s_i\}_{i=1}^k$ of those, meaning that utility is $u^s_i(\overline{r_i})$ once $C^s_i(\overline{r_i}) \geq 0$. For placement optimization (see § IV) that would amount to splitting the seed into several copies, at most one is to be placed.

c) Polling aggregation: For polling aggregation the seeder must know to what extent the two filters x_1 `.what` and x_2 `.what` for `poll` variables x_1 and x_2 share polling subjects, e.g., the same TCAM entries or the same logical interfaces. To determine that, we use $\varphi^s[\![\cdot]\!]$ again to evaluate the expression into a boolean formula over *fil* with constants. A single filter may require polling multiple statistics from the ASIC (e.g., from several TCAM rules) so the precise sharing depends on the actual filter encoding. We assume a filter encoding function φ_{enc} that returns a set of polling subjects given an output of $\varphi^s[\![\cdot]\!]$. Finally, since the seed programmer may choose y`.ival` to depend on actual resource amount (encouraging allocation with `util`), we capture that dependency as well. For a seed s with poll variables Y^s and for each $y \in Y^s$ defined as `poll` y`=Poll{.ival=`ex_1`,.what=`ex_2`}`, we derive y`.ival`$(\overline{r_i}) = \epsilon^s[\![ex_1]\!]$ and y`.what` $= \varphi_{\text{enc}}(\varphi^s[\![ex_2]\!])$.

TABLE I: 16 well-known network monitoring and attack examples implemented in FARM with numbers of lines of code. The numbers include all code, e.g., also abstracted functions.

Use case	Seed	Harv.	Use case	Seed	Harv.
Heavy hitter (HH)	29	12	Link failure [23]	31	8
Hier. HH [24]	21	26	Traffic change [25]	7	5
(inherited)			Flow size distr. [26]	30	15
Hier. HH [24]	38	26	Superspreader [13]	58	21
DDoS [10]	71	30	SSH brute force [27]	34	9
New TCP conn. [28]	19	5	Port scan [29]	44	23
TCP SYN flood [28]	63	18	DNS reflection [30]	83	22
Partial TCP flow [28]	73	18	Entropy estim. [31]	67	15
Slowloris [32]	44	29	FloodDefender [33]	126	35

```
machine HH {
  place all;  S^m = {s_1, ..., s_|N|};  ∀i. N^{s_i} = {n_i}
  poll pollStats = Poll {
    .ival = 10/res().PCIe, .what = port ANY
  };  y.ival(r_1, r_2, r_3) = 10/r_3;  y.what = {eth0, eth1, ...}
  long threshold;
  action hitterAction;
  list hitters;
  state observe {
    util (res) {
      if (res.vCPU>=1 and res.RAM>=100) then {
        return min(res.vCPU, res.PCIe);
      }  u^s(r_1, r_2, r_3) = min(r_1, r_3);
    }  C^s(r_1, r_2, r_3) = {r_1 − 1, r_2 − 100}
    when (pollStats as stats) do {
      hitters = getHH(stats, threshold);
      if (not is_list_empty(hitters)) then {
        transit HHdetected;
      }
    }
  }
  state HHdetected {
    util (res) { return 100; }
    when (enter) do {
      send hitters to harvester;
      setHitterRules(hitters, hitterAction);
      transit observe;
    }
  }  u^s(r_1, r_2, r_3) = 100;  C^s(r_1, r_2, r_3) = ∅
  when (recv long newTh from harvester)
  do { threshold = newTh; }
  when (recv action hitAct from harvester)
  do { hitterAction = hitAct; }
}
```

List. 2: Heavy hitter (HH) seed example.

C. Illustration

Tab. I shows scenarios implemented with Almanac. We detail heavy hitter (HH) detection (see List. 2) for illustration. HHs are flows with sizes larger than a given threshold. The example has two states: In the (1) `observe` state, none of the observed ports are identified as an HH; as soon as a port's transmitted bytes reach the defined `threshold`, a state transition to (2) `HHdetected` occurs. In state (2) the current port list is sent to the HH **`harvester`**, enabling reaction to the HHs in the network. In addition, local reaction is triggered to install TCAM rules through auxiliary functions (abstracted in *`setHitterRules`*) for the detected flows altering QoS policy for respective packets. The two events for receiving messages are defined outside of (1) and (2), applying to both. The harvester sets up the threshold for a HH and can dynamically change it based on the overall network load. If the network policy changes, the harvester can also modify the action that seeds apply locally to detected HHs. Auxiliary function *`getHH`* uses common programming constructs to determine which flows are HHs and is abstracted (thus *italicized*) for brevity. See [18] for seed code for all Tab. I examples including hierarchical HH (HHH) detection.

IV. M&M Seed Placement

This section formulates FARM's seed placement problem and discusses our algorithmic solution, which fulfills our network monitoring requirements of optimization ([OPT]).

