Tiny packet programs as a p4-backed DSL

Peter Li (pl488@cornell.edu), Tyler Ishikawa (tyi3@cornell.edu)

December 11, 2018

Source: https://github.com/peteli3/p4-tiny-packet-programs

1 Abstract

Active networking poses an interesting take on leveraging network resources to access the global state with scale in mind. The dream is to have instantaneous freeze and read for gathering information on the state of the network or instantaneous freeze and write for managing the state of the network. But real world constraints make this impossible. Tiny packet programs [1] (henceforth referred to as TPP), a variant of active networking, seeks to provide compact primitives for dataplane programming that can leverage network packet travel time to get work done. In this paper, we will discuss our design and implementation of TPP using the p4 language and highlight the following: 1) the specific challenges we faced as a result of choosing p4 to back TPP, and 2) exploring alternative primitives to those specified in the original TPP paper.

2 Introduction

This paper will serve as: a design doc for TPP in p4, a report on the p4-specific challenges we faced, and also an informal proposal for some alternative primitives we implemented over the existing ones. We felt that p4's existing packet parsing mechanisms were very conducive TPP parsing: the FSM-like parsing control in p4 switches gives us fast and interpretable processing on the header stacks at the sacrifice of a single bit for separation for each stack member. This is a desirable property for TPP because it means we will be able to write variably-sized instruction sequences and memory stacks without bloating TPP interpretation logic. We note that an implementation of this flexible nature strictly subsumes its fixed-size counterpart (e.g. the original TPP that Jeyakumar et al. implemented).

For the purposes of exploring expressivity, we allow for larger TPPs than the original implementation since our TPPs will already be sacrificing performance: they are interpreted over p4 which is a compiled language, so the amount of underlying assembly instructions per TPP instruction will be bloated to begin with. As such, to differentiate our TPP from the original, we will refer to it as TPP*.

3 TPP* Design

We implemented TPP* in its 2 major components in p4: 1) the packet formatting and 2) the switch interpreter. Additional work was done in Python to enable testing in the bmv2 and mininet framework, but we will only briefly cover it in this report since is not part of TPP*.

The culmination of our TPP* implementation is a single `tpp.p4` module and an associated header module that can be imported by an existing `switch.p4` (call it the baseline switch). The only integration effort required to run TPP* is to:

1. Call into the TPP* controls at the end of each analogous baseline switch control. For instance, switch parse should call TPP* parse, switch ingress should call TPP* ingress, and so on.
2. Ensure that the imported TPP* header formats are included in baseline switch's header struct.

Section 3.1 will discuss in detail TPP* packets and their parsing schemes, section 3.2 will cover the p4 module for interpreting and executing TPP* instructions, section 3.3 will discuss our proposals to extend TPP, and finally section 3.4 will cover our testing environment (e.g. program logic, network models, topologies).

3.1 TPP* packets

TPP* packets have 3 main sections: 1) a header which holds program metadata, 2) an instruction sequence, and 3) a memory section. In order to accommodate for the lack of looping semantics in p4, we chose to implement each of the 3 sections as separate header types. We implemented the metadata as a standard header that can be read in 1 call to `packet.extract`, and instructions and memory words as header stack elements with 31-bit "almost-words" (for instructions we will refer to these as "opcodes", and for memory we will refer to these as "memory slots") that have 1 bit of separation between them.

Metadata header
We remained faithful to the TPP header specification by tracking the following: length of packet, length of memory stack, memory mode, stack pointer, length of memory hop, and checksum. As of right now, we only implemented stack memory for testing so the "memory mode" and "length of memory hop" fields are unused. The only change we made to the TPP specification was to add an additional byte of metadata — 1 validity bit and 7 bits counting the number of instructions — to satisfy the scapy packet formation script which requires layers to be bound to constant values from their direct parent header.

Instructions in TPP*
Each instruction on the packet was implemented as a 32-bit header struct of type `header_insn_t` which contains a 31 bits of opcode data and 1 bit of separation. The separation bit is 1 for the

31 bits total

3 bits	8 bits	8 bits	8 bits	4 bits
instruction	rd - first operand	rs1 - second operand	rs2 - third operand	flex

Figure 1: TPP* opcode format

last instruction in the TPP* sequence and 0 for all others. Also similarly to source routing, the separation bit is used to determine the end of the recursively applied parsing mechanism.

