Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks

Tushar Swamy
Stanford University
United States of America

Annus Zulfiqar
Purdue University
United States of America

Luigi Nardi
Lund University, Sweden
Stanford University, USA

Muhammad Shahbaz
Purdue University
United States of America

Kunle Olukotun
Stanford University
United States of America

ABSTRACT

Support for Machine Learning (ML) applications in networking has significantly improved over the last decade. The availability of public datasets and programmable switching fabrics (including low-level languages to program them) presents a full-stack to the programmer for deploying in-network ML. However, the diversity of tools involved, coupled with complex optimization tasks of ML model design and hyperparameter tuning while complying with the network constraints (like throughput and latency), puts the onus on the network operator to be an expert in ML, network design, and programmable hardware.

We present Homunculus, a high-level framework that enables network operators to specify their ML requirements in a declarative rather than imperative way. Homunculus takes as input the training data and accompanying network and hardware constraints, and automatically generates and installs a suitable model onto the underlying switching target. It performs model design-space exploration, training, and platform code-generation as compiler stages, leaving network operators to focus on acquiring high-quality network data. Our evaluations on real-world ML applications show that Homunculus's generated models achieve up to 12% better F1 scores compared to hand-tuned alternatives, while operating within the resource limits of the underlying targets. We further demonstrate the high performance and increased reactivity (seconds to nanoseconds) of the generated models on emerging per-packet ML platforms to showcase Homunculus's timely and practical significance.

CCS CONCEPTS

• **Networks → In-network processing**; *Programmable networks*; • **Software and its engineering → Retargetable compilers**; • **Computing methodologies → Supervised learning**.

KEYWORDS

Per-packet ML; Self-driving Networks; ML Compilers

ACM Reference Format:
Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. 2023. Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3582016.3582022

1 INTRODUCTION

It is the *reaction time* that dictates how robust, performant, and secure a given network is [101]. For example, (a) mitigating faults quickly (such as gray failures [42, 101]) minimizes network downtime and improves availability; (b) reacting to traffic imbalance due to short-lived traffic bursts lasting a few microseconds (microbursts) [102], swiftly, alleviates network congestion [27, 37, 56, 91, 99] and server load [2, 51]; (c) identifying malicious behavior early limits network disruptions and attacks [9, 19, 63, 87]; and so on. At the same time, dealing with these events requires complex operations, e.g., Machine Learning (ML) inference, to decide what actions to perform as events happen, thereby affecting our speed and response time to tackle such events [80, 85, 97].

Up until now, these inference operations were carried out on a logically-centralized control plane [63, 86, 91, 99], with decisions stored as flow rules in the network data plane (i.e., switches and routers) [18]. The assumption here was that a decision made for the first packet would remain the same for all subsequent packets of a given flow (or connection). For routing and switching, where packets destined for a given destination must always reach the same server, this holds true. However, for performance and security objectives (like traffic engineering [9, 12, 22, 57] and threat mitigation [58]), it is not the case, as network conditions can vary quickly within the duration of a given flow; hence, rendering any cached (per-flow) decisions stale and obsolete.

Recently, with the emergence of programmable switches (such as Intel Tofino [46, 47] having P4 programmable match-action tables or MATs), SmartNICs and network accelerators (like Microsoft Catapult [75, 76], Azure AccelNet [32], Xilinx Alveo Data Center Accelerators [93] having a field-programmable gate array or FPGA), the network data plane is no longer limited to flow-caching only. These data planes can now execute more complex operations and ML models directly in the network at line-rate. For example, a P4-based switch can execute support-vector machines (SVMs) [30, 40, 63], K-Means [57], decision trees [5, 50] or binary neural networks (BNNs) [79, 80] directly in the data plane [97] to carry out tasks

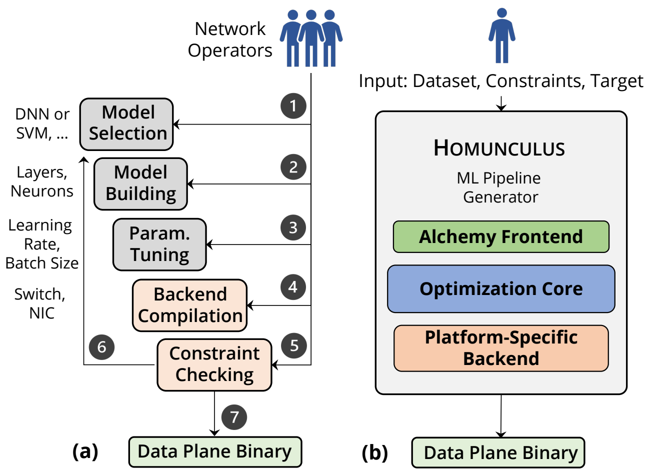

Figure 1: Comparison of data-plane ML pipeline development process. (a) Current approach: requires domain expertise in model selection, building, hyperparameter tuning, target compilation, and constraint checking; (b) HOMUNCULUS: requires only dataset, network performance constraints, and target type for resource estimation.

such as packet classification [50], load balancing [2, 51], resource-aware scheduling [61], and DDoS mitigation [54, 55, 58]. Similarly, SmartNICs and FPGA-based accelerators [21, 24, 32, 75, 76, 93] at end-hosts can run convolutional and recurrent deep neural networks (i.e.CNNs and RNNs) to dynamically adjust congestion windows [7, 99], adapt bitrates for incoming video chunks [60, 98], predict network queue sizes [38] from the edge [35], and more. Emerging data plane platforms, like Taurus [85], have enabled per-packet ML inference for more complex ML algorithms, such as deep neural networks at line rate.

Equipped with these programmable switches, NICs, and accelerators, datacenter networks can now react to network changes intelligently (running sophisticated ML models) and swiftly (at line speed). However, programming these data-plane ML pipelines today is extremely challenging, even for the most expert network operators in the industry. There is a stark difference between designing a new feature (e.g., adding or removing a protocol header) versus running a sophisticated model on a data plane [85, 97]. As shown in Figure 1, the latter requires domain expertise in ML model selection ❶, architecture construction, training ❷, and hyperparameter optimization (HPO) ❸, as well as an intimate familiarity with P4 and hardware-description languages (such as Verilog and VHDL) and emerging high-level DSLs (e.g.Chisel [8] and Spatial [53]) ❹, while meeting network constraints (latency and line rate) ❺. This process is iterated by network operators many times ❻ to compile these models on the underlying hardware backends ❼. So, to build and execute these data-plane ML pipelines, a network operator would have to be an expert in ML model development, hardware architecture, and compiler design, all at the same time—a Sisyphean task for the network operators.

In this paper, we present HOMUNCULUS, a framework that alleviates the burden from network operators and automatically generates efficient data-plane ML pipelines for the various use cases (Figure 1b). With HOMUNCULUS, network operators now only have to specify (a) the training data set (e.g., [87, 97]), (b) constraints of the operating environment, i.e., minimum throughput and acceptable latency, and (c) particular targets for the ML pipeline to run on (e.g., Tofino, Taurus [85], or else). HOMUNCULUS then performs design-space exploration (DSE) [68] to find a model suitable for the given use case, tunes the hyperparameters, and auto-generates the code for the specified backend. It iterates over these steps until the final output meets the constraints (or no feasible solution exists).

HOMUNCULUS leverages several key characteristics and recent advances in the fields of machine learning (including hyperparameter optimization [16, 67, 68, 84]) and networking (including programmable data planes [18, 85]). First, the recent proliferation of ML hardware accelerators (such as GPUs, TPUs, and NPUs) makes it possible to build large-scale systems for model search [39]; with open-source AutoML tools [15, 49], developers can now automate all stages of the ML development life-cycle, including model architecture search and training. Second, multi-objective black-box optimization frameworks (like HyperMapper [68]) decouple model training from model search (for resource estimation and compliance) [16, 28, 53, 67, 84]. Third, HOMUNCULUS exploits the unique characteristics of datacenter networks (including operating constraints, such as minimum throughput and maximum acceptable latency) to further minimize the search space of data-plane ML models by ruling out infeasible models during the search. Lastly, modern data centers—equipped with programmable data planes [18, 85], SmartNICs [45, 69], and FPGA-based network accelerators [32, 93] with open interfaces (such as P4 and Spatial)—allow HOMUNCULUS to automatically generate efficient code for these individual backend targets.