A. Rationale and Overview

As introduced in § II and § III, FARM enables the description of switch-local tasks and their deployment in a distributed manner as seeds on switches. Seeds $S^t = \bigcup_{m \in M^t} S^m$ of a task t may be positioned on different switches, seed s on exactly one of N^s. A seed may also be migrated; either due to placement constraint for the seed not being satisfied anymore or due to a new seed with a higher utility needing the same switch. Migration induces extra resource usage due to a seed's own inner state being synchronized between different nodes. The polling periods of a seed can depend on the actual resource allocation. Also, certain seeds can benefit from aggregation as they consider the same data. Considering the best performance for tasks and the many parameters and options available, in § IV-D we propose a heuristic algorithm to optimize seed placement in FARM. Tab. II summarizes notation. Lowercase letters represent specific elements, while the corresponding uppercase letter denotes the entire set of elements of that kind. E.g., n denotes a switch, N denotes the set of all switches.

TABLE II: Elements and notation of optimization model.

Description	Element	Set	Description	Element	Set
`poll` variables	y	Y	Seeds	s	S
Poll subjects	p	P	Resource types	r	R
M&M tasks	t	T	Switches	n	N

TABLE III: Functions and variables of the optimization model.

Optimization *input* description	Notation	Source
Set of seeds that belong to t	S^t	**`place`**
Set of switches where s can be placed	N^s	**`place`**
Polling interval function for variable y	y`.ival`	**`poll`**
Polling subject for variable y	y`.what`	**`poll`**
Utility function for seed s	u^s	**`util`**
Set of resource constraints for seed s	C^s	**`util`**
Available resources of type r on n	ares(n, r)	soil
Optimization *variable* description		**Function**
Returns 1 if all t's seeds are placed, else 0		tplc(t)
Returns 1 if s is placed on $n \in N^s$, else 0		plc(s, n)
Returns 1 if s is being migrated		migr(s, n)
Amount of type r res. assigned to s at $n \in N^s$		res(s, n, r)
Amount of type r_{poll} res. assigned to p at n		pollres(n, p)

B. Monitoring Utility

The optimization model for seed placement is captured below. The goal of optimization is to *maximize* the *monitoring utility (MU)*, which reflects the monitoring quality for the entire system based on the resources assigned to each seed. The monitoring utility is defined as the sum over all seeds $s \in S$ and all switches $n \in N$, values returned by s's **`util`** callback with every resource $r \in R$ set to the assigned resources $\mathsf{res}(s, n, r)$ (Tab. III lists helper functions and variables):

$$\text{maximize} \quad \sum_{s \in S} \sum_{n \in N^s} \mathsf{plc}(s, n) \cdot u^s(\overline{\mathsf{res}(s, n, r_i)}) \quad \text{(MU)}$$

a) Migration overhead: While seed migration can generally optimize the seed layout, it incurs costs that must be considered. Migrating a seed consists of installing its description on the target switch and transferring its state over from the source switch. As the state is being transferred, and before it is deleted on the source switch, the seed resource utilization is temporarily doubled. Below, migr(s, n) checks

for all $s \in S$ and $n \in S$ if seed s is migrating from n. We use plc'(s, n) to denote the *known value* of plc(s, n) from the previous placement run, i.e., current placement:

$$\mathsf{migr}(s,n) = \mathsf{plc}'(s,n) \cdot \textstyle\sum_{n' \in N^s \setminus \{n\}} \mathsf{plc}(s,n')$$

b) Aggregation benefits: Aggregating seeds at the same switch can reduce data polling cost. Let $r_{\text{poll}} \in R$ be the resource type reflecting polling capacity. Our key assumption is that the demand for r_{poll} arising from trigger variable y is inversely proportional to the polling interval $y.\texttt{ival}(\overline{\mathsf{res}(s,n,r_i)})$ with coefficient $\alpha_{\text{poll}}(n)$, which may depend on switch n's architecture. As the demand is shared among s's poll variables having the same polling subject, we introduce a single variable $\mathsf{pollres}(n,p)$ for the amount r_{poll} consumed by a given polling subject p, s.t., $y.\texttt{what} = p$, at n:

$$\mathsf{pollres}(n,p) \geq \alpha_{\text{poll}}(n) \cdot \mathsf{plc}(s,n) / y.\texttt{ival}(\overline{\mathsf{res}(s,n,r_i)}) + \alpha_{\text{poll}}(n) \cdot \mathsf{migr}(s,n) \; / y.\texttt{ival}(\overline{\mathit{res}'(s,n,r_i)})$$

C. Constraints

Several constraints have to be taken into account when optimizing monitoring seed placement, we present them below as (C1)–(C4).

a) Every seed is placed at one switch at most.: For any task $t \in T$, if one of t's seeds is placed, then every seed $s \in S^t$ must be placed at exactly one switch $n \in N^s$:

$$\textstyle\sum_{n \in N^s} \mathsf{plc}(s,n) = \mathsf{tplc}(t) \qquad \text{(C1)}$$

b) Resources assigned for placed seeds: The amount of type $r \in R$ resources assigned to $s \in S$ on $n \in N^s$, $\mathsf{res}(s,n,r)$, shall satisfy s's `util` callback function's domain:

$$\mathsf{plc}(s,n) \cdot c(\overline{\mathsf{res}(s,n,r_i)}) \geq 0 \qquad \text{(C2)}$$

