Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

Are Graph Databases Fast Enough for Static P4 Code Analysis?*

Dániel Lukács[a], Gergely Pongrácz[b], Máté Tejfel[c]

[a]Faculty of Informatics
Eötvös Loránd University, Budapest, Hungary
dlukacs@inf.elte.hu
ORCID: 0000-0001-9738-1134

[b]Ericsson Hungary Ltd., Budapest, Hungary
Gergely.Pongracz@ericsson.com
ORCID: 0000-0002-5115-9973

[c]3in Research Group, Eötvös Loránd University, Martonvásár, Hungary
matej@inf.elte.hu
ORCID: 0000-0002-5115-9973

Abstract

P4 is a new protocol-programming language for smart switches and virtual network functions linked up in vast software defined networks (SDNs). Our current work in progress is focused on analysing the execution cost of P4 programs using hierarchical control flow graphs (CFGs). We found that cost analysis of P4 requires storing and working with diverse information about various execution environments.

In theory, versatile graph backends, such as graph databases (GDBs) could provide a unifying representation in which we can store, access and traverse the CFGs and the environment information together. On the other hand, analysis efficiency is a requirement both for large-scale testing and end user application. Unfortunately, we cannot conclusively assess – without trying it out in practice – whether GDB disk and memory reads will ruin our performance or not.

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

In this work, we briefly detail how we realized P4 control flow analysis based on GDBs, and present our measurements to conclude if the overhead inherent in GDBs is justified for our purposes.

Keywords: static analysis, P4, cost analysis, graph database, software defined networking

1. Introduction

Softwarization of modern networks introduces new challenges for tools supporting agile development and analysis of new protocols. P4 is a relatively new programming language targeting network switches [2]. As the overall functioning of the network requires switches to process packets with an acceptable latency, it is important for switches, and P4 programs (*protocols*) executed by the switches, to execute as fast as possible. The overall goal of our research is to develop a static cost analysis framework for P4 to assist network engineers in estimating the execution cost of the P4 program during development, well before deployment.

To perform cost analysis, we take as input the P4 program code, and an appropriate model of the target environment. Then, we transform this information into an abstract cost formula. Missing runtime information will be represented in the formula as variables. In turn, the derived cost formula can be used in many ways: the cost of a P4 program can be inferred given an existing hardware configuration, or inversely, we can infer the set of configurations that satisfy a given cost constraint.

This paper has two contributions. First, we reformulated our earlier cost analysis algorithm (see below) as a graph query to enable the reuse of established graph database (GDB) services (Section 3). Second, we implemented this query, along with our earlier algorithm in Java, and performed experiments to measure the performance of the two cost analyses on real-world P4 programs (Section 4). Here, our intention is to establish the feasibility and limits of performing P4 cost analysis real-time, i.e. during development, or possibly, at runtime of P4 programs. The results suggest that – despite the overhead – graph databases are viable platforms for implementing cost analysis.

1.1. Motivation

The goals of this paper are motivated by practical – instead of theoretical – concerns. Modern software engineering involves both transactional and analytical operations: multiple developers armed with their IDEs asynchronously work together on a large code base, which is cyclically inspected (augmented with static analysis and verification tools) for testing and code review purposes. In hope that we can innovate industrial-scale software – not "just" an algorithm –, we intend to assess whether we can reuse common database features (instead of reinventing the wheel) without significant losses in runtime efficiency. Among these are scalability (through ease of multi-core or multi-machine parallelization), multi-user

concurrency (transaction processing, rollback, etc.), fault-tolerance, and security features. Cost analysis has exponential complexity. For this reason, we rely on empirical measurements: we expect our algorithm to be near instantaneous for programs with small number of execution paths, but we also test the analysis on programs with large number of exeuction paths to illustrate usability limits.

