Automatic Optimization of Software Data Planes

Sebastiano Miano
Politecnico di Torino, IT
sebastiano.miano@polito.it

Gábor Rétvári
Budapest University of Technology and Economics, HU
retvari@tmit.bme.hu

Fulvio Risso
Politecnico di Torino, IT
fulvio.risso@polito.it

Andrew W. Moore
University of Cambridge, UK
andrew.moore@cl.cam.ac.uk

Gianni Antichi
Queen Mary University London, UK
g.antichi@qmul.ac.uk

ABSTRACT

In this poster, we make a case for a compiler that continuously optimizes software data planes at run-time. Furthermore, we propose its architecture and discuss the challenges associated with its design.

KEYWORDS

Network Functions, Data Plane Compilation, eBPF, DPDK

1 INTRODUCTION

Implementing network data planes entirely in software is a widely recognised possibility, e.g., Linux Foundation's Open vSwitch, Facebook's Katran load balancer. Traditional approaches to design and develop those solutions are based on static compilation: the compiler receives as input a description of the forwarding plane semantics and outputs a binary code that is agnostic to both its configuration, i.e., entries in the match/action tables, and input traffic patterns.

Improving the performance of generic software programs at runtime is possible with compilers, e.g., GCC, LLVM, that support Profile Guided Optimizations (PGO) and Feedback Directed Optimizations (FDO). This is the case, for example, of Google's AutoFDO [2] or Facebook's Bolt [5] that take advantage of local data conditions to *recompile on demand* portions of the original program based on the runtime execution profile. Unfortunately, those solutions have been conceived to optimize only generic computer programs and do not easily address the peculiar characteristics of *packet processing programs*. To demonstrate this, we tested two different applications: one built on top of DPDK, the other on linux eBPF/XDP. The former, i.e., *flow-classify*[1], is a five-tuple classifier that matches incoming packets against a configurable set of rules and outputs them to different destination ports. The latter is Katran[2], Facebook's open-source layer-four load balancer. We connected two servers back-to-back through Intel XL710 40GbE NICs. We used one server to generate high-throughput traffic with the DPDK application *pktgen-dpdk*[3], while in the other we installed the data plane program under test, pinned to a single core. We evaluated *flow-classify* with 50 statically configured rules matching on the 5-tuple and with a input traffic distributed among all of them with an high skewness: 5 elephant flows sending the 95% of traffic. We

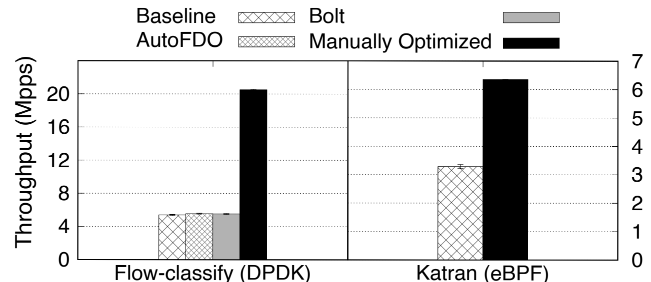

Figure 1: Improvements with manual optimizations applied to both DPDK and eBPF based data planes.

optimized at runtime this application using two different state-of-the-art FDO-based tools, i.e., AutoFDO and Bolt. We then manually tweaked the original application code by exploiting the knowledge of the configured rules and the input traffic. Specifically, we embedded the lookup results for the most-matched entries directly into the code and we removed all the unnecessary branches and instructions: if no rule matches a specific header field, then its parsing can be avoided. In Figure 1, we compare the performances we obtained with this handcrafted optimizer, i.e., *Manually Optimized*, and with state-of-the-art FDO tools. We repeated a similar test with the Katran eBPF-based load balancer. Here, we configured five different Virtual IPs, each of them with a hundred of backend servers and we exploited the knowledge of the runtime configuration as well as the input traffic patterns to manually optimize its implementation. Specifically, we embedded into the code the mapping for the most common flows and we changed the LPM table into a hash-based lookup table that performs better for the type of input traffic we generated. On top of this, we applied some of the standard packet processing program optimization tricks, see e.g., [1].

In Figure 1, we show a performance improvement of around 100%[4]. In both cases, the gain strongly depends on the input traffic patterns: the modifications cannot be valid for every scenario. Nevertheless those tests provide a qualitative assessment on how much it is possible to optimize a network function. Bolt and AutoFDO did not sensibly improve the performance of `flow-classify` because of the different nature of generic programs and packet processing functions. The performance of the latter indeed depends on metrics (e.g., number of table accesses, packet burst size, table's size and algorithm) that are not taken into account by such tools.

[1]https://doc.dpdk.org/guides/sample_app_ug/flow_classify.html
[2]https://engineering.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/
[3]https://github.com/Pktgen/Pktgen-DPDK/

[4]We were unable to run the FDO tools on eBPF-based application, since it is not possible to correctly isolate the perf profile for a single eBPF application.

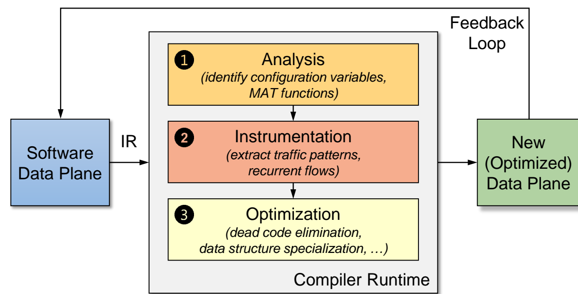

Figure 2: The compiler pipeline.