c) Only placed seeds consume resources: A seed $s \in S$ on a switch $n \in N^s$ can get at most the total amount of type $r \in R$ resources available at n, but *only if* s *is placed on* n:

$$\mathsf{res}(s,n,r) \leq \mathsf{ares}(n,r) \cdot \mathsf{plc}(s,n) \qquad \text{(C3)}$$

d) Switch resource limit for all seeds: The total of type r resources assigned to seeds at n cannot exceed the available amount at switch n with a special account for aggregated r_{poll}:

$$\begin{aligned} &\textstyle\sum_{s,\ N^s \ni n} \mathsf{res}(s,n,r) + \mathsf{migr}(s,n) \cdot \mathsf{res}'(s,n,r) \\ &\quad \leq \mathsf{ares}(n,r) \;\forall\; n \in N,\ r \in R \setminus \{r_{\text{poll}}\} \qquad \text{(C4)} \\ &\textstyle\sum_{p \in P} \mathsf{pollres}(n,p) \leq \mathsf{ares}(n, r_{\text{poll}}) \;\forall\; n \in N \end{aligned}$$

D. Placement Optimization Algorithm

If u^s and C^s, and inverse of $y.\texttt{ival}$ are linear, then we almost have the mixed-integer linear program (MILP) formulation. Unfortunately, in equation (MU), inequality for $\mathsf{pollres}(n,p)$, and in (C2), there is an occurrence of $\mathsf{plc}(s,n) \cdot f(\overline{\mathsf{res}(s,n,r_i)})$ term where f is linear, but the expression as a whole violates linearity. Fortunately, thanks to (C3) we know that $\mathsf{plc}(s,n) = 0$, implies $\overline{\mathsf{res}(s,n,r)} = 0$ for any r, hence we can rewrite the term as $f(\overline{\mathsf{res}(s,n,r_i)}) - (1 - \mathsf{plc}(s,n)) \cdot f(\overline{0})$.

Optimal assignment of seeds to switches is an NP-hard problem [18]. To maintain scalability in larger deployments, we propose a heuristic in Alg. 1: it *places* seeds greedily with minimum utility and no migration, *redistributes resources* with linear programming, and only then *migrates* placed seeds.

Algorithm 1 FARM seed placement optimization heuristic.

1) Sort tasks T by decreasing minimum utility as $t_1, t_2, \ldots, t_k$.
2) For every $t = t_1, t_2, \ldots, t_k$
   a) Repeat while possible: among $s \in S^t$ choose and *place* such s that adds the most to the utility *without unnecessary migration* (for existing seeds with valid placement).
   b) If there remains $s \notin S^t$, remove S^t from the placement.
3) Redistribute resources using linear programming formulation.
4) For every seed s and every switch $n \in N^s$ compute the *migration benefit* as the increase in utility when s is migrated to n (expressible as a linear program).
5) Migrate the seeds in the order of decreasing *migration benefit*.

V. Implementation

We elaborate on the platform-independent ([IND]) implementation of FARM touching on Almanac and the seed placement.

A. FARM Components

a) Switch integration: FARM implements two drivers for the communication between the CPU and the ASIC via the PCIe bus, one for Stratum [14] and one for Arista's EOSSDK [34]. Stratum is an OS module that abstracts the HW layer of major ASICs to provide common interfaces. FARM is thus deployable on all ASICs supported by Stratum and Arista EOS switches. We ensured that communication over the PCIe bus between the (i) CPU running the soil and seeds and (ii) ASIC can be scheduled to fully exploit the bus' capabilities.

b) Switch-local components: seeds and soil are optimized to execute directly on swiches. Seeds can run as isolated processes or as threads of the soil process. They communicate over a generic interface supporting 1. gRPC [35] and 2. a tailor-fitted shared memory buffer usable when seeds are implemented as threads of the soil. gRPC's poor performance motivated the development of 2. We evaluate both seed execution models and communication schemes in § VI-E.

c) Centralized components: FARM's centralized components, the seeder and harvesters, are implemented in Python and contain a communication service to interact with the soils' communication service to exchange data with both soils and the seeds they support. Communication between seeder / harvesters and soils is performed via RabbitMQ [36].

d) Almanac: Seeds' state machines are described in Almanac, compiled by the seeder into XML, and transformed from XML to one or more seeds by each switch's soil. XML is used for interoperability and portability across OSs. The *ttype*s in Almanac (cf. § III) are used by the soil to optimize communication with the ASIC over all running seeds.

B. Placement

The M&M placement function optimizes network resource utility using defined heuristics (§ IV), considering the switches, their topology, local resource consumption, current seed placement, and seed resource consumption. The function outputs the new seed placement and allocated resources, i.e., period for probing packets and polling statistics. We implemented the function in Rust [37] using a MILP library [38] and compare its performance to Gurobi [39] in § VI-D.

The seeder calls the placement optimization algorithm every time an input parameter of the M&M placement function changes, e.g., when a switch's soil notifies the seeder that its resources are depleting. The seeder takes the actions necessary to realize the optimizer's output, e.g., by migrating seeds. When migrating a seed, the seeder first deploys the seed's description to its new location, then transfers its state there; seed execution resumes once the state is migrated.