We make the following key contributions:

- **Alchemy Frontend (§3.1):** We introduce Alchemy, a framework that serves as a frontend for HOMUNCULUS to express user intent for a data-plane program in the form of objectives, data, and constraints.
- **Optimization Core (§3.2):** We illustrate methodologies for mapping multiple applications to a data-plane device. These methodologies form the optimization core of HOMUNCULUS, which takes the Alchemy input and explores the design space of ML model topologies, trains the models via supervised learning, and tests for constraints to find a compliant and well-performing model.
- **Backend Generator (§3.3):** A HOMUNCULUS backend, which generates Spatial [53] and P4 [17] code for Taurus [85] or MAT-based [18] switches, respectively, using the ML model generated by the optimization core.
- **Empirical Evaluation (§5):** We provide extensive evaluations and microbenchmarks of HOMUNCULUS using real-world applications. HOMUNCULUS's generated ML pipelines achieve much higher F1 scores compared to hand-tuned baselines for the Taurus and P4-SDNet (FPGA) data planes [44, 85] backends: 83.1, 68.75, and 79.8 versus 71.1, 61.04, and 77.0 respectively.

2 BACKGROUND & MOTIVATION

ML for Networking. ML algorithms are actively replacing existing heuristics (e.g., for load balancing, packet classification, resource scheduling, and intrusion detection) in datacenter networks [19, 48, 59, 89, 100]. These heuristics are narrow in scope and tackle specific aspects of networking (e.g., load balancing tailored to a leaf-spine

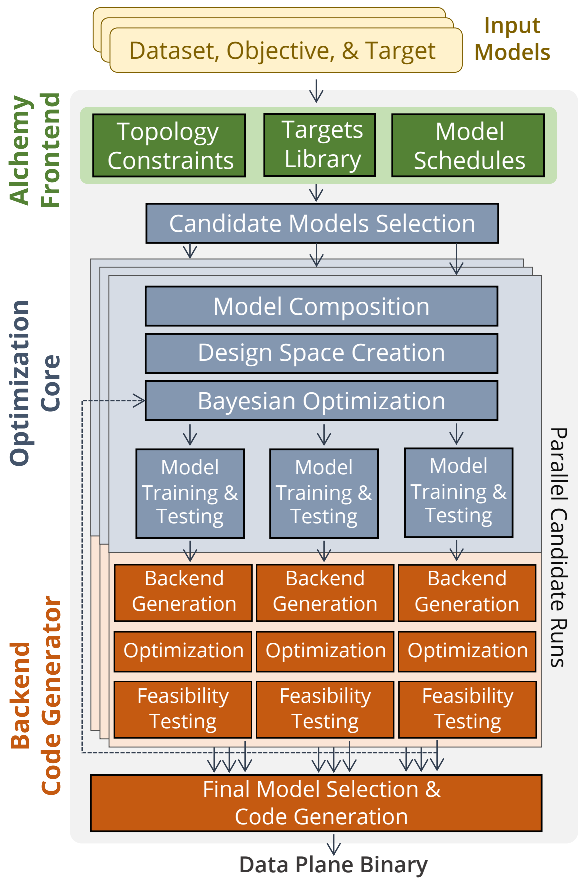

Figure 2: High-level HOMUNCULUS framework with the Alchemy frontend (top), the optimization core (middle), and the backend generator (bottom).

topology [2]) or require manual tuning to adapt to new operating conditions [3, 56, 104]; therefore, when running with other heuristics, they typically result in sub-optimal behavior [19]. Furthermore, one-size-fits-all heuristics, which simultaneously handle multiple aspects of a network, are complex or even impossible for human operators to design [65, 83].

ML, on the other hand, can handle these complex relationships efficiently. By training on the dataset captured for a given datacenter network, the model can customize itself to the particular environment. Over time, it can sample more data and retrain itself to reflect changes in the network [19, 48, 59, 60, 98, 100]—understanding relationships between activities that network operators may not have been aware of. For example, instead of just matching incoming flows against a statically known set of IP addresses, ML can learn the correlation between fine-grain features (e.g., connection duration, bytes transferred, protocol type, service type, packet size, and arrival time) to make informed decisions in new and unseen scenarios [26, 85, 87]. The rise in the number of ML-based use cases in the last couple of years (e.g., traffic classification [97], optimal routing [73], queue management [35, 38], and other network operations [19]) further highlights (and strengthen) the importance of ML for networking.

Platforms for Data-Plane ML. With the advent of programmable data planes, the networking community is actively devising methods and platforms to run ML models on such data planes [79, 80, 85, 97]. Recent efforts primarily focus on transforming a manually-designed, pre-trained model on the underlying data-plane target using programming abstractions, like P4. For example, IIsy [97] is a data-plane pipeline that maps classical ML algorithms (such as SVM, KMeans, and decision trees) onto existing MATs in PISA switches. They exploit the structural similarity between these algorithms and the layout of MATs, which lend themselves to efficient hardware implementations. Similarly, Taurus [85] introduces a new compute block (MapReduce) and an accompanying abstraction (Spatial [53]) in PISA switches—a dense array of compute and memory units, structured in a SIMD-like pattern, capable of executing various ML models. The SIMD patterns (map and reduce) enable efficient implementation of linear-algebra-based ML algorithms at line rate. However, the onus of developing a model that can meet the constraints and fit within the switch resources is still on the network operators (i.e., to manually design, transform, and program these models using low-level switch abstractions).

Automated ML Frameworks. ML methods are sensitive to many design decisions, which poses significant barriers during model development and training. This is particularly visible in the emerging field of artificial neural networks (ANNs), where ML developers are charged with selecting the suitable neural architectures, training procedures, and regularization methods along with tuning their hyperparameters to achieve high prediction accuracy [10, 65]. Today, the developers are left with tedious episodes of trial-and-error until they identify a feasible set of choices for a particular ML application, given its dataset.

The systems and ML communities are placing considerable efforts in automating these decisions by providing useful abstractions for the developers [43, 68]. With open-source frameworks, such as AutoPytorch [105] and Auto-sklearn [31], a new field of automated ML (AutoML) [43] is emerging to provide dutiful abstractions and runtimes to automatically generate trained ML models given only the training dataset as input. These algorithms cover a wide range of algorithms to choose from. For example, Neural Architecture Search (NAS) frameworks [15, 31, 105] for Convolutional Neural Networks (CNNs) attempt various permutations of well-tested, pre-built NN basic blocks (combinations of convolution kernels, activation functions, batch normalization, and more) to explore the design space of CNNs to find an optimal model for the task at hand. Similarly, frameworks like AutoKeras [49] let users specify ranges of tunable parameters (e.g., whether to perform data augmentation and/or data normalization). The underlying systems explore this range, stochastically picking a set of (tunable) parameters while keeping track of various metrics (e.g., accuracy or F1 score) for each iteration, ultimately selecting the best-performing model. Lastly, frameworks such as HyperMapper [53, 68] use advanced Bayesian optimization methods to formulate the task of model search as a black-box optimization problem with a limited optimization budget. The goal is to establish algorithms that traverse the large search space with minimal trial and error [16, 67, 78]. For example, HyperMapper adopts a Random Forest (RF) surrogate model instead of Gaussian Process (GP) models to reduce the computation overhead—without resorting to more advanced approaches [4], GPs could suffer the overhead of building co-variance matrices and inverting them.

In this paper, by exploiting the domain-specific characteristics of networking, we extend these algorithms and devise an AutoML approach for generating data-plane ML pipelines to systematically and automatically search a design space and train models while adhering to the various performance constraints dictated by the datacenter networks and target data-plane resources.

3 THE HOMUNCULUS DESIGN

Homunculus provides a high-level and declarative interface for network operators to specify their application objectives (e.g., minimizing false positives in an anomaly-detection model or maximizing the throughput of a traffic-classification algorithm). Operators only specify the datasets—either proprietary or publicly available—along with the application objectives and a target backend (Figure 2); together, these implicitly describe the desired application behavior (e.g., anomaly detection or packet classification). Given these datasets and objectives, Homunculus explores appropriate model architectures (e.g., KMeans, SVMs, or DNNs), and automatically generates an optimal binary for the target backend (e.g., Taurus [85]) while respecting the environmental constraints (i.e., network topology, hardware resources, and service-level objectives). It hides the low-level implementation details—that would otherwise require domain expertise in model selection and building, parameter tuning, backend compilation, and constraint checking (Figure 1a)—and allows operators to focus on curating network datasets and specifying performance goals (which they have expertise in).

3.1 The Alchemy Frontend

We implement Alchemy as an embedded framework (inside Python) that provides a declarative interface for the user to interact with Homunculus. The user, being a domain expert, specifies the dataset for their respective ML application (without explicitly writing the model definitions) and an optimization objective to reduce data bias (e.g., minimizing false negatives for intrusion detection). Alchemy, in essence, defines this *performance* abstraction that works alongside the functional description of the switch, provided by a networking DSL (like P4).