1.2. Earlier work

Our approach to cost analysis of P4 programs is based on control flow graph (CFG) path-analysis [3]. In our earlier work, we introduced an approach that, in essence, enumerates all π execution paths in a g CFG, and derives its average execution cost as $E(g) = \sum_{\pi \in \text{paths}(g)} P(\pi)\text{cost}(\pi)$. That is, the average execution cost is the sum of the execution cost of each paths, weighted by the probability of those paths occurring.

In the main section of that work, we managed to decompose path costs into compositions of conditional expectations that encoded the dependence of a program statement on the effect of previous statements. With this, it became possible to analyse the components separately. Moreover, this step formalized the input requirements of our analysis either as conditional cost formulas, or as models from which these formulas can be derived.

In that work, we also presented a procedural algorithm to calculate the above formula given a CFG. The algorithm, starting from the root, traverses the nodes of each CFG paths in a breadth-first manner, collecting both the cost and probability information, and the conditions node-by-node on each path. The algorithm also features an optimization: many CFG paths share the same prefix, and the traversal avoids processing these segments multiple times.

In Section 3, we reformulated this algorithm as a GDB query in Gremlin. Precisely comparing the two notation is difficult, since in Gremlin, we only program one traverser, that is automatically split by Gremlin when it meets a branching point in the CFG. Without actually looking behind the abstraction and inspecting the GDB implementation, we can only guess that the query denotes the same behavior. On the other hand, this abstract formulation has advantages in itself, beyond those GDB features listed earlier. It generalizes our original algorithm for CFGs represented with different schemas, and it is ready to be parallelized, specifically at those points, where the traverser meets a branching point. In Section 4 of this work, we use empirical claims to argue that these benefits can be achieved without a significant loss in efficiency.

2. Related work

A question similar to our title was also asked by [7] about applicability of GDBs for static analysis (SA) in general. The authors emphasize the advantage of schemaless GDBs over RDBMSs in addressing cross-cutting concerns in SA queries. By extending vertices and edges with new properties, data flow, CFG, call graph, type

Name	In-memory	Data model	Lightweight edges	Single query parallelization
TinkerGraph-3.4.4	Yes	Native	Yes	No
Neo4J-0.7-3.2.3	No	Native	Yes	No
OrientDB-3.1.0M3	No	Document	Yes	Yes
JanusGraph-Cassandra-0.4.0	No	Wide-column	No	Yes

Table 1: Characterization of GDB providers

hierarchy, etc. information can be overlaid on the same graph structure (usually the syntax tree). In turn, these overlays can be queried using one, universal graph query. The authors also tested practical feasibility of GDBs for SA: they used Neo4J to store overlays on 2 million lines of JAVA source code and formulated graph queries in the declarative Cypher language. They found that loading took around 10 times longer than *javac* compilation, queries took 3-5 times longer with a cold (as opposed to warm) cache, and that query response time scaled linearly with program size. As even their slowest query took around 10 seconds on this large code base, they conclude that GDBs are indeed feasible for representing static analysis data structures.

In this paper, we implemented our algorithm as a graph query in Gremlin [6]. Gremlin is a compositional programming language embedded in Java (and many other languages) for describing graph traversals. In Gremlin, complex traversals are realized as compositions of elementary traversals, called steps (vertex and edge selections, conditions, repetitions, etc.). The language-specific Gremlin traversal is compiled to the language-agnostic bytecode of the Gremlin traversal machine. This traversal machine interacts with the graph database backends (*providers*) through a common graph API.

One important feature of Gremlin is that supports many graph providers: very different providers can be exchanged without any modification of the Gremlin queries. The survey in [1] provides a comprehensive classification of the most important graph backends as of 2019. In this work, we only utilize a fraction of these. (TinkerGraph – intended mostly for testing purposes – is not in the survey.)