VI. Evaluation

We evaluate FARM in a production DC of SAP w.r.t. four research questions:

§ VI-B How does FARM compare to state-of-the-art solutions w.r.t. responsiveness, network load, and switch CPU load?
§ VI-C How does FARM's monitoring accuracy (which affects responsiveness) scale with a large number of – possibly CPU-intensive – seeds executed concurrently?
§ VI-D How does FARM's placement algorithm scale in terms of monitoring utility and runtime?
§ VI-E How efficient is FARM's implementation?

A. Setup

a) Platforms: We used (i) APS BF2556X-1T, (ii) Accton AS5712, (iii) Accton AS7712, and (iv) Arista 7280QRA-C36S switches. (i) run Open Network Linux (ONL) with a 2.0 Tbps Intel Tofino ASIC and an Intel Xeon 8-core 2.6 GHz x86 processor with 32 GB SO-DIMM DDR4 RAM with ECC. (ii) run ONL and have an Intel Atom C2538 quad-core 2.4 GHz x86 processor with 8 GB SO-DIMM DDR3 RAM with ECC. (iii) are like (ii) with twice the RAM. (iv) run EOS and have an AMD GX-424CC SOC quad-core 2.4 GHz with 8 GB DRAM.

b) Topology: We deployed FARM on a cluster with a spine-leaf topology in a production DC of SAP. As FARM is undergoing a long-term evaluation period before being globally rolled out we report performance results on 20 switches.

c) Scenarios: We first investigate the HH detection task of § III-C deploying one seed per port on all switches. Additionally, we assess FARM with a CPU-intensive task using machine learning (ML) to directly react to events on switches. Leveraging ML for prediction is a budding field in networks [40]–[42] and adding support for prediction thus a need often stated for monitoring [43]. The ML task relies on support vector regression [44] using matrix-matrix multiplications with 1000×1000 matrices. The Python implementation is executed via `exec()`, parameterized by the polled statistics.

B. Scalability

Identifying HHs serves various purposes (e.g., DoS detection, traffic engineering). While it doesn't show FARM's full potential, its widespread use in literature allows the best comparison (responsiveness, network and CPU load) against existing systems, including HH detection specific ones. We evaluate in detail against sFlow [2] and Sonata [6] (and Newton [7], which extends Sonata with dynamic loading), two representatives of generic collector- and stream-based approaches. Other recent approaches show promising results, but have the same conceptual limitations as sFlow or Sonata, and are not publicly available.

a) Responsiveness: Tab. 4 compares the time needed to recognize an HH with FARM also to the more specialized Planck [11] (using specialized HW) and Helios [17] systems. FARM shows great speedups while being at least as generic (sFlow) or more (Planck, Helios, Sonata). Transitively, FARM also greatly reduces mitigation time. Note that Sonata only computes a switch-local HH (cf. § VII). FARM achieves such speedups by analyzing traffic directly on switches while the other solutions send simpler statistics and data to centralized instances. Another advantage of FARM is the ability to *react* on a switch when recognizing an HH, e.g., to install a rate limit for HHs, an `action` can be described with Almanac. Both HH recognition *and* mitigation happen within 1 ms.

b) Network load: Fig. 4 depicts network load for HH detection, showcasing FARM's benefits over sFlow and Sonata. We chose HH parameters based on observations in our production DC – HHs usually affect 1% of network ports, 10% at worst, and the HH ratio changes up to once a minute. sFlow periodically sends packets to probe every port in the network. We thus run it with a 1 ms probing period to achieve a similar detection time as FARM, and with a 10 ms period to reduce load since it increases linearly with network size with collector-based solutions. Assuming Sonata could aggregate over several switches to compute HHs, the raw statistics issued by the switches to the Spark system deployed by Sonata would still create further network load. We run Sonata assuming an aggregation factor of 75%, the best achievable with an HH ratio changing up to once a minute. Further decentralizing aggregation with more levels would only add yet more traffic. In comparison, FARM's bandwidth consumption increases by only 1 packet per minute for every 100 additional ports. As shown in Fig. 4, FARM exhibits a lower linear gradient than sFlow, with reduced total volume and slope. It consumes less bandwidth and is over 1000× more computationally efficient than the centralized collector. Importantly, FARM promptly detects changes in the HH ratio, which is especially valuable when changes occur frequently.

c) CPU load: Fig. 5 depicts FARM's and sFlow's CPU loads as they poll statistics from multiple flow rules with equal monitoring accuracy. We don't compare against Sonata as it mirrors the traffic and thus its bottleneck is the sampling rate of the PCIe bus (cf. § VI-E). Its number of individual instances is not meaningful due to the lack of samples. sFlow's CPU load is always higher than FARM's except with 100 flows. sFlow's CPU load is stable since it is a (locally) lightweight approach that samples packets and forwards them to its collector without filtering. On the other hand, FARM analyzes the data and manages its own state, thus CPU load increases with the number of monitored ports. Yet, as long as not all ports are affected, the SDN control plane is not congested with FARM unlike with sFlow (cf. Fig. 4).

Tab. 4: HH detect times of generic (G) systems FARM, sFlow, Sonata, and specialized (S) link utilization monitoring systems Planck (10 Gbps) and Helios.

