P4 VPP Status

September 18, 2017

Andy Keep

Today’s Discussion

- P4 VPP Status Update
- Overview of using the current compiler
- Technical Overview of Current Compiler (time allowing)
    - Added passes
    - P4 to C translation strategy
    - Extern handling strategy
    - Package handling strategy
    - Wiring up V1Switch Package to VPP

P4 VPP Status Update Overview

- Supports most of the $P4_{16}$ base language (and $P4_{14}$ through translation)
    - **Missing features:** match table implementation; encap/decap deparser; parameterization of controls & parsers; variable sized bit fields (varbit); some operations on non-C-type signed & unsigned integers
- Supports recognizing P4 program package (architecture)
    - **Currently supported packages:** V1Switch ($P4_{14}$ translation package); AdHoc (compiler testing package)
- Supports only a few basic externs: packet_in, packet_out (partial), & verify
    - **High priority externs to add:** mark_to_drop; ip checksum; counters; etc.
- Generates single-node VPP plugin, with CLI enable/disable and table config message stubs
    - A simple Makefile, generated into the result directory, can be used to build the plugin on a system that has the VPP development headers and libraries installed. This needs testing, but will be uninteresting without tables.
- Immediate next steps: finish initial match table implementation; begin implementing externs
- Available in private github repo, targeting full open source release soon

P4 VPP Language Feature Support

- Supported features:
    - Basic expression language (including bit slice/concatenate)
    - Parser and parser states
    - Deparsers (partial, allows for in place edits, but not encap/decap operations)
    - Controls
    - Actions
    - P4 Types: Basic scalar types: bit, int, boolean, enum, error; headers; structs; header unions; header stacks; unsigned arbitrary sized integers (missing arithmetic/inequality operations)
- Unsupported features:
    - Tables (stubbed out, but not yet implemented)
    - Deparser (encap/decap operations)
    - Parameterized Parser and Controls
    - P4 Types: Variable sized fields; signed arbitrary sized integers

P4 VPP Package Feature Support

- Packages are supported by subclassing Package class and registering in package
    - Packages are looked up by name and a method can be overridden to check package type
    - Packages hold maps for extern functions and extern objects
- V1Switch - the $P4_{14}$ translation target package
    - Basic package support, wires up standard_metadata.egress_port to VPP packet next
    - No support for externs (yet)
- AdHoc - a compiler testing package
    - AdHoc is used for all unrecognized packages, this is useful for testing that the code generator produces compilable C code (and VPP-compatible) plugin; however, we do not know the egress port for the actual package, so we cannot fully wire it in to VPP.
- Future packages to support: PSA (currently under development); VPP-specific package(s)

P4 VPP Extern Support Status

- Currently there is very basic support for packet_in, packet_out, and verify externs
- Package class provides a new model for handling externs, allowing the package to map extern function and object names to an ExternFunction or ExternObject implementation
- Externs will come in two basic flavors: package independent and package dependent
  - Package independent externs can be implemented and registered in a master registry and used within the AdHoc package: examples include packet_in, checksums, and counters
  - Package dependent externs need to be implemented for each package, and depend on something within the package (like the package specific internal metadata): examples include mark_to_drop
- Next steps: The P4 VPP compiler still needs some infrastructure and examples, but as we make these available, we would welcome help implementing these

P4 VPP Table Support Status

- Basic table stubs exist for both the data plane (table apply) and control interface (table configuration) through generated VPP API and C API wrapper
- All the pieces are in place to implement tables:
  - working on initial implementation, but we are looking at using VPP hash table for exact matches, VPP IPv4 (<= 32 bit) and IPv6 (<= 128 bit) LPMs for lpm matches, and ACL-style table for ternary matches
- Once we have the basic implementation in place, there are two areas we could use some help:
  - First, on the VPP side, we need high-quality implementations of generic table matchers to support P4's tables
  - Second, on the P4 side, we could use help implementing the P4Runtime to VPP message API shim.

P4 VPP Unsupported Feature Reporting Status

- We have put some effort into reporting unsupported features, either as:
    - errors, when the lack of implementation impacts packet processing, or
    - warnings, when the lack of implementation does not directly impact packet processing
- Errors are implemented using p4c's BUG macro, which immediately halts the compiler, or ::error function, which stops the compiler after the current pass
- Warnings are implemented using p4c's ::warning function
- The big hole in this checking, currently, is in checking that the deparser does not re-arrange the outgoing headers in any way. We compile on the assumption it does not, so be aware of this in testing.
- As we begin to have a more complete implementation, we will need fewer of these, but there will always be some need for this, as a P4 program could contain features we have never encountered before.

