The architecture of the P4_{16} compiler

May 17, 2017 - P4 workshop, Stanford

Mihai Budiu, VMware Research
Chris Doss, Barefoot Networks
P4\textsubscript{16}

- Newest version of the P4 language (finalized yesterday!) [https://github.com/p4lang/p4-spec/tree/master/p4-16/spec](https://github.com/p4lang/p4-spec/tree/master/p4-16/spec)
- This talk is about the (reference implementation) compiler for P4\textsubscript{16}
- Compiles both P4\textsubscript{14} (i.e., P4 v1.0 and P4 v1.1) and P4\textsubscript{16} programs
- Apache 2 license, open-source, reference implementation
- [http://github.com/p4lang/p4c](http://github.com/p4lang/p4c)
Compiler goals

- Support current and future versions of P4
- Support multiple back-ends
  - Generate code for ASICs, NICs, FPGAs, software switches and other targets
- Provide support for other tools (debuggers, IDEs, control-plane, etc.)
- Open-source front-end
- Extensible architecture (easy to add new passes and optimizations)
- Use modern compiler techniques
  (immutable IR*, visitor patterns, strong type checking, etc.)
- Comprehensive testing

*IR = Intermediate Representation
Compiler data flow

\[\text{P4}_{14} \rightarrow \text{P4}_{14} \text{ parser} \rightarrow \text{convert} \rightarrow \text{IR} \rightarrow \text{frontend} \rightarrow \text{IR} \rightarrow \text{mid-end} \rightarrow \text{mid-end} \rightarrow \text{BMv2 back-end} \rightarrow \text{mid-end} \rightarrow \text{your own backend} \rightarrow \text{mid-end} \rightarrow \text{C code} \rightarrow \text{JSON} \rightarrow \text{target-specific code}\]
Compiler structure

P4 $\rightarrow$ Front-end $\rightarrow$ IR $\rightarrow$ Mid-end $\rightarrow$ IR $\rightarrow$ Back-end

- Target-independent
  - ~25 distinct passes
  - Same IR

- IR with target-specific extensions
  - ~25 distinct passes
  - Target-specific
Structure

Library of pre-built passes

Fixed

Mix and match

Custom

P4

Front-end

IR

Mid-end

IR

Back-end

libFrontEnd.a

main()

Simplify IR eliminating constructs gradually
Implementation details

- Common infrastructure for all compiler passes
  - Same IR and visitor base classes
  - Common utilities (error reporting, collections, strings, etc.)
- C++11, using garbage-collection (-lgc)
- Clean separation between front-end, mid-end and back-end
  - New mid+back-ends can be added easily
- IR can be extended (front-end and back-end may have different IRs)
- IR can be serialized to/from JSON
- Passes can be added easily
Intermediate Representation (IR)

- Immutable
  - Can share IR objects safely
  - Even in a multi-threaded environment
  - You cannot corrupt someone else’s state
- Strongly-typed (hard to build incorrect programs)
- DAG structure, no parent pointers
- Manipulated by visitors
- IR class hierarchy is extensible
Visitor pattern

  
  “In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.”

- “Structure” = IR
- “Algorithms” = program manipulations
IR rewriting using visitors

Input DAG

Modified DAG

New DAG

Output DAG

Dead code

Modified DAG 

Visitor
IR definition language compiled to C++

```cpp
interface IDeclaration { ... }

abstract Expression { ... }

abstract Statement : StatOrDecl {}

class AssignmentStatement : Statement {
  Expression left;
  Expression right;
  print{ out << left << " = " << right; }
}
```
IR ↔ P4

• Front-end and mid-end maintain invariant that IR is always serializable to a P4 program
• Simplifies debugging and testing
  • Easy to read the IR: just generate and read P4
  • Easy to compare generated IR with reference (testing)
  • Compiler can self-validate (re-compile generated code)
  • Dumped P4 can contain IR representation as comments

• IR always maintains source-level position
  • can emit nice error message anywhere
Learning the IR by example

• *Front-end and mid-end passes can all dump IR back as P4 source with IR as comments*

```c
/*
<P4Program>(18274)
 <IndexedVector<Node>>(18275) */
/*
<Type_Struct>(15)struct Version */
struct Version {
/*
 <StructField>(10)major/0
 <Annotations>(2)
 <Type_Bits>(9)bit<8> */
  bit<8> major;
...
```
v1model.p4: A P4$_{14}$ switch model

- A P4$_{16}$ switch architecture that models the fixed switch architecture from the P4$_{14}$ spec
- Provides backward compatibility for P4$_{14}$ programs
Testing the compiler

• Dump program at various points and compare with reference
• Compare expected compiler error messages (on incorrect programs)

• Recompile P4 generated by compiler
• Run v1model.p4 programs using BMv2 on packet traces and compare to expected output
Lessons

• $P4_{16}$ is a simple language, ... but the P4 environment is complicated
  • Supports arbitrary architectures
  • Arbitrary functionality in the architecture
  • Arbitrary extensions ($\texttt{extern}$ blocks)

• $P4_{16}$ is designed for extensibility
  • Compilers must support extensibility while preserving stability

• Modularity/extensibility seems to work
  • At least 5 existing back-ends, for software, simulators, FPGAs, ASICs

• Specification — Implementation

• Great community: thank you all!