2 RUNTIME DATA PLANE COMPILATION

The performance of software data planes depends on a number of factors: (1) network configuration, i.e., CPU cycles can be saved by pruning (at compilation time) instructions that result unreachable given the current configuration pattern [3], e.g., an application is designed to handle VLAN traffic may not actually enforce any VLAN. (2) run time table content, i.e., the lookup process can be improved by selecting the most appropriate data structure to store current data [4, 6]. (3) traffic patterns, i.e., by creating a fast path for the most frequently accessed entries [1].

Our idea. We propose a new runtime compiler for data plane programs that takes into account all the above-mentioned aspects. Its execution can be triggered at given time slots, or as a consequence of an external event which may invalidate the optimizations applied within the previous cycle, i.e., an update of some match-action table (MAT) entries. Figure 2 shows its building blocks. The input is the Intermediate Representation (IR) of the application code including the modifications applied in the last optimization cycle. Working at the IR level makes the compiler agnostic to the specific language being used to write the software data plane, thus making our solution technology independent, e.g., DPDK, eBPF. The compiler first *analyzes* the IR to identify specific patterns e.g., retrieve configuration variables, operations on match-action tables (MAT). The output of this stage is an IR code marked with debug information that is stored into the compiler's internal data structures. This is fed into the second stage that, starting from the received debug data, *instruments* the code by adding more instructions that can help in characterizing input traffic patterns.

By doing so, it is possible to retrieve the most frequent flows on a particular code branch or other runtime information such as the content of a specific MAT. The final step is to perform code *optimization*; Here, the results from the previous two stages are taken into account: for example, if the analysis stage spots an unused code branch in the current data plane configuration then this branch can be safely removed temporarily; if the instrumentation installed in the previous optimization cycle has spotted a hot code path, then this information can be used to optimize that piece of code by, for example, hard-coding a specific flow processing and thus reducing the number of MAT accesses (Table 1 shows a list of optimizations). Once this stage is completed, the compiler emits the optimized IR code and calls the compiler's back-end to generate the final executable. The main challenges to be faced are the following:

Challenge #1: Identify relevant logic. Writing a packet processing program in software can be done in a number of different ways.

Pass Name	Description
Dead Code Elimination	Prunes instructions unreachable within the runtime config.
Data Structure Specialization	Changes MAT layout and size to better fit runtime config.
Cached Computation	Caches the most accessed entries within the code itself.
Fast Path Creation	Aggregates multiple MATs.

Table 1: Example of compiler optimization passes.

The compiler shall be able to spot and extract only relevant patterns into the code. This is possible by having a clear separation between stateless and stateful code so that it is easier to find MAT lookups and associate them with the forwarding behavior of the data plane. eBPF, Vigor and Click-NF follow this approach by design.

Challenge #2: The cost of instrumentation. Adding additional logic to characterize input traffic patterns can negatively affect performance, to the point that that it may nullify the effect of the optimization. This overhead can be reduced by instrumenting only performance-critical parts in the code.

Challenge #3: Preserve original data plane semantic. The optimizations shall not modify the semantic of the original application. The compiler should introduce safety measures (e.g., *guards*) to restore to the original data plane code when the optimization is not valid anymore.

Challenge #4: Harmless pipeline swap. At each iteration, the compiler creates a new optimized version of the original program. This shall substitute the program currently running avoiding packet processing inconsistencies or losses. The compiler can use a combination of *indirect jumps* and *relocations* to load and substitute the newly generated code.

Acknowledgments. We thank the anonymous reviewers. This work is sponsored by the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL project (EP/P025374/1), the EPSRC New Investigator Award NEAT (EP/T007206/1) and the European Commission (project ASTRID, Grant Agreement no.786922). Gábor Rétvári is also with the MTA-BME Network Softwarization Research Group and the MTA-BME Information Systems Research Group.

REFERENCES

[1] Omid Alipourfard and Minlan Yu. 2018. Decoupling Algorithms and Optimizations in Network Functions. In *Workshop on Hot Topics in Networks (HotNets)*. ACM.
[2] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Symposium on Code Generation and Optimization (CGO)*. IEEE/ACM.
[3] Bangwen Deng, Wenfei Wu, and Linhai Song. 2020. Redundant Logic Elimination in Network Functions. In *Proceedings of the Symposium on SDN Research (SOSR)*. ACM.
[4] Molnár, László and Pongrácz, Gergely and Enyedi, Gábor and Kis, Zoltán Lajos and Csikor, Levente and Juhász, Ferenc and Kőrösi, Attila and Rétvári, Gábor. 2016. Dataplane Specialization for High-Performance OpenFlow Software Switching. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[5] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Symposium on Code Generation and Optimization (CGO)*. IEEE/ACM.
[6] Rétvári, Gábor and Molnár, László and Enyedi, Gábor and Pongrácz, Gergely. 2017. Dynamic Compilation and Optimization of Packet Processing Programs. In *Workshop on Networking and Programming Languages (NetPL)*. ACM.