Verification and Performance Evaluation of a LPM Router

Simon Perriard, Hédi Sassi

June 2019

Contents

1 Introduction

Performance and reliability have always been the top priorities for every networking application, especially in the domain of IP routing. Data centers, clusters and the whole Internet are supported by a backbone of performant and reliable IP routers. Millions of dollars are at stake if any of these routers go down.

To ensure the robustness and performance of these routers, manufacturers design routers that run on specific hardware and that are rigorously tested. Such process takes a long time, is not cheap and cannot be quickly deployed.

On the other hand, an interesting but often overlooked solution consists of having a software that runs on a standard server and performs the desired routing algorithm. They are easier to set up, cost much less than hardware routers and can be formally verified using formal verification tools like VeriFast. However, this comes at the expense of the performance. But, for most of the non performance-critical applications like a home router, the benefits of having a software router outweigh the performance loss.

The goal of this project is to formally verify an LPM (Longest Prefix Match) data structure that can be used within a software router and compare the performances of a state-of-the-art software router with our verified implementation.

2 Verification

2.1 Symbolic Execution

Symbolic execution is a way to analyze a program by replacing every inputs to the program by symbols. These symbols don't have any value and thus, at every branch, the program takes the two paths, generates constraints for each paths and continues the symbolic execution. That way, symbolic execution tries all the possible paths in the program and, eventually, solves the accumulated constraints to see if there exists a path that leads to a program failure.

Example of symbolic execution:

```
void *foo(){

    // symbolic execution assigns a symbol to x. x = s
    void *x = receiveValue();


    // Here Symbolic execution takes two paths.
    // Path 1 constraint: s == NULL
    // Path 2 constraint: s != NULL
    if(x == NULL){
        fail();
    }

    return x;
}
```

Since the number of paths grows exponentially as the number of branches increases, symbolic execution does not scale.

2.2 Formal Verification

Formal verification is a way of proving the correctness of a program, using formal methods. These are techniques that allow a rigorous reasoning about a program to show its validity with respect to a formal specification, they are based on the program semantics. The formal verification tool we used for this project is VeriFast.

VeriFast is mainly developed and maintained by the imec-DistriNet research group from the Department of Computer Science of the KU Leuven - University of Leuven in Belgium. It is a research prototype tool for formal verification and correctness properties of single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic (axiomatic semantics). VeriFast lets the programmer express the specifications with inductive datatypes, recursive functions over these datatypes and abstract separation logic predicates. To verify these specifications, the programmer will have to write lemmas (functions that serve as proofs that their precondition leads to their postcondition). The verifier (SMT solver) will then check that the code complies with the expressed specifications. (More on VeriFast here.)

2.3 Vigor Approach

Symbolic execution is known to lead to path explosion and, therefore, cannot be applied in general to verify a whole program. But, if the code is stateless, symbolic execution is not subject to path explosion. The Vigor approach lever-

ages the ease of use of symbolic execution by splitting the code into a stateless part that can be verified by symbolic execution and another part that is verified with formal verification methods.

This way, only a relatively small proportion of the code needs to be formally verified and the proofs of correctness for the data structures can easily be reused for other software.

2.4 Verifying the Router

The verified LPM uses the DIR-24-8-BASIC LPM implementation described in the "Routing Lookups in Hardware at Memory Access Speeds" paper. For now, only the stateful code, i.e. the DIR-24-8 data structure, is formally verified. According to the Vigor approach, to achieve a fully verified LPM router we still need to replace the data structure with a model that can be verified with symbolic execution.

2.4.1 Stateful code

The stateful code in the router is the DIR-24-8 data structure, along with its functions, that are lpm_allocate, lpm_free, lpm_lookup_elem and lpm_update_elem. The choice of the DIR-24-8 implementation, which is limited in its number of rules, over a Trie implementation is motivated by its lookup cost efficiency.