Table 1 lists some of the features of the GDB providers we tested. Apache TinkerGraph is the only in-memory GDB, other GDBs have to cache persisted data as part of the queries. All providers are NoSQL databases: unfortunately, we could not find RDBMS-based GDBs that are compatible with modern Gremlin APIs. Native GDBs are systems that were built specifically to store graphs, as opposed to systems that "retrofit" other NoSQL data models, such as document-based or wide-column DBs. In practice, this means that native GDBs use efficient, close-to-machine implementations of the typical graph representations (adjacency lists, edge lists, etc.). This does not mean non-native data models can not represent

graphs efficiently. OrientDB, for example, supports lightweight edges: this means that vertices may store pointers to each other's location (as opposed to storing a vertex-index, on which a costly lookup is performed). While some GDBs support utilization of multiple cores for a single query, we used each GDB in single core mode to aid comparability.

3. Experiment design

To answer the presented question, we selected one small, and one large example P4 CFG, ran the algorithm in various configurations, and measured runtime. In this section, we describe these examples and algorithms.

3.1. P4 use cases

For profiling the algorithm, we selected two publicly available P4 programs, and then selected one CFG from each. In this section, we describe the selected components and our motivation as well for selecting these instead of other components. We collected the graph characteristics of the program components influencing our choices into Table 2.

The small example – chosen for testing and illustration purposes – is the `ingress` component from `basic_routing-bmv2.p4` [4], depicted in Figure 1. This switch program essentially describes a simple L3 IP router, and was found among the test cases of the official P4 reference compiler, P4C. The simplicity of P4 is well illustrated by the fact that `ingress` is the most complicated component of this L3 switch program.

This component is executed on a packet that was previously parsed into the structure named `hdr`, which stores the IP4 header in its `ipv4` field. First, it declares various match-action tables (packet matching algorithms) and defines actions (packet transformation algorithms) that are executed on matching packets. Tables basically map packets to actions. For terseness, we omitted the full declarations and definitions. Note, that tables are usually not defined in the P4 programs: these are received from the SDN controller during runtime. Then, the program describes the actual control flow. The packet is to matched by tables `port_mapping`, `bd`, and `ipv_fib`. In case that last table mapped the packet to action `on_miss`, `ipv_fib_lpm` is applied as well. Finally, table `nexthop` is applied.

As a real-world, large-size use case, for conducting plausible measurements, we chosen the `egress` component of `switch.p4` [5]. This switch program is capable of handling all of the most important protocols used in L2/L3 data planes. Unfortunately, it was written in version 14 of the P4 language ($P4_{14}$), so we had to use P4C to transpile it to the current version ($P4_{16}$). It is also unmaintained, so an older version of P4C is required for this process.

[1] For these measurements, we used a simplified cost model, and so the cost column displays the average number of basic blocks visited when traversing from entry to exit.

Name	*|V|*	*|E|*	*Choice*	Path prefixes	Full paths	Min	Max	Avg	Cost[1]
				basic_routing-bmv2.p4					
ParserImpl	4	3	1	3	2	1	2	1.5	1.5
ingress	8	9	2	11	3	2	7	5	4.25
egress	3	2	0	2	1	2	2	2	2
				switch.p4					
ParserImpl	50	133	24	15860	7477	2	20*	12.16	4.15
ingress	123	180	58	?	204750249984	30	?	?	?
egress	67	90	24	1015381	374547	3	62	44.77	18.0625

Table 2: Component CFG characteristics of two P4 programs

```
1  control ingress(inout headers hdr,
2                  inout metadata meta,
3                  inout standard_metadata_t stm) {
4    action set_vrf(bit<12> vrf) { ... }
5    action on_miss() { ... }
6    action fib_hit_nexthop(bit<16> nexthop_index)
7      { ... }
8    action set_egress_details(bit<9> egress_spec)
9      { ... }
10   action set_bd(bit<16> bd) { ... }
11   table bd { ... }
12   table ipv4_fib { ... }
13   table ipv4_fib_lpm { ... }
14   table nexthop { ... }
15   table port_mapping { ... }
16
17   apply {
18     if (hdr.ipv4.isValid()) {
19       port_mapping.apply();
20       bd.apply();
21       switch (ipv4_fib.apply().action_run)
22       {
23         on_miss: {
24           ipv4_fib_lpm.apply();
25         }
26       }
27
28       nexthop.apply();
29     }
30   }
31 }
```