Figure 3 shows our anomaly-detection (AD) application written using the Alchemy frontend. Alchemy provides several constructs (classes and operators) to express and convey the programmer's intent to Homunculus, i.e., the training dataset, application objectives and constraints, as well as interactions with other models or interfaces (to external devices). The `DataLoader` decorator wraps a custom function listing which datasets to use in the model search. The function loads and preprocesses the dataset into a form that Homunculus can parse. Next, Alchemy's `Model` class specifies the objective metric (and an optional list of algorithms) to measure the performance of the candidate models. To enforce environmental constraints (e.g., data-plane resources, throughput, and latency), Alchemy supports the `Platforms` class that contains a list of supported backends (e.g., Tofino, Taurus, and P4-SDNet) with their accompanying constraints. Moreover, Alchemy provides a collection of scheduling operators to specify how multiple models interact with each other (either sequentially or in parallel). Finally, the `generate` function informs Homunculus to start model search and code generation. Table 1 lists the various constructs available in Alchemy.

```python
import homunculus
from homunculus.alchemy import
    DataLoader, Model, Platforms
import ad_loader

@DataLoader # training data loader definition
def wrapper_func():
    tnx, tny = ad_loader.load_from_file(
        "train_ad.csv")
    tsx, tsy = ad_loader.load_from_file(
        "test_ad.csv")
    return {
        "data": {"train": tnx, "test": tsx },
        "labels": {"train": tny, "test": tsy }}

# Specify the model of choice
model_spec = Model({
    "optimization_metric": ["f1"],
    "algorithm": ["dnn"],
    "name": "anomaly_detection",
    "data_loader": wrapper_func })

# Load platform
platform = Platforms.Taurus()
platform.constrain(
    "performance": {
        "throughput": 1, # Giga-Packets/second
        "latency": 500 },   # ns
    "resources": { "rows": 16, "cols": 16 })

# Schedule model and generate code
platform.schedule(model_spec)
homunculus.generate(platform)
```

Figure 3: Alchemy syntax for the anomaly-detection use case. The pipeline is defined for the Taurus switch, scheduling a single model on the data plane (no model composition).

3.2 The Optimization Core

Homunculus's optimization core performs a design space search for model selection that maximizes the application objective while respecting the data-plane resources and network constraints. The core chooses among various ML algorithms and implements the model generation and data-plane mapping steps as a Bayesian-optimization problem [25, 33, 72] when selecting the best performing configuration.

3.2.1 Candidate Models Selection & Composition. Homunculus aims to satisfy application objectives by exploring a variety of ML models (from the pool of supported algorithms). The core initiates multiple parallel runs to find the most efficient and performant model for the given application. We take inspiration from other (iterative) compiler systems (such as Xilinx Vivado [96]) that execute multiple parallel strategies to find the optimal resource placement and timing closure [94, 96], often called recipes. Some algorithms may consume fewer resources while others may perform better with bigger datasets. As a first step, the core tries to rule out as many algorithms as possible based on the data-plane platform and network constraints. For example, the number of multiplication and addition operations required for a modest DNN model can quickly exceed the computational capacity of a MAT-based switch; however, the mapping would indeed be feasible if ample MATs are available [80].

Next, when mapping multiple models to a given target, the optimization core ensures that (individual) model constraints are

Table 1: Alchemy's constructs: classes and operators.

Construct	Symbol	Description
`Model`	Model(optimization_metric, algorithm, data_loader, ...)	Specify model objectives and datasets
`@DataLoader`	@DataLoader()	Load and preprocess model dataset
`Platforms`	Platforms.[Taurus, Tofino, FPGA]	Declare a backend target
`>, |`	Platforms.schedule(mdl1 > mdl2) or .schedule(mdl1 | mdl2)	Sequential > and parallel | composition
`IOMap`	IOMap(mapper_func)	Connects models' inputs and outputs
`@IOMapper`	@IOMapper([io_ins], [io_outs])	Specifies input to output mapping
`<`	Platforms < (performance, resources)	Apply network and data-plane constraints

consistent with each other. For example, if one model operates at 1 Giga-Packet/second throughput and feeds into another model operating at 0.5 Giga-Packet/second, the first model must also operate at 0.5 Giga-Packet/second. The core further disqualifies any such models that fail to meet these constraints.

3.2.2 (Automated) Design Space Creation.

After creating a list of candidate algorithms, Homunculus's core uses the accompanying models' parameters and constraints to build a design space. Homunculus defines the search space by setting upper and lower bounds for these tunable parameters (e.g., the minimum and maximum number of neurons in a DNN layer or the range of learning rate values attempted) to determine models' performance. These bounds are applied to three sets of variables:

Hyperparameters. These are parameters of an ML model that are not decided by the training process. For a DNN, these include parameters for the neural architecture search, such as the number of layers and neurons as well as training parameters (e.g., learning rate and batch size). Traditionally, these parameters are hand-tuned. They have a significant impact on the performance of the resulting model; however, the massive design space they cover makes hand-tuning extremely challenging. Homunculus instead explores the hyperparameter design space using Bayesian optimization (BO), by assigning upper and lower bounds to these parameters—typically calculated based on the target being considered.

Physical Resources. The availability of resources on a data-plane platform has a large impact on the types of algorithms that the platform can support. For example, MATs in Tofino and the extent of loop unrolling in Taurus place restrictions on the supported models. In Homunculus, we encode data-plane resources (such as CUs, MUs, and MATs) as feasibility constraints; if a certain algorithm configuration (e.g., number of layers or neuron count of DNN) exceeds the provided resources, that configuration is marked infeasible and removed from the candidate set. Subsequent iterations of the Bayesian optimization will recommend model configurations that use fewer resources.

Network Constraints. Network topology imposes further performance constraints. For example, a model running on a switch in the core of the network must operate at multi-terabits per second, whereas on an end-host NIC, it can run at 40 Gbps–100 Gbps. As with resources, Homunculus encodes these constraints as feasibility requirements, providing further opportunities for the optimization process to drop infeasible model configurations.

3.2.3 Bayesian-Optimization (BO) Guided DSE.

With three classes of variables discussed earlier, Homunculus puts a finite bound on a previously infinite design space. Each additional variable expands the design space and adds complexity to the optimization process. The difficulty of managing a space of this magnitude quickly outpaces human capabilities. And, at a certain point, it can even hamper automated search processes as well. Fortunately in Homunculus, rather than expanding the search, the additional variables for physical resources and network constraints help reduce the design space by disqualifying infeasible configurations, quickly. Even still, the search space is too large to manage using basic heuristics and demands efficient optimization processes to guide DSE. We formulate the DSE as a black-box optimization problem (as discussed in Appendix A).

3.2.4 BO-Suggested Model Training, Testing, & Fusion.

With the BO-suggested hyperparameter configurations, the optimization core can now use these parameters to build an ML model and train it with the user-specified dataset. The model is trained with a loss function corresponding to the metric designated by the application objectives, such as accuracy or an F1 score. These metrics will be returned to the DSE stage (§3.2.3) to inform the next set of suggested hyperparameter configurations.

To further save resources, some models can be combined into a single model. Models operating on similar datasets are most likely learning similar characteristics [88, 90]. Since most data coming into a data-plane model comes in the form of packets, datasets will have a number of features in common (e.g., packet headers). Homunculus will assess the feature sets for similarities and if there are certain features in common, it will attempt to build a single model to serve both datasets by sharing learned characteristics between tasks, while also reducing resource usage by eliminating redundancies between learned weights. At this point, the only elements left to evaluate are the feasibility constraints. To test this, Homunculus generates the hardware code (using ML model templates for each platform) to map the model to the underlying backend.

3.3 The Backend Code Generator

Each target platform that Homunculus supports (e.g., Taurus, Tofino, or P4-SDNet FPGA) has an associated backend compiler to allow for performance and feasibility testing. Each backend is responsible for generating the platform's hardware code that can be tested in either simulation or on a physical testbed. We utilize cycle-accurate simulators (such as the SARA framework [103] for Taurus) that allow us to precisely measure resource utilization and

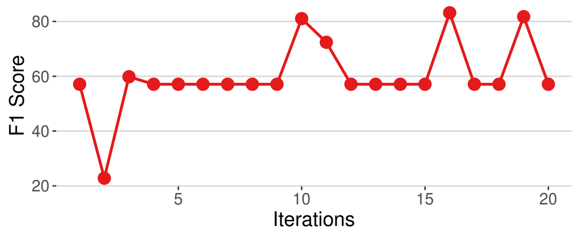

Figure 4: Regret plot with F1 score metric for anomaly-detection DNN on the Taurus' Map-Reduce grid.

latency/throughput of the ML model under test. Code is generated by assembling existing templates for common ML components before optimizing and testing.

After a predetermined number of optimization iterations, the best performing, constraint-compliant model is selected, and its data-plane platform code and binary are generated. The amount of iterations an application takes to reach a desirable configuration is highly dependent on situational factors, such as the resources and data available. To give programmers an intuition of how this may affect their application, we show a plot in Figure 4 for the AD application with the achieved F1 score in each optimization iteration. We see that while initial results are poor, Homunculus quickly begins to find a stable F1 score. For the in-network ML applications evaluated in §5, we observe that Homunculus reliably achieves satisfactory results after 20 iterations (as in Figure 4); newer (more complicated) applications may require more iterations. Once it finds a variant that performs significantly better, it trades off between its earlier stable configuration and the discovery of more high-performing variations, effectively balancing the exploitation of known parameter combinations and exploration of unknown ones.