We wish to be flexible with the number of instructions and memory slots for exploratory purposes. To make TPP* conform to the original specification of exactly 5 instructions and 50 slots of memory, we can simply impose assertions on the packet construction script in Python and the TPP* interpreter could still do its job while being blissfully unaware of the original TPP restrictions.

Each opcode is 31 bits long and the breakdown of our opcodes are shown in Figure 1. We used 3 bits for the instructions since there are only 6 instructions in the original TPP specification, 8 bits for each operand since they are either switch register indices or packet memory slot indices and will most likely not be a decimal value above 256, and 4 bits as “don’t care” (call these flex bits). However we will propose a use case for them in a later section of this paper.

We note that despite CSTORE being the only existing TPP instruction that requires a third operand, we opted to make all opcodes be read as if they had 3 operands. This decision vastly simplified our TPP* code in development without sacrificing expressivity of the TPP language. We say this because only 6 bits are realistically needed to index into the packet memory since, by specification, the max amount of available memory slots on a TPP packet is 50. We give it 8 bits in order to keep the operands evenly sized. Registers on a switch however may potentially be much larger, in which case, our design suffers from inability to index past the the first 256 registers.

We leave to future work to explore the feasibility of using the 4 flex bits to augment to any operand of choice. For example, if the first operand `rs1` with 8 bits is not enough to index into the switch registers, we can consider bit shifting it to the left by 4 and then adding in the value stored in the 4 flex bits. This creates a virtual 12-bit switch register index. We believe that it will be a very nontrivial matter to encode this operand augmentation as it may involve writing new instructions like `STORE_AUG_FLEX` which say: use the normal `STORE` semantics but with the second argument, which supplies the switch register to which we want to write a TPP memory value, augmented by using the 4 flex bits. From this idea, one can also imagine introducing instructions that support immediate values, bringing TPP* closer to an assembly language (which may defeat its purpose but also make it usable in a much larger variety of contexts). We justify this future work by our belief that expressivity should outweigh potential performance degradation as long as the costs are

Instruction	Description
`PUSH, FROM_REGISTER`	Pushes the value at `FROM_REGISTER` onto the packet memory's stack
`LOAD, FROM_REGISTER, TO_LOCATION`	Copies the value at `FROM_REGISTER` to the packet memory at `TO_LOCATION`
`POP, TO_REGISTER`	Copies the value from the top of the packet memory stack to the `TO_REGISTER`
`STORE, TO_REGISTER, FROM_LOCATION`	Copies the value from the packet memory at `FROM_LOCATION` to the `TO_REGISTER`
`CEXEC, REGISTER, LOCATION`	Move on to the next instruction if the value at `REGISTER` is equal to the value at `LOCATION`, else stop
`CSTORE, REGISTER, OLD_LOC, NEW_LOC`	If the value at `REGISTER` is equal to the value at `OLD_LOC`, copy the value at `NEW_LOC` to `REGISTER`, else do nothing

Figure 2: all TPP* instructions with operands and brief descriptions

not crippling. One can imagine having multiple levels of priority queues on each switch to handle TPP*s at a lower priority for the sake of mitigating performance costs, if necessary.

Figure 2 displays all the instructions we support and the expected syntax used for parsing. In each instruction of figure 2, the TPP* parser will read the 31-bit opcode from left to right as: `instruction`, `rd`, `rs1`, `rs2`. Refer to figure 1 for the bit lengths of each member. For example, in `CSTORE`: `instruction` = `CSTORE`, `rd` = `REGISTER`, `rs1` = `OLD_LOC`, and `rs2` = `NEW_LOC`. For instructions with less than 3 operands, the remaining unused opcode sections will simply be ignored after parsing. This style of parsing is intentionally similar to the way that assembly instructions are parsed.

Memory in TPP*

Similar to the formatting of instructions, each word of memory is represented by a 32-bit header struct of type `tpp_mem_t` which contains 31 bits of memory slot data and 1 bit of separation. Parsing of memory is identical to that of instructions.

While constructing a TPP* packet from the sender-side, we used a parametrized constant to determine how many memory slots the packet should have. We allowed the TPP* constructor script to initialize packet memory with a predefined set of values (dependent on use-case). Remaining memory slots were then filled with zeros to give room for TPPs to collect data from switches.

