CS434/534: Topics in Network Systems

High-Level Programming for Programmable Networks: Compiler and Network OS

Yang (Richard) Yang
Computer Science Department
Yale University
208A Watson
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Outline

- Admin and recap
- Datapath programming
  - Reactive, high level logic driven user programming
  - Reactive, high level automatic programming
  - Proactive, high level automatic programming (aka compiler)
- Controller software framework (network OS) supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
    - overview
    - basic Paxos
Recap: Basic Ideas of Automatic Datapath Generation

- **Insight**
  - Although the decision function $f$ does **not** specify how flow tables are configured, if for a given decision (e.g., drop), one can know the dependency on the input (pkt attributes) of the decision, one can construct the flow table.

- **Requirement**
  - Program $f$ uses a library to access pkt attributes and environment state variables, to track dependency access.

- **Basic data structures and algorithms**
  - Trace tree: tracking dependency on pkt attributes
  - Dependency table: tracking dependency on environment states

- **Problems**
  - Single table; latency
Recap: Proactive Compiler

- Proactive, fully generate multi-table datapath pipeline
- Basic idea: A function $f$ consists of a sequence of instructions $I_1, I_2, \ldots, I_n$, and one can (conceptually) consider each instruction in the dataflow as a table

```java
Map<MAC, Switch> hostTable;
0. Route onPacketIn(Packet p) {
1. Switch srcSw = hostTable.get(p.ethSrc());
2. Switch dstSw = hostTable.get(p.ethDst());
3. Route aRoutes = AllPairRoutes();
4. return aRoutes.get(srcSw, dstSw);
}
```

Ex: instruction data flow?
Recap: Map Each Instruction to a Flow Table

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw1, 33->sw2,others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}
Set openPorts; // {22, 53, 80, 8000, 8080, 9090}

void Route onPacketIn(Packet p) {
    if (p.dstPort < 1025) { // privileged port, SW_FW
        dstSw = SW_FW;
        dstCond = V;
    } else if (openPorts.contains(p.dstPort)) { // spec user
        dstSw = hostTable[p.dstMac];
        dstCond = condTable[p.dstMac];
    } else {
        return Drop;
    }
    // All pairs paths for all dstConds
    allPaths = AllPairsCond.execute(...);
    srcSw = hostTable[p.srcMac];
    return allPaths[dstCond].get(srcSw, dstSw); }

Recap: High-Level Pipeline

L1: p.dstPort < 1025

L2: dstSw
L3: dstCond
L9: allPairPaths
L11: egress
L10: srcSw
L5: dstSw
L5: dstCond
L8: Drop

L4: openPort.contains [ p.dstPort ]

N

Y
Outline

- Admin and recap
- Datapath programming
  - Reactive, high level logic driven user programming
  - Reactive, high level automatic programming
  - Proactive, high level automatic programming (aka compiler)
    - instruction analysis (symbolic analysis)
Observation: There are compact-mapable (CM) instruction: Despite large input domain, small # of rules to express.

<table>
<thead>
<tr>
<th>p.dstPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reg(_g_1) = true jump L2</td>
</tr>
<tr>
<td>2</td>
<td>Reg(_g_1) = false jump L2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>(2^{16}-1)</td>
<td>Reg(_g_1) = false jump L2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0000 00xx</td>
<td>reg(_g_1) = true jump L2 // 0-1023</td>
</tr>
<tr>
<td></td>
<td>xxxx xxxx</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0000 0100</td>
<td>reg(_g_1) = true jump L2 // 1024</td>
</tr>
<tr>
<td></td>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>xxxx xxxx</td>
<td>reg(_g_1) = false jump L2</td>
</tr>
<tr>
<td></td>
<td>xxxx xxxx</td>
<td></td>
</tr>
</tbody>
</table>

**L1.** \(g_1 = p\text{.}dstPort < 1025\);
**More Example of CM Statements**

\[ y = p.ethSrc == p.ethDst \]

<table>
<thead>
<tr>
<th>p.ethSrc</th>
<th>p.ethDst</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>y=true</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>y=false</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2^{48}-1</td>
<td>2^{48}-1</td>
<td>y=true</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p.ethSrc</th>
<th>p.ethDst</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>***0</td>
<td>***1</td>
<td>false</td>
</tr>
<tr>
<td>***1</td>
<td>***0</td>
<td>false</td>
</tr>
<tr>
<td>*<em>0</em></td>
<td>*<em>1</em></td>
<td>false</td>
</tr>
<tr>
<td>*<em>1</em></td>
<td>*<em>0</em></td>
<td>false</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>true</td>
</tr>
</tbody>
</table>

Offline exercise: \[ y = p.ethSrc \neq p.ethDst \]
Observation: Despite large input domain, system state provides (small) potential values.