P4 VPP Status Summary

- Close to a working, if simple model, with error check for most unsupported features
- Once we have enough complete that we can test a few simple programs, we plan to release this through the P4 VPP gerrit repo fully as open source
- We can provide early access to the current private github repo, contact akeep@cisco.com
- As we continue to expand this code we will need help with testing, extern implementation, library implementations, etc.

USING P4VPP

Using P4 VPP

- Prerequisites:
  - Install p4c perquisites: https://github.com/p4lang/p4c#dependencies
  - Install fd.io packages (needed for testing results of compiler only)
- Clone the project
  - git clone --recursive https://github.com/akeep/p4c-vpp.git
- Run configure (this will also pull p4c, if you forget the —recursive)
  - $ ./configure   (in p4c-vpp directory)
- Make the compiler
  - $ make   (in p4c-vpp/build directory)

Using P4 VPP

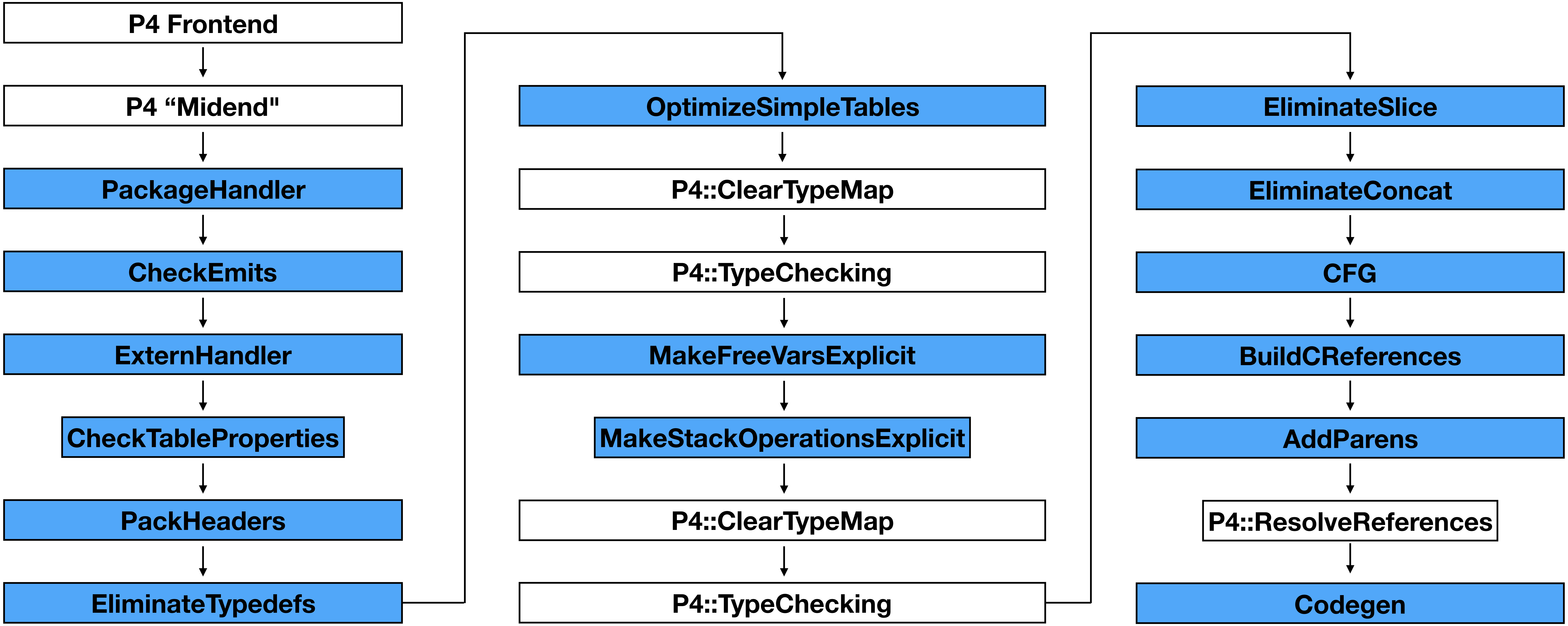
- Compiler is now built and linked in build directory
    - Note: It currently expects to find a p4vpp_static directory in the current working directory when it is run
    - $ ./p4vpp --help (from the p4c-vpp/build directory)
    - $ ./p4vpp -o <output dir> --use-new-codegen --p4v <version> <P4_version source file>
    - $ ./p4vpp -o /tmp/foo --use-new-codegen --p4v 14 ../testdata/p4_14_samples/gateway1.p4
- Small testing script, p4-programs, compiles p4 to C, compiles C code, reports failures
    - $ ln -s ../../../scratch/p4-programs . (from the p4c-vpp/build directory)
    - $ ./p4-programs --help
    - $ ./p4-programs   (compiles all programs in testdata/p4_*_samples)
    - $ ./p4-programs --exclude-all-failing (skips known failing programs)