The assumptions we made for the implementation are:

- Rules are inserted once and for all at program start-up, in order to add, modify or delete rules, the user will have to update the "routes" file, which describes the rules as IP address, prefix length, next hop, and then re-run the program.
- Rules are inserted in ascending order of prefix lengths, so that general rules will be overwritten by more precise ones. If a rule with a prefix length < 25 is going to be overwritten by a rule having a prefix length > 24, the previous rule has to be inserted in the remaining TBLlong entries for the corresponding index. If this is not done, the state of the routing tables will encounter inconsistency from what was intended.

These assumptions were made to simplify the data structure semantics, thus easing the formal verification process.

2.4.2 The LPM API

The LPM API contains a struct and four functions that can be used by the user:

- struct rule: represents a rule, it contains three fields, (uint32_t) ipv4, (uint8_t) prefixlen, (uint16_t) route. They represent the IP address, the rule's length and the next hop respectively.
- lpm_allocate(): allocates memory for the data structure, initializes every entry to 0xFFFF (invalid) and returns a pointer to the data structure.
- lpm_free(struct lpm *_lpm): frees the memory allocated for the data structure.
- lpm_update_elem(struct lpm *_lpm, struct rule *_rule): updates the routing table with a new rule.
- lpm_lookup_elem(struct lpm *_lpm, uint32_t ipv4): returns the next hop, if defined by a rule that applies to 'ipv4', 0xFFFF (invalid) otherwise.

2.4.3 VeriFast Representation

In VeriFast, the data structure is internally represented by two lists. TBL24 is a list<option<pair<bool, Z> > > and TBLlong is a list<option<Z> >, Z is an integer binary representation. The option inductive type is used to express the presence, or not, of a rule, if there is no rule for a specific IP address, 'none' will be found in the list and 0xFFFF will be returned. The boolean in the first table representation indicates whether a lookup in TBLlong is needed or not. Z is either the index for the TBLlong lookup, or the next hop. The whole DS is represented as dir(TBL24, TBLlong, next TBLlong index to be used).

The representations of the lookup and update functions in VeriFast follow quite directly the algorithm described in the paper that was mentioned earlier.

The lookup function reads the entry in TBL24 at an index computed from the searched IP address, from the result it returns either the next hop directly or from the result of the lookup in TBLlong at an index computed from the value given by TBL24, or 0xFFFF if no rule was matched.

The lookup is formalized as follows:

```
fixpoint int lpm_dir_24_8_lookup(Z ipv4, dir_24_8 dir){
  switch(dir){
    case tables(lpm_24, lpm_long, index_long): return
      switch(lookup_lpm_24(index24_from_ipv4(ipv4), dir)){
        case none: return 0xFFFF;
        case some(p): return
          switch(p){
```

```
          case pair(f, v): return
            f ?
              switch(lookup_lpm_long(indexlong_from_ipv4(ipv4,
                  int_of_Z(v)), dir))
              {
                case none: return 0xFFFF;
                case some(vl): return int_of_Z(vl);
              }
            :
              int_of_Z(v);
        };
    };
  }
}

fixpoint option<pair<bool, Z> > lookup_lpm_24(int index, dir_24_8 dir){
  switch(dir){
    case(tables(lpm_24, lpm_long, long_index)): return nth(index,
        lpm_24);
  }
}

fixpoint option<Z> lookup_lpm_long(int index, dir_24_8 dir){
  switch(dir){
    case(tables(lpm_24, lpm_long, long_index)): return nth(index,
        lpm_long);
  }
}
```

The update function first decides whether it is TBL24 or TBLlong that will hold the new rule from the prefix length.

If the rule belongs to TBL24, the first index and the size (number of entries) the rule will take are computed. Then TBL24 is updated at the correct indexes.

If the rule belongs to TBLlong, the index in TBL24 where the index for TBLlong should be stored is computed and a lookup at this index in TBL24 is performed. Now from the result of this lookup, it is decided whether a new index of TBLlong needs to be borrowed. If the TBL24 entry already points to TBLlong we use the existing index and update TBLlong in consequences, otherwise we borrow a new index for TBLlong and we update the entry in TBL24 and the entries in TBLlong accordingly.