While implementing TPP* memory access, we needed a way to fit each word (or slot) of memory into a data structure that can be indexed into. The alternative would be to read an entire TPP* packet as a bit stream and access values in the packet via pointer arithmetic. However that option seemed difficult if at all possible in p4 and would make poor use of the language's features. We thus settled on using the header stack structure for TPP* memory so we could easily index into it

```
insn = hdr.tpp_insns[insn_index];
if (insn valid && no prior cexec check says to stop) {

    # extract relevant instruction operands
    insn_encoding, rd, rs1, rs2 = parse_insn(insn);

    # match & run the actual instruction (for cstore, set bool insn_is_cstore)
    tpp_insn_action.apply();

    # if cstore predicate is true, do the actual store and reset flag
    if (insn_is_cstore) {
        tpp_cstore_store();
        insn_is_cstore = false;
    }
}
```

Figure 3: TPP* eval as pseudocode in p4

to read and write to memory slot values.

We found that the p4 compiler did not allow us to write push or pop semantics in an intuitive way since those instructions indexed into the TPP* memory header stack using the stack pointer obtained from the TPP* header, a value that is not known at compile-time. To work around this challenge, we ultimately had to create a copy of TPP* memory to be kept on the switch at the start of TPP* ingress, make the sequence of instructions to only modify that copy, and finally flush the copy back to the TPP* packet at the end of ingress. This workaround was highly undesirable as the amount of read/write instructions introduced is a big price to pay (it also scales linearly with the size of memory) to be able to index into a header stack using a runtime value.

3.2 Switch interpreter

TPP* interpretation and execution happens during ingress processing. Figure 3 displays the high-level procedure of evaluating each TPP* instruction.

Ideally we should be able to loop on the instructions parsed since we previously permitted processing TPP* packets that have variable length instruction sequences. But by design p4 does not allow arbitrarily long loops making this ideal implementation impossible. As such, we resorted to loop unrolling to evaluate only a predefined number of instructions (call this number N, we use $N = 5$ in our testing to mimic the original TPP specification); in other words, our TPP* module can only run the first N instructions of any TPP* packet that it sees.

We believe that in the future, a sufficient way to deal with this constraint is to expose N to the switch as a parameter that can be set only via TPPs that carry sufficient authorization tokens.

This way, the baseline switches themselves can remain unaware of what N is but a trusted SDN controller has the ability to set N to some value within reason, thus effectively adding a control layer on how TPP* switches operate.

Another language-specific constraint we uncovered in implementing TPP* interpretation was the lack of reusability of tables. After loop-unrolling instruction evaluation, we found that the p4 compiler was unhappy with our multiple use of the same table. For example, the following sequential logic was not possible since it may use the `tpp_insn_action` table more than once:

```
if (insn 1 valid) { apply tpp_insn_action table on insn 1 };
if (insn 2 valid) { apply tpp_insn_action table on insn 2 };
if (insn 3 valid) { apply tpp_insn_action table on insn 3 };
# and so on
```

The exact error message we observed was: `Program cannot be implemented on this target since there it containsa path from table MyIngress.tppIngress.tpp_insn_action back to itself`. We believe this error message is a bit misleading since we originally interpreted it as "one of the table actions leads back onto a path where the table is called again, resulting in recursive execution." But after some speculation we surmise that this error message may stem from the need to perform branching or jumping logic to reuse the table, which it seems p4 is not fond of by design. To work around this, we created duplicate tables and associated entries on our runtime json to handle N instructions. This was also an undesirable workaround due to how much it bloated the code in a way that is prone to human error. Fortunately, many errors that may arise can either be caught at compile-time (e.g. aligning a table entry with the wrong table), or can be safely ignored (e.g. calling the wrong table).

We reiterate that we believe the error message mentioned above is either: 1) unfitting of the actual problem since the path isn't from the table to itself but rather from an outer string of execution sequentially attempting the same match-action table, or 2) a result of a compiler bug in which the compiler treats sequential application of tables as a "path from one table back to itself."

3.3 Our extensions

During our exploration into applications for our TPP* language, we ran into a few scenarios where we felt that extensions to the language were necessary to implement some key functionality. One example of this came to light when we considered ways to determine where packets are dropped in a network path. A straightforward approach to this problem would be to set up a TPP* packet that knows how many packets have been sent along the path and records the ID of any switches that have seen fewer than that many packets. With this information, we would be able to conclude that the switch directly before the first recorded switch ID must be the switch dropping the packets.