Template-based Code Generation for ML Models. To efficiently generate code for a backend, we use parameterized templates for commonly used operations (e.g., feature extraction, activation functions, dot product, and more). The parameters are calculated as a function of the optimization core's suggested configurations. These template blocks can then be assembled into larger blocks or even full packet pipelines. As an example, consider a Taurus switch, whose MapReduce block is programmed using the Spatial DSL [53]. Homunculus starts with simple parameterized constructions, such as dot products, and builds up to larger structures, like matrix multiplication and eventually DNNs. It expresses simple dot products as a map operation to perform element-wise multiplication with an addition-based reduction to combine the results into a scalar. These dot products then nest inside an additional map operation to build a full DNN layer [74, 85]. Homunculus controls the exact number of neurons and layers with parameters dictated during the optimization core phase (§3.2), and places the trained weights on the on-chip memory. It then stitches these layers together by storing intermediate results for each layer in double-buffered SRAM blocks that feed into subsequent layers. Owing to the regular structure of ML algorithms (e.g., KMeans and SVM) the code-generation process for other platforms (like Tofino using P4 templates) follow a similar methodology.

Feasibility Constraint Testing for Generated Models. Once the hardware code has been generated for an ML model, it can be mapped into the testing infrastructure. The testing setup is required to provide verdicts on the feasibility constraints that the optimization core phase is querying. In particular, the testing infrastructure is responsible for computing throughput and latency as well as identifying whether the application can be mapped within the available resources. This is done using hardware testbed platforms or cycle-accurate simulators (e.g., Tungsten [103] for Taurus or Xilinx Vivado for P4-SDNet FPGAs) depending on the particular backend.

4 IMPLEMENTATION

Alchemy Frontend. We implement the Alchemy frontend in Python. The embedded nature allows programmers to import Homunculus functions along with Alchemy constructs and use them in conjunction with existing Python primitives and libraries. In addition, the ubiquity of Python in machine learning, data science, and dataflow frameworks means that functions in these domains (such as Pandas [62], TensorFlow [1], or Theano [13]) can be incorporated into Alchemy's decorator functions (`IOMapper` or `DataLoader` or even its compositional operators.

Optimization Core. Homunculus's optimization core employs Bayesian optimization (BO) to maximize user objectives while meeting feasibility constraints. To perform our optimization, we use HyperMapper [68], a framework for constrained multi-objective optimization, along with standard machine-learning frameworks (like Keras [23] and Tensorflow [1]). Compared to other frameworks for BO (e.g., OpenTuner [6], HyperOpt [14], or GPflow Opt [52]), HyperMapper provides a more comprehensive feature set, such as multi-objective optimization, varying parameter types (real, ordinal, integer, and categorical variables), and feasibility testing, to allow Homunculus to automatically search and generate efficient ML models. For example, multiple optimization objectives let users specify multiple goals for their ML model (e.g., high F1 score and low false negatives). Likewise, the wide set of parameter types allows BO to explore a range of ML hyperparamaters, such as learning rate (real), parallelization level (ordinal), number of layers (integers), or types of activation functions (categorical). Finally, feasibility testing during the BO process reduces the search space by ruling out ML configurations that fail to meet the resource and performance constraints.

Homunculus parses the design-space restrictions from the application's program (written in Alchemy) and exports it as a JSON configuration file (describing searchable parameters). HyperMapper read this JSON file to start the optimization process. As the optimization is running, Homunculus receives regular parameter suggestions from HyperMapper, and evaluates them for both the user's objective and adherence to feasibility constraints in target backends. The outcomes of these evaluations are then sent to HyperMapper to guide further design-space exploration.

The Backend Code Generator. For backend targets (like the Taurus [85] and P4-SDNet FPGA), we use an intermediate hardware-description language called Spatial [53]. Spatial relies on loop-level constructs to describe applications and translate them into a bitstream that can be applied to the underlying data-plane fabric (e.g., Taurus or a FPGA). The loop-level constructs make functional operators (like map and reduce) available, which allows our templates

Table 2: Comparison of hand-tuned baseline models vs. Homunculus generated models for Taurus ASIC (CU/MU: compute/memory units).

Application	Features	# NN Param	F1 Score	CUs	MUs
Base-AD	7	203	71.10	24	48
Hom-AD	7	254	**83.10**	41	67
Base-TC	7	275	61.04	31	59
Hom-TC	7	370	**68.75**	54	97
Base-BD	30	662	77.0	167	45
Hom-BD	30	501	**79.8**	53	151

to be highly parallel and achieve high performance. In conjunction with Spatial, we use compiler frameworks (such as SARA [103]) to enable efficient mapping to tile-based reconfigurable architectures (e.g., Taurus).

We also support common architectures such as the MAT-based pipelines found in switching architectures (e.g., Tofino [47] or the P4-SDNet FPGA [44]). In these architectures, the number of available MATs becomes the constraining resource. We use IIsy [97] as a backend for mapping ML algorithms (such as SVMs or KMeans) to MATs. IIsy makes the relation between algorithm parameters and MATs explicit, a relation that can be exploited as a constraint by Homunculus. Homunculus will tune parameters in the algorithm that translate to MATs and attempts to fit generated models into the limited pipeline resources. For example, IIsy shows that an implementation of an SVM may use a MAT per feature. If the number of MATs is insufficient, Homunculus will try to remove less impactful features until the SVM model fits.

5 EVALUATION

To investigate the effectiveness of Homunculus, we test the compiler stack with real-world applications, addressing problems in network security and traffic classification. The models generated from Homunculus are constrained to 1 Giga-Packet/second line-rate throughput for all applications. We evaluate several microbenchmarks to demonstrate the capabilities of the Homunculus framework in terms of resource budgeting and support for alternative data-plane architectures. For all these experiments, we setup HyperMapper to use the Random Forest [20] surrogate model, which is known to work well with systems workloads that require modeling of discrete parameters and non-continuous functions [68]. We select the Expected Improvement criterion [64] and a uniform random-sampling initialization phase followed by Bayesian optimization iterations.

Baseline Applications. We evaluate the performance of Homunculus' generated models using a set of real-world applications (Table 2). Our first baseline is a hand-crafted anomaly-detection (AD) model from [85] and [86], rewritten in Spatial [53] and trained offline on labeled packet-level traces from the NSL-KDD [26] dataset. We also evaluate Homunculus on a traffic classification (TC) application, shown in IIsy [97], and a botnet-detection (BD) application, found in Flowlens [11]. The TC application is built from IoT device traces in a data center and requires that an application correctly identifies the device type from packet-header features (packet size, Ethernet, and IPv4 headers). The original TC models from IIsy [97] are statistical models (SVM, K-Means, and Decision Trees), but we

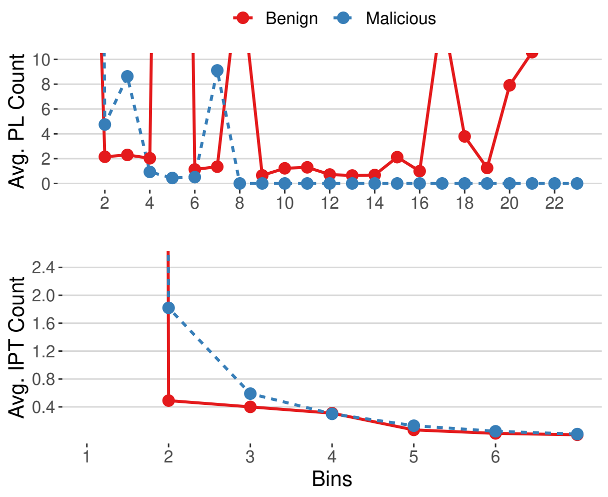

Figure 5: Botnet vs. benign flow-level packet length (PL) and inter-arrival time (IPT) histograms averaged across all flows on Taurus ASIC.

create a hand-written DNN baseline with three hidden layers (10, 10, and 5 neurons) to test against Homunculus's DNN generation capability for a fair comparison. The BD application is built from a dataset consisting of P2P workloads that include traces from botnets (e.g., Storm and Waledac) and benign traces from uTorrent, Vuze, eMule, and Frostwire [77]. The botnet traffic can be segregated from benign P2P traffic by analyzing the histograms of packet sizes and inter-arrival times at the flow level.