The update function is formalized as follows:

```
fixpoint dir_24_8 add_rule(dir_24_8 dir, lpm_rule rule){
  switch(rule){
    case rule(ipv4, prefixlen, route):
      return prefixlen < 25 ?
              insert_lpm_24(dir, rule)
```

```
            :
              insert_lpm_long(dir, rule);
  }
}

fixpoint dir_24_8 insert_lpm_24(dir_24_8 dir, lpm_rule rule){
  switch(dir){
    case tables(lpm_24, lpm_long, long_index):
      return tables(insert_route_24(lpm_24, rule), lpm_long, long_index);
  }
}

fixpoint dir_24_8 insert_lpm_long(dir_24_8 dir, lpm_rule rule){
  switch(dir){
    case tables(lpm_24, lpm_long, long_index): return
      switch(rule){
        case rule(ipv4, prefixlen, route): return
          //Check whether a new index_long is needed
          is_new_index_needed(lookup_lpm_24(index24_from_ipv4(ipv4),
              dir)) ?
            //Check for available index, if not -> no change
            long_index == 256 ?
              tables(lpm_24, lpm_long, long_index)
            :
              //Update the value in lpm_24 and lpm_long
              tables(update_n(lpm_24, compute_starting_index_24(rule),
                              N1,
                              some(pair(true,
                                        Z_of_int(long_index, N16)))),
                     insert_route_long(lpm_long, rule, long_index),
                     long_index + 1)
          :
            //No need to update the value in lpm_24, only in tlb_long
            tables(lpm_24,
                   insert_route_long(lpm_long, rule,
                                     extract24_value(lookup_lpm_24(index24_from_ipv4(ipv4),
                                     dir))),
                   long_index);
      };
  }
}

fixpoint list<option<pair<bool, Z> > >
    insert_route_24(list<option<pair<bool, Z> > > lpm_24,
                                                  lpm_rule rule){
  switch(rule){
    case rule(ipv4, prefixlen, route):
      return update_n(lpm_24, compute_starting_index_24(rule),
                      nat_of_int(compute_rule_size(prefixlen)),
                      some(pair(false, Z_of_int(route, N16))));
```

```
  }
}

fixpoint list<option<Z> > insert_route_long(list<option<Z> > lpm_long,
                                            lpm_rule rule, int base_index){
  switch(rule){
    case rule(ipv4, prefixlen, route):
      return update_n(lpm_long,
                      compute_starting_index_long(rule, base_index),
                      nat_of_int(compute_rule_size(prefixlen)),
                      some(Z_of_int(route, N16)));
  }
}
```

2.4.4 Stateless code

The stateless code is the main reception/transmission loop that processes the packets received by the NIC. This part of the code is not modified by the incoming packets since it calls the stateful part that does the lookup and sends back the packets.
Thus, it can be easily verified using symbolic execution.

2.4.5 Verification Effort Estimate

Here are some data about the verification process.

Step	Lines of Code	Hours spent
router implementation (router + parser)	219	10-12
LPM implementation	260	15-20
LPM verification: logical specification	351	5-7
LPM verification: correctness proof	775	180-200

Clearly, the correctness proof represents the biggest effort in terms of LoC and time. However, the more libVig is enriched with a variety of new fixpoints and lemmas the quicker the correctness proof will be conducted for future data structures or network functions verification projects.

3 Testing

3.1 Tested Routers

All the tested routers are, at their core, built upon DPDK. This framework allows fast packet processing by bypassing the kernel and working directly with the network interface card (NIC). Without that, each packet would trigger a user mode to kernel mode switch that has a significant overhead.
The lookup algorithms used by the routers are modified version of the DIR-24-8 LPM routing algorithm.

3.1.1 Router with Unverified LPM

We decided to write a simple router with an unverified LPM strucuture to compare it against the verified implementation. This router uses DPDK's API for LPM routing. The version of DPDK used is DPDK 17.04.