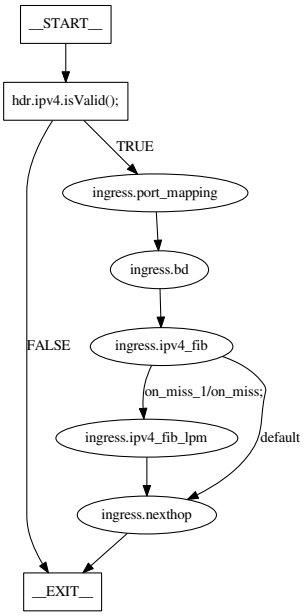

Figure 1: Source code and CFG of `ingress` control from `basic_routing-bmv2.p4` (created using P4C)

Another candidate from `switch.p4` was `ingress`. Unfortunately, this component has so many execution paths, our algorithm ran out of RAM and thus it could not finish in a reasonable time. In practice, we consider such large controls as extreme cases though: these were specific to $P4_{14}$, as $P4_{16}$ introduced custom controls that enabled better separation of concerns (resulting in smaller CFGs).

We also decided not to measure `ParserImpl`. It has the same size as `egress`, yet it is very flat, promising very fast executions. Unfortunately, it contains a self-loop, and cycles are deprecated in current versions of the P4 specification.

```
Double costAnalysis(GraphTraversalSource g,
                    Consumer<Traverser<Edge>> f,
                    Object sid,
                    Object[] eids){
    CostState cs = g.withSack(new CostState())
                    .V(sid)
                    .repeat(__.outE().as("e")
                              .sideEffect(f)
                              .inV())
                    .until(__.hasId(P.within(eids)))
                    .sack()
                    .fold(new CostState(),
                          (s, o) -> s.merge((CostState) o))
                    .next();
    return cs.getCost();
}
```

Figure 2: Executable cost analysis query in Gremlin-Java 3.4.4

3.2. CFG-based cost analysis algorithm

In this current paper, we profile a plain Java implementation of the algorithm presented in our earlier work. Beyond that – considering our motivation in the Motivation section to extend this software with application-centered features –, we adapt this algorithm to GDBs as well. We chose the Gremlin query language to formulate the algorithm, as according to [1], it is well supported across GDBs utilizing different data models.

Figure 2 depicts the query we constructed in the Java 8 language variant of Gremlin 3.4.4. Only `CostState` is a custom class, the rest can be understood using the documentation of the Java and Gremlin standard libraries. `CostState` is basically a pair that keeps account of the probability product and cost sum on the path being traversed.

The traversal starts traversing the graph from the vertex with the provided ID, and in each step creates a new copy of its local `CostState`, with its information updated by function `f`. This function accesses the current path history in the `Traverser` (assumed to include only named elements, which are the edges, in this case), queries the probability of the current edge (conditional on the path) and the cost of the destination vertex of that edge (conditional on the path) to update the probability product and cost sum belonging to this path in the sack. The copying ensures that parallel paths will not affect the state of each other. When the exit nodes are reached, the states from all paths are collected, weighted and summarized into one number, the average cost.

While the high-level nature of GDB query languages makes it difficult to map the Java implementation to the query language exactly, we believe this naive query approximates the runtime characteristics of the original algorithm reasonably well.

In comparison to our earlier paper [3], here we applied an extremely oversimplified environment model (and used the predicted value only for checking consistency). Namely, all vertices have a cost of 1 unit, and sibling edges have uniform probability (that is, if the source vertex of an edge e has n children, then e has $\frac{1}{n}$ probability).

4. Evaluation

Here, we describe the machine environment in which we conducted the experiment, and the measured results regarding two P4 code samples.