Compiler Overview

P4 VPP Compiler Overview

- Initial compiler uses a simple translation strategy
    - Focused on correctness and completeness over performance (to begin with)
    - Largely uses the p4c IR, extended near the end with C pointer operations and types
    - Targets a single VPP node that expects to run traffic from ingress to egress
- Once we have a level of completeness, we will start to shift to focus on performance
    - Begin to break down action, control, and parser boundaries to enable optimizations
    - Target a single VPP feature arc, using multiple nodes when the P4 program is sufficiently complex

P4 VPP Compiler Passes



P4 VPP Compiler Passes

- VPP::PackageRunner/VPP::PackageHandler
    - Pass manager (PackageRunner) that attempts to determine the package based on the package name and if that package provides a preprocessor pass, runs the preprocessor pass. Uses adhoc package otherwise.
- VPP::CheckEmits
    - Checks that deparser emits mirror parser extracts, since we do not currently support encap/decap operations. This pass is currently incomplete and yields too many false positives to be useful, so needs to be revisited.
- VPP::ExternHandler
    - Checks each extern used in the program to see if it is supported and complains if it is not. Needs to be extended to use the package determined by the package handler to determine and apply extern implementations
- VPP::CheckTableProperties
    - Checks table properties to determine when a table uses unsupported properties and warns when the property is known not to impact packet processing, and raises an error when it is known to, or when the property is unknown

P4 VPP Compiler Passes

- VPP::PackHeaders
  - Packs together header fields that do not fit in C types and uses the slice operator where fields are referenced to extract those fields. This attempts to create wire-format structs similar to the way VPP hand-coded headers do.
- VPP::EliminateTypedefs
  - Removes P4 typedefs by replacing references to those typedefs with the underlying type, this is just to remove some unnecessary indirection in the handling of types.
- VPP::OptimizeSimpleTables
  - This pass looks for tables that do not have a match component and effectively wrap a simple action call and remove the table, replacing it with the simple action call.
- VPP::MakeFreeVarsExplicit
  - Adds bindings for variable references in actions that refer to the outer scope. This allows actions to be raised to top-level C style functions.

P4 VPP Compiler Passes

- VPP::MakeStackOperationsExplicit
  - Converts object-oriented style P4 stack operators into extern function calls. Also adds explicit next field to stack.
- VPP::EliminateSlice
  - Eliminates the P4 bit slice operation by translating them into C shift and mask operations
- VPP::EliminateConcat
  - Eliminates the P4 bit concat operation by translating them into C shift and and operations
- VPP::CFG
  - Builds a control-flow graph for the P4 program. This is currently not used, but will be important as we start to look at optimizations. It will also need to have the package information worked in.
- VPP::BuildCReferences
  - Determines where parser, control, and action parameters and variable declarations are pointers, and replaces references to these variables and fields with appropriate operations: pointer member (->), address of (&), and dereference (*).

P4 VPP Compiler Passes

- VPP::AddParens
  - Attempts to add the appropriate parenthesis to expressions to preserve their meaning in the original P4 program.
- VPP::Codegen
  - Generates C code from the IR. Uses the package determined in the first post-midend pass to write the main VPP node function. Generally this translation is fairly straight forward, but is complicated around tables, parser extracts, header and header union operations, and arbitrary integer handling.