3.1.2 Vigrouter

The router with the verified LPM strucutre, Vigrouter, shares some initialization functions with the router with the unverified one but its stateful part of the code has been formally verified using VeriFast.

3.1.3 Moonroute

We chose to test Moonroute because it has very high performances and, since it uses batch processing, should show interesting results during the testing. Moonroute uses DPDK 16.04.

3.2 Test Protocol

3.2.1 Assumption

The only function being tested is the LPM lookup function and the state of the router doesn't change during the tests.

3.2.2 Hardware Setup

We launched the router on the first server with only one core assigned and one NIC. It is connected to another server that runs the packet generator, the receiver process and a timestamping process to measure the latency.
We choose MoonGen to generate packets for it's ease of use and high performances. The cable connecting the two NIC are 10Gb/s links and the NIC model is the 10 Gigabit Ethernet Intel® 82599ES. The server is equipped with the Intel® Xeon® Processor E5-2667 v2.

This hardware setup doesn't allow a load bigger than 11.57 Mpps with a packet length of 84 bytes and therefore, our tests will only cover that range.

3.2.3 Measures

We tested the routers with two scripts. The first one was designed to trigger cache hits during the lookup by always having the same destination address in the packets or iterating on a small range of possible IP addresses. We did so to see what where the best case performances.
The second script was designed to trigger cache misses by generating packets with pseudo random destination addresses. That way we were able to see what are the impact of a cache miss on the performances.

For both tests, we took different latency and drop rate measures as we increased the load on the router. During the tests, all routers had 20 thousands random routes inserted with one percent of the routes having a prefix length of more than 24 bits. We choose to keep the number of long prefixes low since most of the traffic in real world application has a small probability of having a destination address with a prefix longer than 24 bits.

4 Results

4.1 Cache Hit Test

We can see from the graphs that the routers with the verified and unverified LPM structure have similar performances. They both have a maximum throughput of 7.46 and latency that span from 3.8 micro seconds up to 10 micro seconds before the bottleneck as we can see on Figure 2 After the bottleneck, all packets have a latency of 0.5 milliseconds.

On the other hand, Moonroute has a completely different graph. The latency is higher in general and is extremely bad under a small load but the throughput is much better than the other two routers. Moonroute achieves a throughput of 11.38 Mpps, 52 percent more than other two routers but the latency between 1 and 7.4 Mpps is 4 to 5 times bigger.
This behavior is characteristic of batch processing that tends to trade latency for a higher throughput. At low load, the packets don't fill the batch size quickly and thus wait in a queue and contribute to the big latency peak we can see for Moonroute on Figure 1 . Even at moderate and high load, batching impacts greatly the latency.

The drop rate graphs are very similar for all the routers. No packets are dropped until the queue starts to fill. The inflection point corresponds to the spike in the latency graphs which is expected since both the drop rate and the latency strongly depend on the length of the queue.

Summary of performances during the cache hit test. The latency shown is the latency measured at 95 percent of the maximum throughput

Router	Max Throughput (Mpps)	Latency (μs)
router with unverified LPM	7.47	7.78
router with verified LPM	7.46	7.1
Moonroute	11.38	30.5