System	Type	Time
FARM	G	1 ms
Planck	S	4 ms
Helios	S	77 ms
sFlow	G	100 ms
Sonata	G	3427 ms

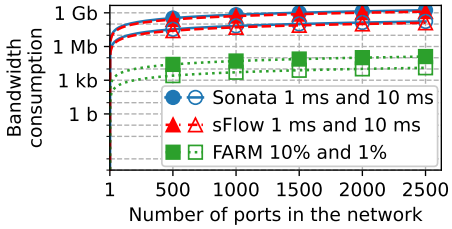


Fig. 4: Network load of FARM with 1 and 10% HH ratios, the sFlow collector with 1 and 10 ms accuracies, and similarly Sonata/Newton.

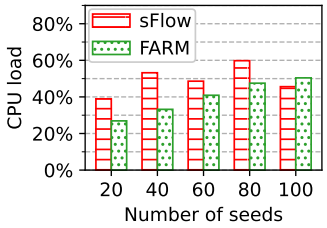
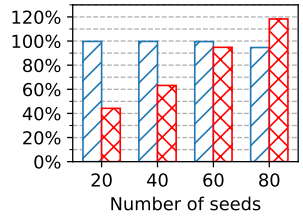
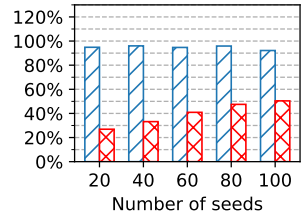


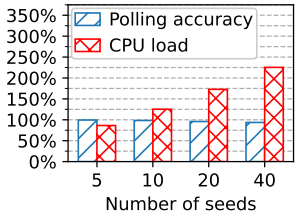
Fig. 5: Switch CPU load of FARM and sFlow for HH detection, 10 ms accuracy.



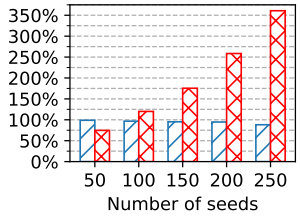
(a) HH: 1 ms accuracy.



(b) HH: 10 ms accuracy.

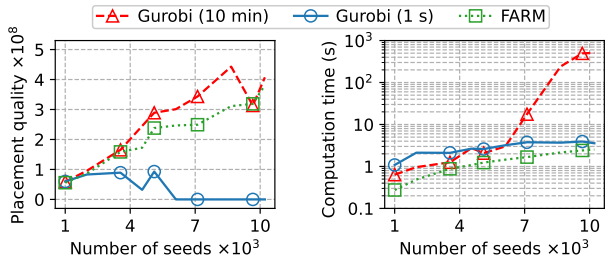


(c) ML: 1 ms accuracy, 1 iteration.



(d) ML: 10 ms accuracy, 10 iterat.

Fig. 6: CPU load of FARM for HH and ML tasks.



(a) Monitoring utility.

(b) Optimization runtime.

Fig. 7: FARM's global seed placement optimization algorithm (cf. Eq. MU) is close in utility to Gurobi with 10 min timeout and as fast as Gurobi with 1 s timeout.

C. Accuracy vs CPU Load

We evaluate the effect of running many collocated seeds on the same switch, specifically from the angle of monitoring accuracy (i.e., polling period) and its impact on CPU load. Fig. 6a and Fig. 6b show CPU load with various numbers of seeds with every seed polling statistics from multiple flow rules every 1 and 10 ms respectively for the HH task. It incurs only light CPU load and easily scales to more than a hundred of seeds per switch with a 10 ms accuracy.

Due to their complexity we run ML seeds with a 1 ms accuracy in parallel (cf. Fig. 6c), and a 10 ms accuracy with statistics polling once but executing 10 iterations of the algorithm (cf. Fig. 6d) reducing the parallel seed count by a factor of 10. Fig. 6c shows the CPU load is $\approx 150\%$ higher for the ML task with 1 ms accuracy than for the HH task, leading the CPU being unable to handle all seeds in parallel due to the many context switches. By dividing the seed into partitions (cf. Fig. 6d), the CPU load decreases and the system scales well up to 250 seeds of this ML task.

D. Global Seed Placement Optimization

To evaluate FARM's global placement optimization algorithm, we compare it against a commodity MILP solver using Gurobi [39] (used by Sonata). Two timeouts are used for Gurobi: 1 s to get runtime similar to FARM (at the expense of utility), and 10 min as an absolute practical upper bound (to get similar utility). Testing involves up to 10 different tasks (cf. Tab. I) comprising up to 10200 seeds and deploying them on 1040 switches. For each seed count, we conduct 10 runs with varying resource and placement needs for M&M tasks. Fig. 7a shows the average monitoring utility (cf. Eq. MU) and Fig. 7b the average time needed to find a corresponding solution. FARM's placement optimization algorithm achieves similar utility as Gurobi but much faster, which is crucial with a large number of tasks and seeds to deploy.