Although this would be trivial to implement in a full-fledged programming language, we realized

Instruction	Description
`CMPEXEC, REGISTER, LOCATION, CMP_OP`	Move on to the next instruction if the values at `REGISTER` and `LOCATION` satisfy the comparison operator `CMP_OP`, else stop. The encodings for `CMP_OP` are: equals = 0b000001, greater than = 0b000010, greater than or equal to = 0b000100, less than = 0b001000, less than or equal to = 0b010000, not equal = 0b100000.

Figure 4: syntax and brief description of the CMPEXEC instruction

that it is not possible to implement within the expressiveness bounds of TPP*. TPP* has an operator, `CEXEC`, for conditionally executing instructions, but `CEXEC` is limited to conditional execution on equality of values. If we want to conditionally execute on not-equal, greater-than, or less-than relations, we're out of luck using TPP*.

To combat this limitation, we worked to extend the language with a new operation, which we denote as `CMPEXEC`. For this instruction, in addition to the switch register and the TPP* memory location to compare, the third operand is used as an encoding of which comparison operator to use (less than, greater than, not equal, etc.). This is a one-hot encoding, which was chosen because it simplifies implementation-specific details in p4. Figure 3 explains these details.

With this extension to TPP*, the expressiveness of the language increases greatly, opening up possibilities for many more kinds of applications. At the same time, the conciseness of the language is maintained, allowing each switch to execute the TPP* program instructions in a non-blocking, small amount of time.

3.4 Testing framework

With our new `CMPEXEC` instruction, we tested our extended language by implementing the application of finding where packets are dropping, as was described in the previous section. For this test, we made a few simplifying assumptions. First, we assume that the hosts are SDN controllers with knowledge of the paths to other hosts. In addition, we assume that all switches along a path have "seen packet" counts that are monotonically decreasing downstream (drop detected) or flat (no packets dropped between switches). In other words, we assume that packet counts will never increase along the path if packets are dropped.

For our test, we used a simple linear topology, but our example should be easily extendable to more complex topologies.

Network update
First, to set up our test, we used TPP* to update switches along a path with dummy data on "amount of packets seen for a specific flow". The program used for this can be found in the file

set_packets_seen.json. For our test, we set switches s1 and s2 to have seen 250 packets and switches s3 and s4 to have seen 225 packets. Thus, the expected solution is that the packets dropped between s2 and s3. In our next test, we abstracted this network update away by thinking of the packet count as being generated from an actual counting mechanism.

Packet drop finder
We used TPP* with the new CMPEXEC instruction to figure out where the number of packets seen for that flow suddenly dropped. The program used for this can be found in the file find_bad_switch.json. The resulting packet contains three non-zero values in the TPP* memory: 250, 225, and 225. This signifies that the expected value was 250, but two switches at the end reported only seeing 225 packets. This means that packets were dropped before s3 and s4 but after s1 and s2. Note that this kind of analysis was not possible with just the TPP primitives in the original paper. Although it makes sense that Jeyakumar et al. wanted TPPs to be limited enough to be run quickly on the limited switch hardware, operators like greater than, less than, and not equal are not much more expensive than the equals operator that is already used in the CEXEC instruction. As a result, we believe the CMPEXEC instruction to be a valuable addition to the TPP* language, as well as a viable replacement to CEXEC since CEXEC is a strict subset of our proposed instruction.

4 Conclusions & future TPP* work

Through this project we began to better understand the limitations of p4's expressivity and by extension the limitations of domain specific languages that can be built on top of it. We also learned a lot about TPP's existing primitives and have still come to the conclusion that some of them seem much too limiting to be useful in any capacity, even considering the original reason for their genesis. We reiterate an earlier point that for a DSL, expressivity is the main goal and it should outweigh any non-crippling inefficiencies created as a result, especially if we can develop mitigation strategies for them. We have shown that some of the potential programs that Jeyakumar et al. deferred for future work are not actually implementable in the existing TPP primitives, and that alternatives need to be considered. Finally for reference, we refer the reader to our functioning implementation of TPP* on GitHub.

Post script: Due to time constraints, our repository is not in the most readable state. We would love for it to be eventually added to the P4 Language Consortium if deemed a worthy contribution, and we'd be happy to prepare our code upon request.

References

[1] Vimalkumar Jeyakumar, Mohammad Alizadeh, Changhoon Kim, and David Mazières. 2013. Tiny packet programs for low-latency network control and monitoring. In Proceedings of the

Twelfth ACM Workshop on Hot Topics in Networks (HotNets-XII). ACM, New York, NY, USA, Article 8, 7 pages.