5.1 Microbenchmarks

5.1.1 Homunculus Reaction Time. The botnet detection (BD) problem has been studied, primarily, at the conversation level (tracking source and destination IP, while ignoring ports) [11, 66, 77]. The idea is to aggregate packet sizes and packet inter-arrival times into coarse-grained histograms (called *flow markers*) for up to 3,600 seconds (even on programmable switches like Tofino [11] using registers) before making a prediction of benign or malicious (botnet) flow. This is possible because botnets communicate via low-volume and high-duration flows compared to benign P2P applications, which makes them identifiable using their packet size and inter-arrival time histograms over the duration of their flows [11, 66, 77].

We demonstrate that by making predictions on per-packet-level partial histograms, we can greatly reduce the reaction time of a BD model compared to using full flow-aggregated flow markers. Figure 5 shows histograms of packet sizes (bin size: 64 bytes) and inter-arrival times (bin size: 512 seconds) for benign and malicious classes averaged over the entire traffic set when running on the Taurus backend. We observe that due to characteristic differences in the network traffic of benign and botnet P2P traffic, the resulting histograms of both kinds of applications start to look different—as certain bins are not expected to fill for botnet applications—early on, even with few packets seen. This insight and empirical results serve as evidence for the need for per-packet ML and AutoML pipeline generators (such as Homunculus).

5.1.2 Homunculus vs Baselines Resource Usage. As shown in Table 2, for AD, TC, and BD applications, Homunculus is able to build models that outperform the hand-tuned baselines. This is because Homunculus is aware of the platform and environment where the application is being run and can make better use of available resources. Using more resources means choosing a bigger model (for example, more layers and neurons for a DNN), which can result in an improvement in the model's test performance. This is evident from the Compute Unit (CU) and Memory Unit (MU) usage numbers (in Table 2) on a Taurus platform. We argue that this is a point in favor of the compiler. If resources are unused, the platform will simply be under-utilized. However, it is challenging for users to cater models to the available resources without knowledge of the platform and environment. Bridging this gap results in potentially higher improvements in performance. In the case of AD, we see a full 12-point increase—this could be the difference between allowing malicious traffic to compromise a system and a stalwart defense.

For both the BD models, training was done on full flow-level histograms, while the F1 scores are reported on the per-packet-level partial histograms (using 120 Million test packets). The original FlowLens [11] model for BD used a flow-marker size of 151 bins (94 bins for packet size, the rest for inter-arrival time), while for our baseline and generated application, we use only 30 bins (23 for packet length and 7 for inter-arrival time) by fusing smaller bins into larger ones. With the BD application, contrary to AD and TC, the baseline is the bigger model (4 hidden layers of 10 neurons each) in terms of parameter count and resource usage, as shown in Table 2. Yet, Homunculus manages to outperform the baseline model with a smaller model (10 hidden layers with a smaller neuron count per layer) by distributing neurons across more layers. The baseline BD model is more compute-intensive (using 167 CUs) due to increasing hidden-layer compute demands, while the generated model uses more memory units (151 MUs) because it needs to store the neurons and weights of a larger number of layers. The FlowLens [11] BD model can perform with a high F1 score on large flow-level histograms accumulated over 3,600 seconds; despite our reduction in feature size and performing on per-packet-level histograms, we still manage to get an F1 score of 77.0, which is much more reactive than waiting for an entire hour before labeling a malicious flow. Moreover, Homunculus manages to overcome the shortcomings of a hand-tuned design and offers a better-performing model with an F1 score of 79.8. Thus, not only are we able to reduce flow-marker size by 5× (hence increasing the number of flows we can handle on a switch proportionally), but we also reduce the reaction time from 3,600 seconds to a few hundred nanoseconds, while improving the F1 score.

5.1.3 Multi-Application Scaling & Model Fusion.

We also demonstrate Homunculus's ability to support multiple applications on a single target. The Alchemy frontend allows a user to specify how different models interact with each other via sequential and parallel operators (Table 3). Here, we show several examples of application chaining and how it affects resource usage. We chain copies of the anomaly-detection (AD) DNN in various configurations to show the execution of multiple user models on a single Taurus switch. In Table 3, we see that the increase in resources for different chaining strategies stays constant with the number of models, regardless of the strategy itself. This is because additional logic for managing models is negligible and can be fit into existing CUs, already in use; hence, allowing for much more efficient scaling of applications and sharing of backend resources (like Taurus).

Table 3: Resource scaling for different application chaining strategies using a Taurus ASIC.

Model	CUs	MUs
DNN > DNN > DNN > DNN	24	24
DNN | DNN | DNN | DNN	24	24
DNN > (DNN | DNN) > DNN	24	24

Table 4: Fused resource usage in Taurus ASIC (CU/MU: compute/memory units).

Application	CUs	MUs
AD: Part 1	44	81
AD: Part 2	51	96
AD: Fused	48	83

Table 5: Timing breakdown of Homunculus's various phases for our AD DNN. The backend is specific to a particular target/vendor (e.g., Xilinx).

Homunculus Phases		Latency (s)
Alchemy	IR Generation	0.0015
Opt. Core	DSE Creation	0.1535
	BO Initialization (10)	62.2593
	BO Iterations (10)	61.0445
Backend	Target Specific	–

For model fusion, in Table 4, we perform an experiment that divides the dataset of our AD application into two separate models. The two separate models map onto the same switch, with each allocated half of the switch's resources. Since the datasets of the two models will share features, we can let Homunculus fuse them into a single model. Table 4 shows the resource counts for the two split models (AD: Part 1 and AD: Part 2) evaluated individually, and the resource count for a fused model that uses both datasets to create a single model. The resource count is about the same for the fused model since the two split models each learn the same network characteristics. Instead of duplicating this knowledge, Homunculus encodes it into a single model, effectively cutting the resource usage by a factor of two.

5.1.4 Homunculus Compilation Times.

Table 5 shows the time taken by each of the different phases in Homunculus when compiling our AD DNN model.[1] We measure the end-to-end time Homunculus takes to compile the DNN—from Alchemy's IR generation to DSE creation in the optimization core (along with BO initialization and iterations). The time to generate an input (e.g., Spatial or P4) for a backend is included in the results; however, the time to generate the final bitstream or binary is target-/vendor-specific, e.g., Xilinx [96] or Taurus [85] (not shown).

The IR generation as well as DSE creation, each takes less than a second, while the various other Bayesian optimization (BO) steps take about a minute per 10 iterations. The majority of the time will

[1]We ran these experiments using a commodity server machine (with an Intel i7-4790 CPU @ 3.6 GHz and 16 GB of main memory); these machines are readily available from vendors (like Dell) and are deployed in data centers and public clouds (e.g., Microsoft Azure, Google Cloud, and AWS).

Table 6: Resource consumption and utilization of applications running on the Taurus FPGA testbed [85].

Application	Model	LUT%	FFs%	BRAM%	Power (W)
Loopback	-	5.36	3.64	4.15	15.131
Base-AD	DNN	6.55	4.30	4.15	16.969
Hom-AD	DNN	6.61	4.43	4.15	17.440
Base-TC	DNN	6.69	4.48	4.15	17.553
Hom-TC	DNN	7.48	4.77	4.15	18.405
Base-BD	DNN	7.29	4.68	4.15	17.807
Hom-BD	DNN	6.72	4.49	4.15	17.309

be incurred at the backend verification and bitstream generation steps, which can take minutes to hours depending upon the particular target. However, we note that the overhead of Homunculus's additional phases (§3) is minimal compared to the target-specific bitstream generation. Furthermore, a model is developed once, ahead of time, and only the final model runs on the backend.

5.2 End-to-End Hardware Evaluation

Testbed Setup. We use the Taurus testbed [85] for our end-to-end evaluation. A 32-port programmable Tofino Wedge100BF-32x switch running Stratum OS [71] is used to implement the PISA pipeline of the Taurus [85] switch. Its MAT processing pipelines are configured for pre- and post-processing, as necessary, to manage the Taurus ML core, which is emulated as a bump-in-the-wire FPGA. The switch bypasses its internal traffic through a Xilinx Alveo U250 FPGA [93] over a 100 Gbps connection, which is used to emulate the MapReduce ML logic of a Taurus switch. A CMAC core [95] with an AXI interface is used to forward the packets to the FPGA. The control plane runs the ONOS controller [70] and a Python REST API is used to install forwarding rules on our switch. Two 80-core Intel Xeon servers generate and receive traffic via MoonGen [29]. Both, the baseline and the Homunculus-generated models are compiled to Verilog using the Spatial compiler and downloaded to the FPGA for evaluation. (All models operate at line rate.)