E. Implementation Microbenchmarks

We show via a series of microbenchmarks the need for the optimizations implemented in FARM (cf. § V-A), e.g., that FARM performs best with seeds executing as threads within the soil process and using a shared buffer for soil-seeds communication. We used this implementation for the rest of FARM's evaluation. We deploy the ML task to benchmark switch HW utilization and identify HW bottlenecks. We use Accton AS5712 and AS7712 switches for the benchmarks and plot the averaged (similar) results.

a) PCIe bus capacity: Fig. 8 shows that the main bottleneck M&M tasks is the PCIe bus, quickly congesting as seeds poll the ASIC's TCAM. The PCIe bus capacity for polling traffic statistics is limited to 8 Mbps on both tested switches while their ASICs support 100 Gbps (i.e., a 1:12500 ratio). To circumvent the PCIe bus bottleneck, FARM enables, in addition to data sample polling, the soil to aggregate the seeds' requests before sending them over the PCIe bus.

b) Aggregation cost: Aggregating seeds' requests requires the soil, thus trading PCIe bandwidth for CPU load. The latter is only noticeable when seeds run as processes, while thread-based seeds in the soil perform equally well regardless of aggregation, even with more than 100 seeds (Fig. 9).

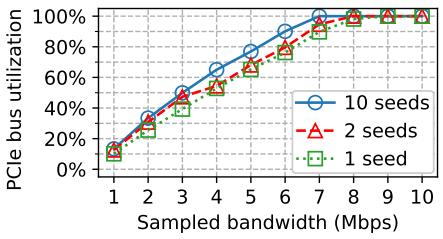


Fig. 8: The PCIe bus easily congests compared to the ASIC bus, calling for polling aggregation.

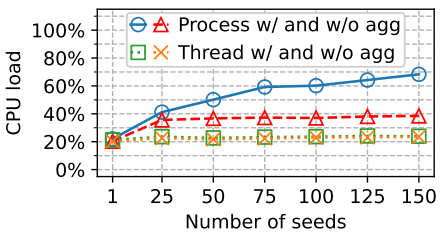


Fig. 9: Soil's CPU load showing the cost of seed requests' aggregation when seeds are threads vs processes.

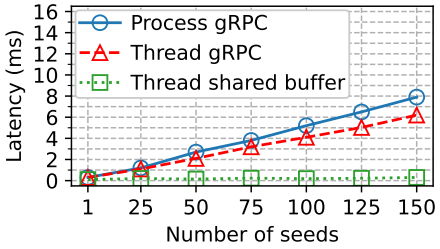


Fig. 10: Shared buffer vs gRPC communication latency between seeds being threads or processes and soil.

c) Latency overhead: Fig. 10 shows that gRPC scales linearly with deployed seed count, becoming the latency bottleneck. As a fix we implemented a lightweight soil-seed communication scheme based on a shared buffer where seeds run as threads within the soil. Fig. 10 shows a marginal latency overhead of the shared buffer scheme even with 150 seeds.

VII. Related Work

This section discusses related work on *generic* monitoring systems summarizing their shortcomings with respect to features and requirements introduced in § I. From the many *specialized* solutions introduced for specific monitoring scenarios (e.g., HH detection [11], [15]–[17], DoS detection [33]), we refer to a few later for comparison or influence on FARM's design. Note, dynamic deployment (migration), which we view as a prerequisite to global optimization, has not been attempted in conjunction with optimizing across concurrent monitoring tasks, as done by FARM, by any prior work.

sFLow [2] is a network traffic standard encompassing: *sFlow agents* implementing traffic sampling mechanisms; and a centralized ([DEC]) *sFlow collector* analyzing samples or statistics. sFlow uses minimal switch-local processing or triage, performing all analysis on (2). This hampers latency as all statistical data has to be transferred there, and limits scalability. sFlow is not an IETF standard (cf. RFC 3176), but a golden standard widely deployed on many switch types of many vendors; thus we include it in our evaluation (§ VI).

Sonata [6] emphasizes "stream processing-like" network telemetry [45]. It's partially implemented in the data plane via P4 [9], offloading complex query parts to Spark Streaming [46]. The state of monitoring tasks is, however, limited to that of simple aggregation operations hampering ([EXP]). Moreover, Sonata does not support merging of streams from several switches, and hence can not be used in many standard scenarios like global HH detection.[3] Sonata optimizes switch data plane resources for queries via a MILP using the Gurobi solver out of the box [39], limiting scalability.

Newton [7] inherits Sonata's P4-based streaming approach, but allows dynamic deployment of monitoring tasks and query updates without switch rebooting. It can also merge streams from several switches, yet despite ideas to reduce streaming overhead, processing remains logically centralized ([DEC]), leading to scalability and responsiveness akin to Sonata's.

OmniMon [1] tackles the collector bottleneck by separating tasks on end hosts and switches directly. Nevertheless, a centralized controller has to synchronize all hosts and switches, and share a global state. It lacks resource utilization optimization ([OPT]) across monitoring tasks and a generic abstraction; all evaluated tasks are individually designed.

Beaucoup [48] abstracts hardware design similarly to FARM. The authors implement a memory-efficient, performant *coupon* system upon the TCAM over queries similar to Sonata and Newton. It focuses on monitoring tasks solvable with a probabilistic distinct-counting limiting ([EXP]).