4.1. Configuration

Table 3 collects the specifications of the machine we performed the experiments on. We should note that all providers were running the query on a single JVM thread, so these numbers can characterize the execution environment only to a limited extent. All our experiments are *warm cache* measurements: each provider was tested in its own JVM instance, but all tests were running directly one after the other, without restarting the JVM.

CPU	**Clockspeed**	**Cores**[2]	**L1**	**L2**	**L3**	**RAM**
Intel Core i7-7500U	3.5 GHz	4	128KiB	512KiB	4MiB	7872 MiB

Table 3: Specifications of the processing environment

4.2. Case `basic_routing-bmv2.p4`/ingress

We measured algorithm runtime w.r.t. the two cases differently. For the small case, we simply repeated the experiment 100 times to average out random noise. These results are depicted in Table 4.

Provider	Java	TinkerGraph	Neo4J	OrientDB	JanusGraph
Runtime (μs)	172.16	1964.64	2517.68	5253.09	6728052.78

Table 4: Algorithm runtime on a small CFG input

In this small case, the GDB overhead results in at least one magnitude worse performance, compared to the Java implementation, but still, the CFG was handled almost instantaneously by most backends. (The large, 6 second overhead of JanusGraph was probably caused by improper configuration on our part).

Not all cost analysis approaches can provide this performance: traditional syntax tree transformation-based cost analysis methods are expected to have many magnitudes worse performance.

We can expect this performance to be stable even if new environment information is added to the model. Expanding nodes (as described in [3]) into sub-CFGs should only result in linear increase (unless we have no information about the input distribution of the expanded component, in which case this must be inferred from the incoming paths).

[2]All queries were executed on a single core.

4.3. Case `switch.p4`/egress

For the large case, we examined what happens if we only process an initial segment, or *prefix* of the graph. For all possible path lengths n, we selected the subgraph induced by the vertices in n distance from the start vertex. We then applied the algorithm to this subgraph, and measured its runtime. We picked 10 path lengths as a percentage of the longest path π in the CFG: so vertices in the prefixes were at least $\frac{1}{10}|\pi|, \frac{2}{10}|\pi|, \ldots, |\pi|$ distant from the start vertex (possibly more because of joining roundabout paths). For each length, we repeated the experiment 25 times and averaged the samples. The results are depicted in Figure 3.

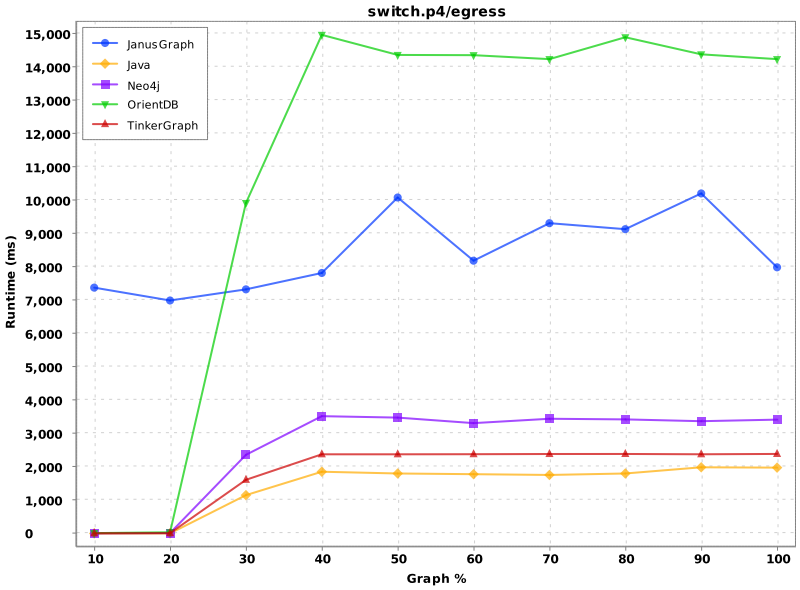

Figure 3: Algorithm runtime on a large CFG input

Almost all charts show a similar curve: presumably, the 40%-length prefix introduces a large number of paths, but this number does not grow significantly in higher length prefixes. We speculate that this CFG is of a diamond shape, where the diamond resides around $\frac{4}{10}|\pi|$ distance from the start vertex.

