Dynamic Compilation and Optimization of Packet Processing Programs

Gábor Rétvári*, László Molnár†, Gábor Enyedi†, Gergely Pongrácz†
*MTA-BME Information Systems Research Group, BME-TMIT
†TrafficLab, Ericsson Research, Hungary

ABSTRACT

Data plane compilation is a transformation from a high-level description of the intended packet processing functionality to the underlying data plane architecture. Compilation in this setting is usually done *statically*, i.e., the input of the compiler is a fixed description of the forwarding plane *semantics* and the output is code that can accommodate any packet processing *behavior* set by the controller at runtime. Below we advocate a *dynamic approach* to data plane compilation instead, where not just the semantics but the intended behavior is also also input to the compiler. We uncover a handful of runtime optimization opportunities that can be leveraged to improve the performance of custom-compiled datapaths beyond what is possible in a static setting.

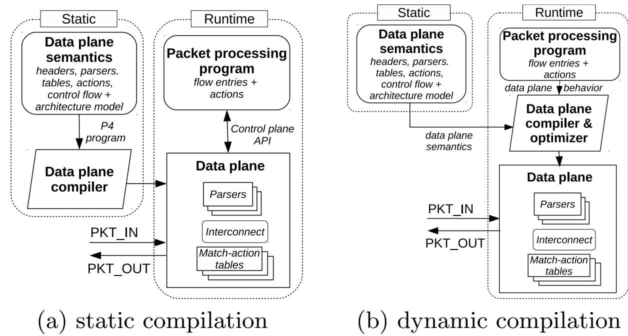
Keywords

data plane programming, template-based code generation, runtime optimization

1. INTRODUCTION

Recent innovations in data plane technologies, from flexible switch ASICs [1,7] and programmable NPUs [3] to versatile software packet forwarding toolkits [4, 5], have one thing in common: reconfigurability. These reconfigurable data plane architectures expose a set of header-parsing, table-matching, and action primitives to the controller so that the required packet processing behavior can be constructed from these elementary templates mid-deployment.

Static data plane compilers are the most prevalent approach to programming such reconfigurable pipelines (POF [8], P4 [2]). Given an abstract, declarative description of the data plane *semantics* (header fields, parsers, matching tables, action primitives, and control flow graph), a static compiler outputs an architecture-specific switch runtime, coupled with a northbound API, that can be readily deployed on the target (see Fig. 1a). Since the actual packet processing *behavior*, that is, the contents of the match-action tables, metadata, etc., is made available by the controller only at runtime, a statically compiled datapath must perform efficiently with respect to *any* packet processing program it may exe-



(a) static compilation (b) dynamic compilation

Figure 1: Static vs. dynamic data plane compilation: the static compiler knows only the forwarding plane semantics, while a dynamic compiler knows both the semantics *and* the behavior for the switch.

cute, subject to the forwarding plane semantics as specified in the compiler input. A *dynamic* approach to data plane compilation, however, would allow to sidestep this *genericity* of statically generated codes, opening up a wealth of new optimization possibilities (Fig. 1b).

Imagine a P4 compiler that, besides the target architecture specification and the basic switch functionality, would also be aware of the actual contents for the generated pipeline, like the flows and actions to be installed into the match-action tables at runtime. Such a dynamic P4 compiler would repeatedly rebuild and re-optimize the data plane every time the controller alters some aspect of the switch's program, like adding a new flow to a table or modifying the action associated with an existing one. This would make it possible to compile a custom *switch runtime that would be optimal to the data plane semantics and the packet processing behavior at the same time* (see also Cilium/eBPF [4])).

In fact, we have started to experiment with such a dynamic data plane compiler on top of ESwitch, a high-performance dynamic data plane compiler currently hard-coded for the OpenFlow semantics [6]. In this memo, we sketch some interesting runtime optimization opportunities that we expect such an ESwitch-based dynamic P4 compiler to uncover.

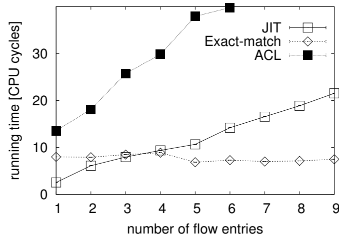


Figure 2: Lookup time on an exact-match table, a SW packet classifier data-structure (ACL), and a just-in-time compiled table (JIT), as the function of table size.