Figure 1: Latency and drop rate results for the cache hit test

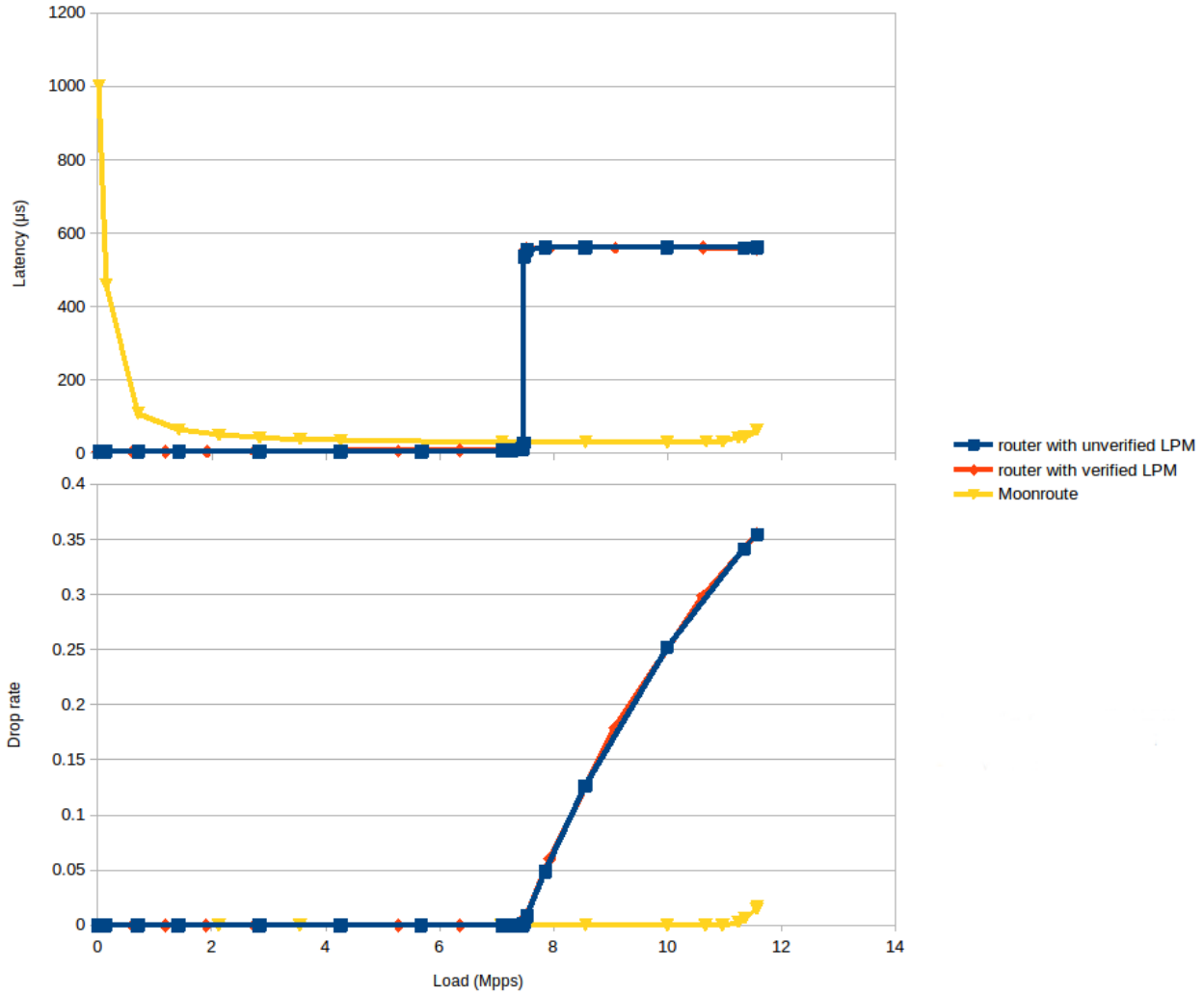


Figure 2: Results before bottleneck for the routers with the verified and unverified LPM