Interestingly, all charts also consistently show a convex valley by the 60%-mark, that is, CFGs with longer and possibly more numerous paths were processed faster. This phenomenon possibly indicates cache warming, or it maybe an interplay of traversal implementations and CFG-topology.

Among all the providers, the Java algorithm were the fastest, closely followed by the query propelled by the in-memory TinkerGraph. This is a positive result, implying that the query and the original algorithm are indeed possessing similar runtime characteristics. But, since Gremlin is a byte-code compiled language, the

difference is not necessarily explained by the overhead, and instead hints to an actual deviance in the formulations.

Somewhat surprisingly, Neo4J (a persistent database) performed quite close to the in-memory solutions. Neo4J's advantage over the other persistent databases is most likely due to the combination of efficient caching and it being a native GDB: the space efficient representation requires less file-level access, which enables Neo4J to behave similarly to the in-memory graphs.

In this experiment, non-native persistent GDBs were showing weaker performance characteristics, but we should note that this could be an effect of the design of our graph query. It concerns only the traversal itself, while data-access was implemented on the heap. Adding this data to the graph itself, and including more features (such as the processing of sub-CFGs, described in [3]) may change these characteristics. Using dedicated configurations on these databases (instead of the default ones) may also introduce significant improvements.

5. Conclusion

In this paper, we reformulated our earlier procedural P4 cost analysis algorithm as a graph database query. Our motivation is to leverage built-in GDB features, such as scalability and multi-user concurrency, but it was not clear if the GDB overhead will make us trade off analysis efficiency. To substantiate that using GDBs is viable for P4 cost analysis, we measured how our GDB query – backed by various backends – compares against our earlier algorithm, over large, real-world P4 use cases.

To answer the question in the title, our results imply that, yes, we can utilize GDBs without losing much of the efficiency. While GDB backends must be chosen and configured properly for the purpose, the portability of the query language will presumably make these efforts feasible as well.

We should note that these measurements are far from conclusive yet. Both our original algorithm and the query presented here can be considered naive: while these can analyse small CFGs real-time, future optimizations (e.g. parallelization) may enable faster processing even for very large CFGs (such as those in `switch.p4`) as well. We treated all GDBs as blackboxes: precise configuration of these may drastically improve their runtime behaviour. Devising further graph queries in languages other than Gremlin would enable us to investigate more graph backends (e.g. compare various in-memory GDBs against each other). We had to leave the in-depth exploring of these topics to the future.

References

[1] Besta, M., Peter, E., Gerstenberger, R., Fischer, M., Podstawski, M., Barthels, C., Alonso, G., and Hoefler, T. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries, 2019.

[2] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[3] Lukács, D., Pongrácz, G., and Tejfel, M. Performance guarantees for P4 through cost analysis. In *Informatics'2019, 2019 IEEE 15th International Scientific Conference on Informatics, Poprad, Slovakia, November 20–22* (2019), W. Steingartner, Štefan Korecko, and A. Szakál, Eds., IEEE, pp. 242–247.

[4] P4 Language Consortium. basic_routing-bmv2.p4, a small test case for the official P4 reference compiler, P4C, 2018.

[5] P4 Language Consortium. switch.p4, a large P4 project handling protocols of L2/L3 data plane, 2016.

[6] Rodriguez, M. A. The Gremlin graph traversal machine and language (invited talk). *Proceedings of the 15th Symposium on Database Programming Languages - DBPL 2015* (2015).

[7] Urma, R.-G., and Mycroft, A. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming 97* (2015), 127 – 134. Special Issue on New Ideas and Emerging Results in Understanding Software.