5.2.1 Data-Plane ML Pipelines on the Taurus Testbed.

Resource Usage. Table 6 summarizes the FPGA power and resource consumption of the seven models we test using the Taurus testbed. Power consumption and resource utilization of loopback is due to the *bump-in-the-wire* FPGA, which would be non-existent on an actual Taurus ASIC. We can observe from Table 6 that for the AD and TC applications, Homunculus generates larger models with higher layer/neuron count, which consume a higher number of FPGA resources (Flip Flops, BRAM, LUTs) and, consequently, also consume higher power. The Homunculus model for the BD application consumes slightly fewer FPGA resources compared to the baseline model because the number of parameters in the baseline model is higher—resulting in a larger difference in LUT consumption compared to other resources (LUTs store the parameters of a model in the FPGA).

Performance. Table 7 shows the ideal and achieved accuracy, as well as speeds, of our three applications (anomaly detection, traffic classification, botnet chatter detection) when running on the Taurus testbed [85]. We see that Homunculus is able to generate FPGA hardware description code for a variety of applications, and the generated code is able to achieve the ideal F1 score—calculated offline in software (Table 2)—while maintaining line-rate performance.

Table 7: F1-Scores of generated applications running on the Taurus FPGA testbed [85].

Application	Ideal F1	Achieved F1	Line-Rate
Anomaly Detection	83.10	83.10	Yes
Traffic Classification	68.75	68.75	Yes
Botnet Chatter Detection	79.8	79.8	Yes

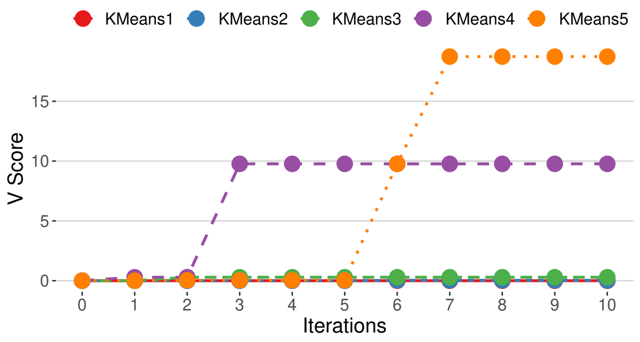

Figure 6: Regret plot with V-Measure score metric for KMeans on match-action tables (MATs).

5.2.2 Data-Plane ML Pipelines on MAT-based switches.

Homunculus can support existing architectures, like MAT-centered pipelines (e.g., Tofino). We show the results for an existing ML-for-switches platform, IIsy [97], plugged into Homunculus as a backend. Homunculus generates input for IIsy, which then compiles onto underlying P4 switches. [2]

Resource Usage. Homunculus conforms the TC algorithm to the constraints dictated by IIsy and the switch hardware (MATs in this case). IIsy restricts a single MAT for each cluster, where a cluster corresponds to one of the five traffic classes. This means that these KMeans models will consume in the range of one to five MATs. For switches with fewer tables, Homunculus creates more coarse-grain clusters, sacrificing fidelity in favor of resource usage.

In Figure 6, we show the V-measure score for a Homunculus-generated KMeans with varying resource availability. The objective is to cluster traffic from multiple IoT devices into a number of groups using packet header features. In the IIsy framework, a KMeans implementation can be mapped onto MATs. However, each cluster grouping takes up an additional MAT. In Figure 6, we see five generated implementations of KMeans traffic classification using the IIsy backend, each with different resource constraints. KMeans5 (K5) represents the application with 5 available tables, K4 has 4, and so on. Homunculus automatically generates models to fit each of the different resource constraints by dropping clusters and opting for coarser groupings at the cost of accuracy (as measured by the V-score).

Performance. An advantage of the Tofino architecture [46, 47] is that it implements a reconfigurable match-action table pipeline [18] to run at line-rate (1 Giga-Packet per second). Unless a packet is directed to recirculate through the packet pipeline, all operations mapped to the pipeline will run at the switch's line rate. Since IIsy is designed to map algorithms into a P4 program and, by extension,

[2] Other approaches to mapping ML models to MATs (e.g., as BNNs in N3IC [81]) can also be added as new backends to Homunculus.

P4 platforms (like the Tofino architecture); a successful generation of the parameters for our KMeans test case, therefore, runs on the Tofino architecture at full speed.

6 LIMITATIONS & FUTURE WORK

Homunculus allows network operators to express and deploy data-plane models without requiring extensive expertise in ML model development. However, additional research is needed to (1) co-compile data-plane ML models with traditional network applications, (2) produce high-quality datasets for more performant and accurate models, and (3) allow faster model retraining and deployment.

Co-Compiling ML Models with Network Applications. At present, Homunculus generates ML models in isolation without taking into account other networking applications (e.g., routing and switching). Ideally, a compiler should consider both the models and other networking tasks, which need to run on the same data plane, and devise an optimal division and distribution of resources among them. Doing so can allow a compiler to find opportunities for resource sharing not just among models (as in Homunculus) but also across other non-ML data-plane applications.

Collecting Higher-Quality Network Datasets. The performance and accuracy of the data-plane ML models not only rely on how efficient a compiler is in mapping a particular model to the underlying target but also on the quality of the dataset it is trained on. Therefore, concerted efforts across industry and academia are needed to collect high-quality datasets, e.g., for capturing fine-grained events (i.e., packet-level traces rather than just flow-level traces). Moreover, the collected datasets must be cleaned and should clearly depict the trends the operators intended their models to learn. Lastly, we need mechanisms to systematically collect and open-source these datasets to the broader community (in a privacy-preserving way) to use and extend upon [41].

Enabling Faster Model Retraining and Deployment. For hardware feasibility testing and mapping, Homunculus relies on backend-specific tools (e.g., Xilinx [96] and SARA [103]). Thus, to expedite model retraining and deployment, the processing time of these tools needs to be improved. For retraining, rather than running the entire synthesis and place-and-route stages, we can explore ways to accelerate/omit certain stages (e.g., by reducing or parallelizing optimization effort [36, 92]) to return resource estimates quickly—using the full set of optimizations for deployment only.

Enforcing Safety Nets for In-Network ML Systems. In-network ML primarily assists with the non-deterministic aspect of networking: performance (e.g., load balancing and traffic engineering [97]) and security (e.g., anomaly detection and firewalls [86]). For deterministic operations, such as ensuring packets destined to a particular server always reach that server (e.g., routing and switching [17]), rule-based MATs are sufficient. These MATs can also be used to enforce deterministic bounds on the output of probabilistic ML models and restrict their decisions [85] to implement safety nets.

7 RELATED WORK

Automated Machine Learning. With so many hyperparameters to tune and ever-increasing model sizes, developers have started relying on Automated ML (AutoML) to explore the space of possible neural architectures and hyperparameters. AutoML tasks typically do not share the unique and stringent set of constraints that arise in datacenter networking, which ultimately allows Homunculus to handle, otherwise untenable search spaces. Computer networks have several requirements that must be considered jointly with the neural architecture search. Multi-objective optimization is crucial because real-world applications often rely on a trade-off between several objectives [52, 68]. Derivatives are also often unavailable because of discrete input variables and noisy black-box functions, or are impractical to compute [34]. All of these features are standard requirements in datacenter networking but are rarely exposed in AutoML frameworks, which usually focus on the accuracy of the ML model alone. In general, there have been only a few attempts to date at AutoML-driven systems in the networking space (such as using an AutoML approach for traffic analysis [41]). Our work focuses on refining network representations and fits well with systems, such as Homunculus, that can consume and utilize this data.

Code Generation for Data Planes. Recent works have explored code generation for data and control planes. Mantis [101] generates data plane and switch-local control plane software optimized for reactivity, allowing a faster control plane to react to congestion conditions in 10s of μ-seconds. Lucid [82] generates efficient P4 code for the data plane via a high-level language and can also specify control-plane functionality. These control-plane functions are mapped to the data plane, allowing faster control decisions compared to the switch-local control plane. While these efforts are primarily meant for data-plane and control-plane code generation to enable either reactivity or data-plane control, our approach is meant for generating efficient ML pipelines for data-plane platforms, which fit within the available resources and constraints.

Configurable Hardware. Plasticine [74], a Coarse-Grained Reconfigurable Array (CGRA), provides a grid of compute and memory units, which can be reconfigured according to the dataflow graph of an application. This SIMD grid of compute and memory units provides the necessary flexibility to implement neural networks in the data plane that are not realizable in the VLIW MAT pipeline. Taurus' MapReduce block is based on this architecture, which has been tailored to support streaming data on switches and NICs [85]. The Reconfigurable Match Table (RMT) is an example of a reconfigurable architecture that has been specifically designed while considering the domain characteristics of the network data plane [18]. While the two platforms are structurally different (SIMD vs VLIW), Homunculus is agnostic to these architectural variations and only needs to query the resource utilization and performance metrics to generate platform-specific code for either of them.