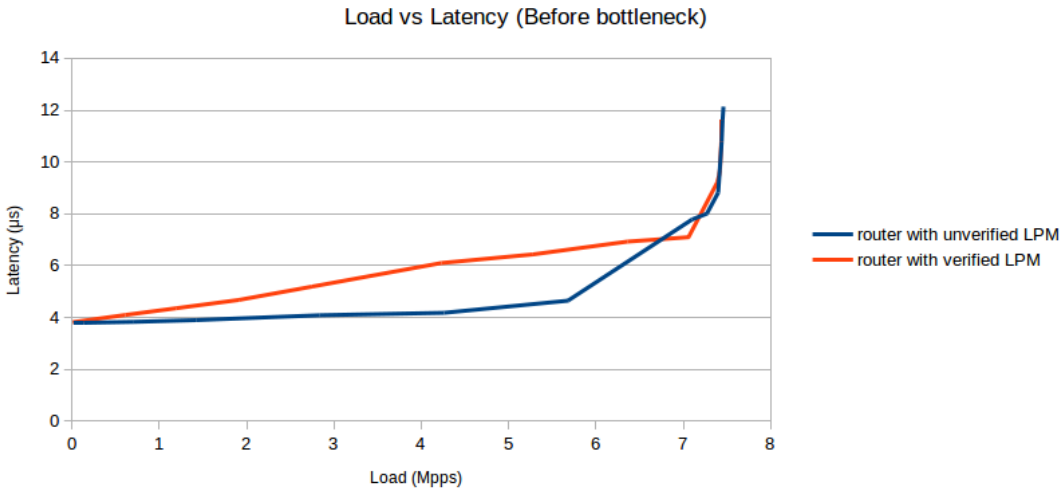
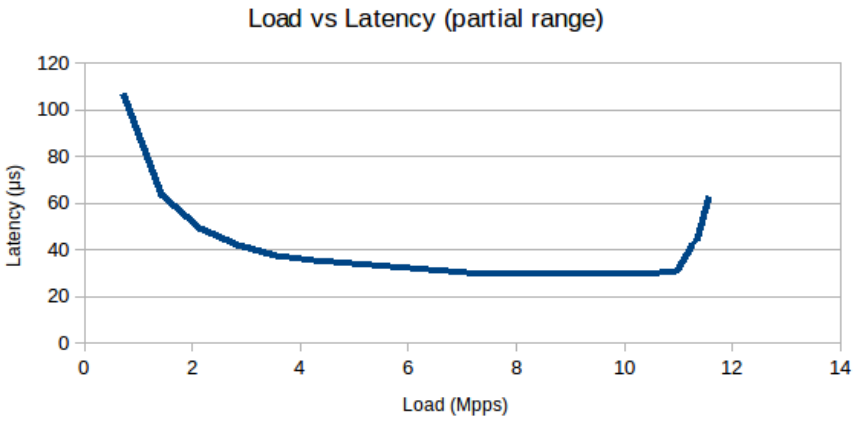