Marple [5] pioneered stream-based monitoring but, unlike other systems, supports data aggregation directly using local state on programmable switches if available ([IND]). However, this support comes with a very limited set of aggregation primitives ([EXP]), which suffice for basic statistics but not for advanced scenarios like HHs. In addition, Marple relies on a specific key-value store design implemented in switch HW.

Recent literature explores monitoring in the data plane using *sketches* [49]–[51], offering accuracy guarantees but relying on a programmable data plane with strict resource limitations. Sketches are complementary to FARM's holistic perspective.

VIII. Conclusions

We introduced FARM, a novel comprehensive solution for large-scale DC network monitoring and management (M&M). FARM relies on the Almanac programming framework to describe autonomous seeds that execute M&M tasks directly on switches; it uses a custom-designed scalable heuristic to optimize their placement, considering heterogeneous HW resources, aggregation benefits, and migration overhead.

We evaluated the accuracy and scalability of FARM against existing generic systems (e.g., sFlow, Sonata) and specialized link utilization/HH detectors (e.g., Planck) showing reduced bandwidth requirements of centralized instances compared to collector-based approaches, and benefits of reacting directly to anomalies on the switch with predefined actions.

Avenues for future work include fault tolerance, extensions to Almanac (e.g., advanced inheritance, combined implementation of seeds and harvester à-la tierless programming [52]), and the integration of sketches into FARM.

[3] Several of the authors propose a separate system for HH [47] where they propose to adapt their work in the future "*to detect network-wide heavy hitters [...] for inclusion in [...] a general network telemetry system.*"

TABLE V: Features of *generic* M&M solutions.

System	Local action	Local reaction	Statistics poll.	Payload poll.	Dyn. deploy.	Poll. aggreg.	HW	SW
	[DEC]		[EXP]		[OPT]		[IND]	
sFlow [2]	○	○	○	○	●	○	●	○
Sonata [6]	●	○	●	●	●	○	○	○
Newton [7]	●	○	●	●	●	○	○	○
OmniMon [1]	●	◑	●	●	●	○	○	○
BeauCoup [48]	●	○	●	○	◑	○	●	○
Marple [5]	●	○	●	○	◑	○	○	○
FARM (this work)	●	●	●	●	●	●	●	●

REFERENCES

[1] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "OmniMon: Re-Architecting Network Telemetry with Resource Efficiency and Full Accuracy," in *SIGCOMM'20*, H. Schulzrinne and V. Misra, Eds., 2020, pp. 404–421.

[2] P. Phaal, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in a Switched and Routed Networks," RFC 3176, Sep. 2001.

[3] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," RFC 5101, Feb. 2008.

[4] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications," *CSUR*, vol. 54, no. 4, 2021.

[5] S. Narayana *et al.*, "Language-Directed Hardware Design for Network Performance Monitoring," in *SIGCOMM'17*, 2017, pp. 85–98.

[6] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven Streaming Network Telemetry," in *SIGCOMM'18*, S. Gorinsky and J. Tapolcai, Eds., 2018, pp. 357–371.

[7] Y. Zhou *et al.*, "Newton: Intent-Driven Network Traffic Monitoring," in *CoNEXT'20*, D. Han and A. Feldmann, Eds., 2020, pp. 295–308.

[8] L. Jose, M. Yu, and J. Rexford, "Online Measurement of Large Traffic Aggregates on Commodity Switches," in *Hot-ICE'11*, A. Shaikh and K. van der Merwe, Eds., 2011.

[9] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *JTIT*, vol. 44, no. 3, pp. 87–95, 2014.

[10] J. Mirkovic and P. Reiher, "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms," *SIGCOMM CCR*, vol. 34, no. 2, pp. 39–53, 2004.

[11] J. Rasley *et al.*, "Planck: Millisecond-Scale Monitoring and Control for Commodity Networks," in *SIGCOMM'14*, F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, Eds., 2014, pp. 407–418.

[12] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic Resource Allocation for Software-Defined Measurement," in *SIGCOMM'14*, F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, Eds., 2014, pp. 419–430.

[13] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *NSDI'13*, N. Feamster and J. C. Mogul, Eds., 2013, pp. 29–42.

[14] Open Networking Foundation, *Stratum*, 2023. [Online]. Available: www.opennetworking.org/stratum/.

[15] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *SOSR'17*, 2017, pp. 164–176.

[16] M. Malboubi, L. Wang, C.-N. Chuah, and P. Sharma, "Intelligent SDN Based Traffic (de)Aggregation and Measurement Paradigm (iSTAMP)," in *INFOCOM'14*, 2014, pp. 934–942.

[17] N. Farrington *et al.*, "Helios: a Hybrid Electrical/Optical Switch Architecture for Modular Data Centers," in *SIGCOMM'11*, S. Kalyanaraman, V. N. Padmanabhan, K. K. Ramakrishnan, R. Shorey, and G. M. Voelker, Eds., 2011, pp. 339–350.

[18] J. Graf, P. Chuprikov, P. Eugster, and P. Jahnke, *Online Artifacts for FARM*. github.com/JeromeGraf/ICDCS24-FARM, 2024.