8 CONCLUSION

We presented Homunculus, a compiler for building efficient data-plane ML pipelines from high-level directives—giving network operators an easy way to inform the compiler of their needs rather than having to spend time tuning hyperparameters and trying to manage the various demands of the network data-planes and topology. It is clear that human operators are no longer the best choice for deploying algorithms in a datacenter network. The environment is far too complex for humans to manage. Instead, they should describe their intent and let automated systems handle the rest. With

Homunculus, we move one step closer to this era of declarative, intent-based networking.

ACKNOWLEDGMENTS

We thank our anonymous shepherd and reviewers for their valuable feedback that helped improve the quality of this paper. We also thank Yaqi Zhang and Alexander Rucker for their assistance with the various backend components used in Homunculus (like PIR and Tungsten), as well as Nathan Zhang for his insights on fast FPGA recompilation. This research was supported by National Science Foundation (NSF) awards (CCF-1937301 and CNS-2211381). Support also came in part from affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Intel, Microsoft, NEC, SAP, Teradata, and VMware. Luigi Nardi was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

A BO-GUIDED DSE FORMULATION

Here we provide a formal definition of the optimization process, mentioned in §3.2.3, and make it tractable by treating models and algorithms as opaque functions (i.e., black boxes). In black-box optimization, the process is unaware of the internals of the models and, therefore, can swap different ML algorithms and configurations without changing the optimization method.

The black-box problem optimizes a (possibly noisy) function $f : \mathbb{X} \rightarrow \mathbb{R}$ over a domain of interest $\mathbb{X}$ that includes lower and upper bounds on the problem variables. The variables defining $\mathbb{X}$ can be real (continuous), integer, ordinal, or categorical, as in HyperMapper [68]. The objective function f for Homunculus is to maximize model performance (say F1 score) under the constraints (c_i) of network performance (latency, throughput) and resource consumption (such as LUTs, BRAM, CU, MU, and MATs). x^* is the optimal ML model configuration (say, the number of layers and neurons per layer for a DNN) that maximizes this objective while respecting the constraints.

Homunculus assumes that the function f is in general expensive to evaluate, e.g., it may take minutes or hours to evaluate one design $\mathbf{x} \in \mathbb{X}$ (even using a software model such as SARA [103] for Taurus), and that the derivatives of f are in general not available. The second assumption implies that off-the-shelf gradient-based optimizers cannot be used to solve the optimization problem; the function is called *black box* because we cannot access other information than the output y of the function f (F1 score, resource utilization, latency/throughput) given an input value $\mathbf{x}$ (ML model configuration). Bayesian optimization (BO) achieves this by building a probabilistic surrogate model on f based on the set of evaluated points. At each iteration, a new point is selected and evaluated using the surrogate model, and this model is updated to include the new point (x_{t+1}, y_{t+1}).

B ARTIFACT APPENDIX

Abstract. The artifact contains the source code for the titular Homunculus compiler, a backend for the Taurus ASIC switch architecture, as well as three representative applications. We used these applications to demonstrate the core results of our paper, i.e., how Homunculus-generated models outperform or match the hand-tuned baseline versions. We include applications for anomaly detection, traffic classification, and botnet detection, as shown in Table 2. Homunculus also generates the appropriate hardware code for each of these applications to run on a Taurus switch architecture. The code repositories accompany detailed instructions on how to build and run the applications using Homunculus.

Scope. This artifact provides three main contributions: the Homunculus compiler, a Taurus backend, and three representative applications to test the compiler with. The objective is to provide users with examples of how to program in-network ML models using the Alchemy frontend (and compiled with Homunculus), and compare their performance with hand-tuned baseline counterparts. The compiler generates these models as Spatial hardware description language (HDL) for the Taurus ASIC backend.

Contents. The source code and documentation for building and running Homunculus with the Taurus ASIC backend with the accompanying applications are published as a collection of GitLab repositories here: https://gitlab.com/dataplane-ai/homunculus/artifact-asplos23. We also provide a Docker image on figshare with a pre-built version of Homunculus that users can run without worrying about setup (https://doi.org/10.6084/m9.figshare.22081901.v1). The primary components are as follows:

- *Homunculus Compiler:* This repository contains the source code and instructions for building the Homunculus Compiler. It can be found at https://gitlab.com/dataplane-ai/homunculus/compiler
- *Taurus ASIC Backend:* This repository contains the source code and instructions for building the Taurus ASIC Backend to accompany Homunculus. It can be found at https://gitlab.com/dataplane-ai/homunculus/backends/taurus-asic
- *Applications and Datasets:* In this repository, we share the Alchemy source code and dataset for the three applications provided with Homunculus. For each application, Homunculus will generate Spatial HDL code for an optimized DNN model, which runs on the Taurus ASIC. The applications can be found at https://gitlab.com/dataplane-ai/homunculus/applications, and the datasets for the applications are available at https://gitlab.com/dataplane-ai/homunculus/datasets. Included applications are: anomaly detection (AD), traffic classification (TC), and botnet detection (BD).

Hosting. The source code and build instructions are publicly available on GitLab[3] and a pre-built Docker image is a available on figshare.[4]

Dependencies. The artifact relies on several third-party software tools. If using the Docker image available on figshare, these software tools are already installed. All dependencies are specified in the artifact's build files, but the primary dependencies are listed below:

- Spatial Hardware Description Language
- HyperMapper Black-Box Optimization Tool
- Plasticine Intermediate Representation (PIR)
- Plastiroute Router and Placement Analyzer
- Plastisim Cycle Approximate Simulator
- Tungsten Plasticine Simulator
- TensorFlow Platform
- Docker (if using the image on figshare)

[3]https://gitlab.com/dataplane-ai/homunculus/artifact-asplos23
[4]https://doi.org/10.6084/m9.figshare.22081901.v1

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX OSDI*.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM*.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM SIGCOMM*.

[4] Sivaram Ambikasaran, Daniel Foreman-Mackey, Leslie Greengard, David W Hogg, and Michael O'Neil. 2015. Fast Direct Methods for Gaussian Processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38, 2 (2015), 252–265.

[5] Nahla Ben Amor, Salem Benferhat, and Zied Elouedi. 2004. Naive Bayes vs Decision Trees in Intrusion Detection Systems. In *ACM Symposium on Applied Computing (2004)*.

[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. 303–316.

[7] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX NSDI (2020)*.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC*.

[9] Jarrod Bakker, Bryan Ng, Winston K.G. Seah, and Adrian Pekar. 2019. Traffic Classification with Machine Learning in a Live Network. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*.

[10] Pierre Baldi and Peter J Sadowski. 2013. Understanding Dropout. *Advances in Neural Information Processing Systems* 26 (2013), 2814–2822.

[11] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando M. V. Ramos, and André Madeira. 2021. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *NDSS*.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *ACM IMC*.

[13] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU Math Expression Compiler. In *SciPy*.

[14] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *International Conference on Machine Learning*.

[15] Ekaba Bisong. 2019. Google AutoML: Cloud Vision. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 581–598.

[16] Bruno Bodin, Luigi Nardi, M. Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan, Björn Franke, Paul H.J. Kelly, and Michael O'Boyle. 2016. Integrating Algorithmic Parameters into Benchmarking and Design Space Exploration in 3D Scene Understanding. In *ACM PACT*.

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

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*.

[19] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M. Caicedo. 2018. A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities. *Journal of Internet Services and Applications* (2018).

[20] Leo Breiman. 2001. Random Forests. *Machine learning* 45, 1 (2001), 5–32.

[21] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-scale Acceleration Architecture. In *IEEE MICRO*.

[22] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. AuTO: Scaling Deep Reinforcement Learning for Datacenter-scale Automatic Traffic Optimization. In *ACM SIGCOMM*.

[23] François Chollet et al. 2018. Keras: The Python Deep Learning Library. *Astrophysics Source Code Library* (2018).

[24] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro* (2018).

[25] Carla Currin, Toby Mitchell, Max Morris, and Don Ylvisaker. 1988. *A Bayesian Approach to the Design and Analysis of Computer Experiments*. Technical Report. Oak Ridge National Laboratory, TN (USA).

[26] L Dhanabal and SP Shantharajah. 2015. A Study on NSL-KDD Dataset for Intrusion Detection System Based on Classification Algorithms. *International Journal of Advanced Research in Computer and Communication Engineering* 4, 6 (2015), 446–452.

[27] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-Architecting Congestion Control for Consistent High Performance. In *USENIX NSDI*.

[28] Adel Ejjeh, Vikram Adve, and Rob A Rutenbar. 2020. Studying the Potential of Automatic Optimizations in the Intel FPGA SDK for OpenCL. In *ACM/SIGDA FPGA*.

[29] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A Scriptable High-speed Packet Generator. In *ACM IMC*.

[30] Alice Este, Francesco Gringoli, and Luca Salgarelli. 2009. Support Vector Machines for TCP traffic classification. *Computer Networks* (2009).

[31] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *Automated Machine Learning*.

[32] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI*.