2. DYNAMIC DATA PLANE COMPILATION

Of course, a dynamic P4 compiler must be very fast in order to keep track with data plane modifications. We argue that the template-based code generation framework we picked for ESwitch would be an ideal choice for a dynamic P4 data plane compiler.

Dynamic template-based code generation. Switch pipelines, simple non-recursive programs executing over a match-action abstract machine, are all about templates [1, 6]. Packet parsing is essentially taking some value from a fixed offset and branching, match tables use prefab exact, LPM, or wildcard classifier primitives, and actions are just a sequence of packet processing primitives that are to be executed on a match. In this context, template-based code generation lends itself as an appealing dynamic compilation and optimization technique, for the speed, the amenability to hardware offloading (primitives are easy to map to HW resources), and the potential to eliminate unused code.

In a real-life deployment at any particular instance of time a lot of switch features go unused, like core switches may not be terminating VXLAN tunnels (in which case parsing VXLAN headers is unnecessary) or would run with empty ACL tables (in which case time-consuming wildcard matches would unnecessarily burden the data plane). Template-based code generation would look into the actual match-action tables and would optimize away all parsing/matching/action templates that are currently unused; with a static compiler the control plane would need to maintain an accurate, per-switch inventory of features currently compiled into the data plane to achieve the same level of code specialization. In other words, *with dynamic data plane compiler we indeed only pay for the features we actually use.*

Dynamic match-table optimization. A look-up table is an abstract finite map that, given a key, produces as a result an entry that most specifically matches the key, along with the corresponding action that is to be executed. Then, it is left for the data plane compiler to choose the best implementation for a given look-up table. In a static setting the implementation must match the most complex header field's semantics that participates in the look-up, e.g., if a table contains a `ternary` field (a wildcard) then it must be implemented in a TCAM or a software-based packet classifier (ACL).

Dynamic code generation, on the other hand, would allow to promote a table to a more efficient implementation if the contents made this possible. For instance, we see many ACL tables that match on only the destination TCP/UDP port, a wildcard match that is very easy to convert into a very fast and cheap exact-matching look-up semantics. In a more complicated case, a complex match table could be converted into a hierarchy of simple (say, exact-matching) and thusly very fast look-up tables [6], but note again that such conversion necessarily depends on the actual *contents* of the tables that a dynamic compiler does have, whereas a static one does not have, at hand. Our early stage experimentation has indicated that *dynamic look-up table optimization can yield orders-of-magnitude higher performance* over realistic data plane programs (see Fig. 2).

Just-in-time (JIT) compilation. Just-in-time compilation is the ultimate level of code specialization: hot code paths are turned into actual machine code that can be directly executed on the target. This allows to customize generic code paths, inline constants, unroll loops, and hard-code jump pointers, in order to minimize the working set size, to eliminate costly branch instructions, to redirect data-cache load to the instruction cache, etc. A dynamic data plane compiler can take full advantage of just-in-time compilation (in contrast to a static one); especially actions and action sets lend themselves as the most plausible candidates. However, *we have found that just-in-time compiling smaller match tables* (e.g., per-ingress-port tables, per-tenant logical datapaths, or small ACLs) *also yields substantial performance benefits* (see again Fig. 2).

In the full version of this memo, we shall look into the dynamic compilation and optimization strategies we used for building ESwitch and discuss how these techniques transform to the more general context of P4.

3. REFERENCES

[1] Bosshart, P., et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM* (2013), pp. 99–110.

[2] Bosshart, P., et al. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (2014), 87–95.

[3] Cavium. Cavium's XPliant Ethernet switch supports the emerging open ecosystems.

[4] Graf, T. Next generation of programmable datapath. In *OVS Conference* (2016).

[5] Intel. Data Plane Development Kit. `http://dpdk.org`.

[6] Molnár, L., Pongrácz, G., Enyedi, G., Kis, Z. L., Csikor, L., Juhász, F., Kőrösi, A., and Rétvári, G. Dataplane specialization for high performance OpenFlow software switching. In *SIGCOMM* (2016).

[7] Ozdag, R. Intel Ethernet Switch FM6000 Series – Software Defined Networking, 2012.

[8] Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *HotSDN* (2013), pp. 127–132.