Figure 3: Results zoom for Moonroute



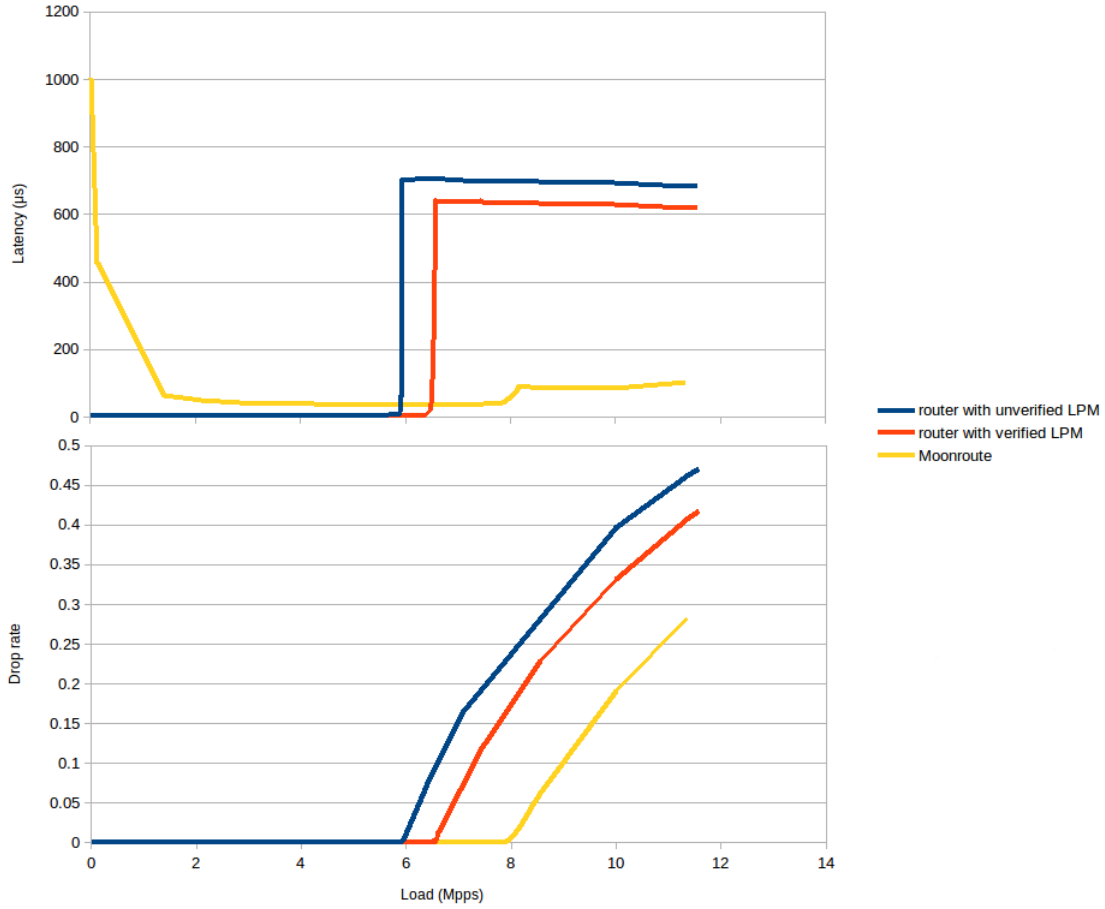
4.2 Cache Miss Test

The results for the test that triggers cache misses give similar curves as the ones from the cache hit test but they are slightly shifted to the left.
The cache misses strongly impacted Moonroute with a 39 percent drop on the maximum throughput. The router with the unverified LPM structure got a 21 percent performance drop and the router with the unverified LPM structure only got a 10 percent performance drop.
The difference in throughput between the router with the verified LPM and the one with the unverified one can be due to bad cache utilization since the two implementations operate in the same way and perform the same memory accesses. The curve for the latency before bottleneck stayed the same for all three routers with only minor changes.
With this test we can clearly see the impact of cache misses on the throughput. The algorithm used for the lookup heavily depends on cache performances since most of the lookup cost is taken by memory access. This appears to be even more critical for Moonroute since a single miss can delay the whole batch.

Summary of performances during the cache miss test. The latency shown is the latency measured at 95 percent of the maximum throughput

Router	Max Throughput (Mpps)	Latency (μs)
router with unverified LPM	6.12	5.9
router with verified LPM	6.74	8.46
Moonroute	8.13	40

Figure 4: Latency and drop rate results for the cache miss test



5 Conclusion

Despite a lengthy verification process, the benefits of having a verified implementations outweighs the initial verification cost. In addition, our verified LPM structure can be reused by anyone without them knowing a single thing about formal verification. Even parts of the proofs used to verify the LPM structure can be reused for other verification projects since some are generic.

From the results we got, we can conclude that our verified implementation has the same performances as the one based on DPDK's API. For applications that don't require a throughput bigger than 7.4 Mpps, our implementation with the verified LPM performs better than Moonroute with a latency 4 to 5 times lower. However, it becomes unusable once the bottleneck is reached and batching should be considered if a bigger throughput is required.

Overall, we have seen that it was possible to write a router with a verified LPM structure that can compete with APIs that are used in the industry.

6 End word

This bachelor project has been supervised by Arseniy Zaostrovnykh, senior Phd at DSLAB, EPFL.

7 References

"A Formally Verified NAT" Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, George Candea

"MoonGen: A Scriptable High-Speed Packet Generator" Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle

"Building Fast but Flexible Software Routers" Sebastian Gallenmüller, Paul Emmerich, Rainer Schönberger, Daniel Raumer, and Georg Carle

"Comparison of Efficient Routing Table Data Structures" Dominik Schöffmann

"Routing Lookups in Hardware at Memory Access Speeds" Pankaj Gupta, Steven Lin, and Nick McKeown