[19] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *SCP*, vol. 19, no. 2, pp. 87–152, 1992.

[20] S. J. Garland and N. Lynch, "Using i/o automata for developing distributed systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds., Cambridge University Press, 2000, ch. 13, pp. 285–312, ISBN: 978-0-521-77164-1.

[21] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable Dynamic Network Control," in *NSDI'15*, 2015, pp. 59–72.

[22] B. Chin and T. D. Millstein, "An Extensible State Machine Pattern for Interactive Applications," in *ECOOP'08*, J. Vitek, Ed., 2008, pp. 567–591.

[23] Y. Zhu *et al.*, "Packet-Level Telemetry in Large Datacenter Networks," in *SIGCOMM'15*, S. Uhlig, O. Maennel, B. Karp, and J. Padhye, Eds., 2015, pp. 479–491.

[24] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications," in *SIGCOMM'14*, A. Lombardo and J. F. Kurose, Eds., 2004, pp. 101–114.

[25] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams," in *IMC'04*, 2004, pp. 207–212.

[26] N. Duffield, C. Lund, and M. Thorup, "Estimating Flow Distributions from Sampled Flow Statistics," in *SIGCOMM'03*, A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall, Eds., 2003, pp. 325–336.

[27] M. Javed and V. Paxson, "Detecting Stealthy, Distributed SSH Brute-Forcing," in *CCS'13*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds., 2013, pp. 85–96.

[28] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative Network Monitoring with NetQRE," in *SIGCOMM'17*, 2017, pp. 99–112.

[29] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast Portscan Detection using Sequential Hypothesis Testing," in *S&P'04*, 2004, pp. 211–225.

[30] M. Kührer, T. Hupperich, C. Rossow, and T. Holz, "Exit from Hell? Reducing the Impact of Amplification DDoS Attacks," in *USENIX Security'14*, K. Fu and J. Jung, Eds., 2014, pp. 111–125.

[31] M. Mitzenmacher and S. Vadhan, "Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream," in *SODA'08*, S. Teng, Ed., 2008, pp. 746–755.

[32] E. Cambiaso, G. Papaleo, G. Chiola, and M. Aiello, "Slow DoS attacks: Definition and Categorisation," *IJTMCC*, vol. 1, no. 3-4, pp. 300–319, 2013.

[33] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui, "FloodDefender: Protecting data and control plane resources under SDN-aimed DoS attacks," in *INFOCOM'17*, 2017, pp. 1–9.

[34] Arista Networks, Inc, *Arista's EOS SDK*, 2023. [Online]. Available: github.com/aristanetworks/EosSdk.

[35] gRPC Authors, *gRPC*, 2023. [Online]. Available: grpc.io/.

[36] VMware, *RabbitMQ*, 2023. [Online]. Available: www.rabbitmq.com/.

[37] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, 2014.

[38] J. Cavat, *lp-modeler*, 2021. [Online]. Available: github.com/jcavat/rust-lp-modeler.

[39] Gurobi Optimization, LLC, *Gurobi optimizer reference manual*, 2020. [Online]. Available: http://www.gurobi.com.

[40] X. Zuo, Q. Li, J. Xiao, D. Zhao, and J. Yong, "Drift-bottle: A lightweight and distributed approach to failure localization in general networks," in *CoNEXT'22*, ACM, 2022, 337–348.

[41] Z. Xia, Y. Zhou, F. Y. Yan, and J. Jiang, "Genet: Automatic Curriculum Generation for Learning Adaptation in Networking," in *SIGCOMM'22*, 2022, 397–413.

[42] V. Đukić, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla, "Is Advance Knowledge of Flow Sizes a Plausible Assumption?" In *NSDI'19*, 2019, pp. 565–580.

[43] G. Sanjit and C. Ted, "Magic quadrant for network performance monitoring and diagnostics," *Gartner Research ID G*, vol. 354365, 2019.

[44] Y. Liang and L. Qiu, "Network Traffic Prediction Based on SVR Improved By Chaos Theory and Ant Colony Optimization," *IJFGCN*, vol. 8, no. 1, pp. 69–78, 2015.

[45] H. Song, F. Qin, P. Martinez-Julia, L. Ciavaglia, and A. Wang, "Network telemetry framework," RFC 9232, May 2022.

[46] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *SOSP'17*, M. Kaminsky and M. Dahlin, Eds., 2013, pp. 423–438.

[47] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-Wide Heavy Hitter Detection with Commodity Switches," in *SOSR'18*, 2018, pp. 1–7.

[48] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time," in *SIGCOMM'20*, H. Schulzrinne and V. Misra, Eds., 2020, pp. 226–239.

[49] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *SIGCOMM'16*, 2016, 101–114.

[50] T. Yang *et al.*, "Elastic Sketch: Adaptive and Fast Network-wide Monitoring," in *SIGCOMM'18*, ACM, 2018.

[51] Y. Zhang *et al.*, "CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query," in *SIGCOMM'21*, 2021, 207–222.

[52] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi, "Tierless Programming and Reasoning for Software-defined Networks," in *NSDI'14*, R. Mahajan and I. Stoica, Eds., 2014, pp. 519–531.