P4 to C translation strategy

- Most P4 expressions translate directly to C
  - Operations that involve non-standard bit widths require additional functions be generated to handle them
  - Bit slice and concatenate operations are translated into shift and and operations
- Header types are translated into C structs with non-C-sized fields packed into C-sized fields
  - Fields that are multiple of 8-bits are translated into arbitrary precision integers (apints) which are structs containing an appropriately sized byte-array. Fields that are not byte aligned are combined with surrounding fields to create a single field that can be represented using either a C type or an apint.
- Headers are represented as pointers into the packet (valid checks for non-null)
- Structs are translated into C structs, without any packing
  - C-sized fields are represented as C types, fields that are a multiple of 8-bits are translated into apints, and other fields are rounded up to the nearest C type (if they are under 64-bits) or the nearest 8-bit multiple.
- Header unions are treated a C unions with an extra “raw” field used to determine “valid”

P4 to C translation strategy

- Header stacks become a C struct containing an array and a next index field
  - The next index field is needed to support P4’s next, nextIndex, last, push_front, and pop_front operations
- Actions, Controls, and Parsers are translated into C functions
  - Controls and Parsers are always defined at the top-level, so they have no additional free variables.
  - Actions may have additional parameters added from their surrounding scope.
- Parser states are translated into labels, with transitions translated into gotos
  - The mid-end inlines all called parsers, so we can always generate a single parser function with labels and gotos that implement the function.
- A struct type is created for each table to represent the results:
  - The struct contains an enum representing the action to be taken, a boolean indicating if this is a “hit” (vs. a default), and a union containing the action data for the configured action

P4 to C translation strategy

- Table application is translated into an explicit switch on the results of a table match
  - Each action is called in the switch, and if the original apply was within a switch the additional expressions are added to the branches of the switch.
- A VPP API message is also generated for each table
- Note: the table implementation is just a stub, every table apply results in the default action

Extern handling strategy

- The currently supported externs packet_in and verify are handled directly in the code generator
    - Applications of the packet_in methods and verify function are identified and the code generator splats out code specific to each method, based on the arguments to the method. Unfortunately, this is not a very scalable way to handle this.
- Strategy moving forward
    - Externs that can be implemented in terms of the P4 IR, can simply be expanded in place. Externs that rely on features within VPP can be expanded into extern function calls that we will treat as compiler intrinsics. In some cases we may also decide we want some intrinsics specifically handled within the code generator.

Package handling strategy

- Packages are identified based on name.
    - When there is an expected type for the package, the type is checked to ensure that the P4 program has not defined its own package by the name of a known package.
- Package information provides the mapping to wire the P4 program into P4
    - Some features wire together in obvious ways. For instance, packet_in and its operations become operations on the vlib buffer structure. Other features require information from the package. For instance, incoming metadata is specific to each package and needs to come from VPP. Feature support for things like multicast groups or the egress port must also be mapped to VPP.
- Each package knows how to write out the body of the node. In time this will likely need to be revisited as we begin to unroll loops and/or move to multi-node implementations.

V1Switch package handling

- The V1Switch specifies a parser, a verify checksum control, an ingress control, an egress control, a compute checksum control, and a deparser control.

- The V1Switch also specifies a standard_metadata that is passed as an inout argument to the parser, ingress control, and egress control.

  - This metadata specifies some ingress metadata and also allows specification of the egress port, multi cast group, drop field, and a number of other fields used to support clone, resubmit, and recirculate (which are not currently supported). In some cases these variables overlap externs, which may require some duplication to follow the rule that externs may not manipulate local variables.

V1Switch package handling

- The following pseudo code is how this code is translated into VPP
  - main(parser(), verifyChecksum(), ingress(), egress(), computeChecksum(), deparser()) becomes
  - standard_metadata_t standard_metadata; headers H; metadata M;
  - <initialize standard_metadata>
  - parser(b0, &H, &M, &standard_metadata);
  - verifyChecksum(&H, &M);
  - ingress(&H, &M, &standard_metadata);
  - if (standard_metadata.drop == 0) {
  - <initialize egress_port from egress_spec, if necessary>
  - egress(&H, &M, &standard_metadata);
  - computeChecksum(&H, &M);
  - deparser(&H);
  - }

Note: We should probably be checking the drop here after egress as well, since it could also be set there, and possibly after parser.

Also worth noting parser errors currently don't have a place to be