[33] Jacob R Gardner, Matt J Kusner, Zhixiang Eddie Xu, Kilian Q Weinberger, and John P Cunningham. 2014. Bayesian Optimization with Inequality Constraints. In *ICML*.

[34] Michael A Gelbart, Jasper Snoek, and Ryan P Adams. 2014. Bayesian Optimization with Unknown Constraints. *arXiv preprint arXiv:1403.5607* (2014).

[35] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2019. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *USENIX NSDI*.

[36] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2022. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–12.

[37] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review* (2008).

[38] B. Hariri and N. Sadati. 2007. NN-RED: An AQM Mechanism Based on Neural Networks. *Electronics Letters* (2007). https://doi.org/10.1049/el:20071791

[39] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A Survey of the State-of-the-Art. *Knowledge-Based Systems* (2021).

[40] M.A. Hearst, S.T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their Applications* (1998).

[41] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. 2021. New Directions in Automated Traffic Analysis. In *ACM SIGSAC CCS*.

[42] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *HotOS*.

[43] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated Machine Learning: Methods, Systems, Challenges*. Springer Nature.

[44] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *ACM/SIGDA FPGA*.

[45] Intel. last accessed: 06/10/2022. Infrastructure Processing Unit (Intel IPU) and SmartNICs. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[46] Intel. last accessed: 06/10/2022. Tofino: P4-programmable Ethernet Switch ASIC that Delivers Better Performance at Lower Power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[47] Intel. last accessed: 06/10/2022. Tofino2: Second-generation P4-programmable Ethernet Switch ASIC that Continues to Deliver Programmability without Compromise. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.
[48] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *ICML*.
[49] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *ACM SIGKDD*.
[50] Radhakrishna Kamath and Krishna M Sivalingam. 2021. Machine Learning Based Flow Classification in DCNs Using P4 Switches. In *IEEE ICCCN*.
[51] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR*.
[52] Nicolas Knudde, Joachim van der Herten, Tom Dhaene, and Ivo Couckuyt. 2017. GPflowOpt: A Bayesian Optimization Library Using TensorFlow. *arXiv preprint arXiv:1711.03845* (2017).
[53] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *ACM SIGPLAN PLDI*.
[54] Angelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. 2019. Offloading Real-time DDoS Attack Detection to Programmable Data Planes. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*.
[55] Guanyu Li, Menghao Zhang, Shicheng Wang, Chang Liu, Mingwei Xu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, and Jianping Wu. 2021. Enabling Performant, Flexible and Cost-Efficient DDoS Defense With Programmable Switches. *IEEE/ACM Transactions on Networking* (2021).
[56] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*.
[57] Yingqiu Liu, Wei Li, and Yunchun Li. 2007. Network Traffic Classification Using K-means Clustering. In *IMSCCS*.
[58] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security*.
[59] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *ACM HotNets*.
[60] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM*.
[61] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*.
[62] Wes McKinney et al. 2011. pandas: A Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing* 14, 9 (2011), 1–9.
[63] Tahir Mehmood and Helmi B Md Rais. 2015. SVM for Network Anomaly Detection using ACO Feature Subset. In *IEEE iSMSC*.
[64] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. 1978. The Application of Bayesian Methods for Seeking the Extremum. *Toward Global Optimization* 2, 117-129 (1978), 2.
[65] John Moody. 1994. Prediction Risk and Architecture Selection for Neural Networks. In *From Statistics to Neural Networks*. Springer, 147–165.
[66] Pratik Narang, Subhajit Ray, Chittaranjan Hota, and Venkat Venkatakrishnan. 2014. Peershark: Detecting Peer-to-Peer Botnets by Tracking Conversations. In *IEEE Security and Privacy Workshop*.
[67] Luigi Nardi, Bruno Bodin, Sajad Saeedi, Emanuele Vespa, Andrew J Davison, and Paul HJ Kelly. 2017. Algorithmic Performance-accuracy Trade-off in 3D Vision Applications using Hypermapper. In *IEEE IPDPSW*.
[68] Luigi Nardi, David Koeplinger, and Kunle Olukotun. 2019. Practical Design Space Exploration. In *IEEE MASCOTS*.
[69] Nvidia. last accessed: 06/10/2022. Bluefield Data Processing Units (DPUs). https://www.nvidia.com/en-us/networking/products/data-processing-unit/.
[70] ONF. last accessed: 06/10/2022. ONOS: Open Network Operating System (ONOS). https://opennetworking.org/onos/.
[71] ONF. last accessed: 06/10/2022. Stratum: Enabling the era of next generation SDN. https://opennetworking.org/stratum/.
[72] Biswajit Paria, Kirthevasan Kandasamy, and Barnabás Póczos. 2019. A Flexible Framework for Multi-Objective Bayesian Optimization using Random Scalarizations. In *UAI*.
[73] Pascal Poupart, Zhitang Chen, Priyank Jaini, Fred Fung, Hengky Susanto, Yanhui Geng, Li Chen, Kai Chen, and Hao Jin. 2016. Online Flow Size Prediction for Improved Network Routing. In *IEEE ICNP*.
[74] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture for Parallel Patterns. In *ACM/IEEE ISCA*.
[75] Andrew Putnam. 2017. FPGAs in the Datacenter: Combining the Worlds of Hardware and Software Development. In *GLSVLSI*.
[76] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ACM/IEEE ISCA*.
[77] Babak Rahbarinia, Roberto Perdisci, Andrea Lanzi, and Kang Li. 2013. PeerRush: Mining for Unwanted P2P Traffic. In *DIMVA*.
[78] Sajad Saeedi, Luigi Nardi, Edward Johns, Bruno Bodin, Paul HJ Kelly, and Andrew J Davison. 2017. Application-oriented Design Space Exploration for SLAM Algorithms. In *IEEE ICRA*.
[79] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2018. Can the Network Be the AI Accelerator?. In *ACM NetCompute*.
[80] Giuseppe Siracusano and Roberto Bifulco. 2018. In-network Neural Networks. *arXiv preprint arXiv:1801.05731* (2018).
[81] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. 2022. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *USENIX NSDI*.
[82] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *ACM SIGCOMM*.
[83] Le Song, Santosh Vempala, John Wilmes, and Bo Xie. 2017. On the Complexity of Learning Neural Networks. *arXiv preprint arXiv:1707.04615* (2017).
[84] Artur Souza, Luigi Nardi, Leonardo B Oliveira, Kunle Olukotun, Marius Lindauer, and Frank Hutter. 2021. Bayesian Optimization with a Prior for the Optimum. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*.
[85] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: A Data Plane Architecture for per-Packet ML. In *ACM ASPLOS*.
[86] Tuan A Tang, Lotfi Mhamdi, Des McLernon, Syed Ali Raza Zaidi, and Mounir Ghogho. 2016. Deep Learning Approach for Network Intrusion Detection in Software Defined Networking. In *IEEE WINCOM*.
[87] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. 2009. A Detailed Analysis of the KDD CUP 99 Dataset. In *IEEE CISDA*.
[88] Lisa Torrey and Jude Shavlik. 2010. Transfer Learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI global, 242–264.
[89] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is Advance Knowledge of Flow Sizes a Plausible Assumption?. In *USENIX NSDI*.
[90] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A Survey of Transfer Learning. *Journal of Big data* 3, 1 (2016), 1–40.
[91] Keith Winstein and Hari Balakrishnan. 2013. TCP ex machina: Computer-generated Congestion Control. In *ACM SIGCOMM CCR*.
[92] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. 2022. PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development. In *ACM ASPLOS*.
[93] Xilinx. last accessed: 06/10/2022. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.
[94] Xilinx. last accessed: 06/10/2022. Running Multiple Implementation Strategies for Timing Closure. https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Running-Multiple-Implementation-Strategies-for-Timing-Closure.
[95] Xilinx. last accessed: 06/10/2022. UltraScale+ Integrated 100G Ethernet Subsystem. https://www.xilinx.com/products/intellectual-property/cmac_usplus.html.
[96] Xilinx. last accessed: 06/10/2022. Vivado. https://www.xilinx.com/products/design-tools/vivado.html.
[97] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning? Toward In-network Classification. In *ACM HotNets*.
[98] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: A Randomized Experiment in Video Streaming. In *USENIX NSDI*.
[99] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: The Training Ground for Internet Congestion-Control Research. In *USENIX ATC*.
[100] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. 2021. ACC: Automatic ECN Tuning for High-Speed Datacenter Networks. In *ACM SIGCOMM*.
[101] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive Programmable Switches. In *ACM SIGCOMM*.
[102] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-Resolution Measurement of Data Center Microbursts. In *ACM IMC*.

[103] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *ACM/IEEE ISCA*.
[104] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *ACM SIGCOMM*.
[105] Lucas Zimmer, Marius Lindauer, and Frank Hutter. 2021. Auto-Pytorch: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).