<table>
<thead>
<tr>
<th>p.dstPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reg$_{g2}$ = false; jump L6</td>
</tr>
<tr>
<td>22</td>
<td>Reg$_{g2}$ = true; jump L6</td>
</tr>
<tr>
<td>$2^{16}-1$</td>
<td>Reg$_{g1}$ = false, jump L6</td>
</tr>
</tbody>
</table>

Compact encoding using system state

L5: g2 = openPorts.contains(p.dstPort);
    // openPorts = {22, 53, 80, 8000, 8080, 9090};

Naïve enumeration
Q: What are some instructions that are fundamentally hard to express by lookups?
Non-Compact Instruction

- srcMac % n, where n is not a power of 2 has no compact representation.
Ex: State and Issue After Symbolic Analysis+Direct System State Plug in

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw1, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}
Set openPorts; // {22, 53, 80, 8000, 8080, 9090}

Route onPacketIn(Packet p) {
  L1 if (p.dstPort < 1025) { // privileged port, SW_FW
    L2: dstSw = SW_FW;
    L3: dstCond = V;
    L4: }
  L5: } else if (openPorts.contains(p.dstPort)) { // spec user
    L6: dstSw = hostTable[p.dstMac];
    L7: dstCond = condTable[p.dstMac];
  L8: } else {
    L9: return Drop; }
    // All pairs paths for all dstConds
  L10: allPaths = AllPairsCond.execute( ... );
  L11: srcSw = hostTable[p.srcMac];
  L12: return allPaths[dstCond].get(srcSw,dstSw); }
Symbolic analysis/direct state-plugin cannot generate tables for some instructions.

Tables computed in isolation can have unnecessary rules.
Discussion: How to Handle the Problems?

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw1, 33->sw2,others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}
Set openPorts; // {22, 53, 80, 8000, 8080, 9090}

Route onPacketIn(Packet p) {
    L1  g1 = p.dstPort < 1025;
    L2  if ( g1 ) {
        L3:  dstSw = SW_FW;
        L4:  dstCond = S;
    } else {
        L5:  g2 = openPorts.contains(p.dstPort); // not compact
        L6:  if ( g2 ) {
            L7:  dstSw = hostTable[p.dstMac];
            L8:  dstCond = condTable[p.dstMac];
        } else {
            L9:  return Drop; }
    }
    // All pairs paths for all dstConds
    L10: allPaths = AllPairsCond.execute( ... );
    L11: srcSw = hostTable[p.srcMac];
    L12: return allPaths[dstCond].get(srcSw,dstSw); // compact inputs?
}
Observation

- Both problems require computing valid inputs to instructions:
  - \( g2 = \) openPorts.contains(p.dstPort);
    If we know valid inputs to dstPort (<1025), we can prune the extra rules (22, 53, 80)
  - allPaths[dstCond].get(srcSw,dstSw);
    If we know the valid values of (dstCond, srcSw, dstSw) reaching this instruction, we can compute a compact flow table for it (not all potential values).
Discuss: How to Compute Valid Inputs (Reachable States) to Each Instruction

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw1, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

0. Route onPacketIn(Packet p) {
L1. Switch dstSw = hostTable.get(p.dstMac());
L2. Cond dstCond = condTable.get(p.dstMac());
L3. Route aRoutes = AllPairCondRoutes();
L4. return aRoutes.get(dstSw, dstCond);

<table>
<thead>
<tr>
<th>Design 1: takes advantage of CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>o Inputs to L4:</td>
</tr>
<tr>
<td>•</td>
</tr>
</tbody>
</table>

Q: Is this the best?
Naïve Approach can Compute Non-Valid Inputs

\[
q \\
L1: m_1 \rightarrow s_1, m_2 \rightarrow s_2, \ldots, m_n \rightarrow s_n \\
L2: m_1 \rightarrow c_1, m_2 \rightarrow c_2, \ldots, m_n \rightarrow c_n \\
L4 \text{ sees only } n \text{ (sw, cond) combinations, not } n^2 \text{ combinations} \\
\text{ (sw, cond) combinations, not } n^2 \text{ combinations} \\
\text{ This is called dependent inputs}
\]

```java
Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

0. Route onPacketIn(Packet p) {
L1. Switch dstSw = hostTable.get(p.dstMac());
L2. Cond dstCond = condTable.get(p.dstMac());
L3. Route aRoutes = AllPairCondRoutes();
L4. return aRoutes.get(dstSw, dstCond);
```
Basic Approach: The FlowExplore Algorithm

- Computes (I, state) to each instruction, using flow tables as edges
```java
0. Route onPacketIn(Packet p) {
L1. Switch dstSw = hostTable.get(p.dstMac());
L2. Cond dstCond = condTable.get(p.dstMac());
L3. Route aRoutes = AllPairCondRoutes();
L4. return aRoutes.get(dstSw, dstCond);
}
```
0. Route onPacketIn(Packet p) {
L1. Switch dstSw = hostTable.get(p.dstMac());
L2. Cond dstCond = condTable.get(p.dstMac());
L3. Route aRoutes = AllPairCondRoutes();
L4. return aRoutes.get(dstSw, dstCond);
}

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

dstMac is already part of state, no need to use flow table, use state propagation

L2, dstSw=sw1, dstMac=11
L2, dstSw=sw2, dstMac=22
L2, dstSw=sw2, dstMac=33
L2, dstSw=swu, dstMac=*
Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

0. Route onPacketIn(Packet p) {
    L1. Switch dstSw = hostTable.get(p.dstMac());
    L2. Cond dstCond = condTable.get(p.dstMac());
    L3. Route aRoutes = AllPairCondRoutes();
    L4. return aRoutes.get(dstSw, dstCond);
}

dstMac is a dead variable and hence should be removed.
Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable;  // {11->S, 22->C, 33->C, others UK}

0. Route onPacketIn(Packet p) {
   L1. Switch dstSw = hostTable.get( p.dstMac() );
   L2. Cond dstCond = condTable.get( p.dstMac() );
   L3. Route aRoutes = AllPairCondRoutes();
   L4. return aRoutes.get(dstSw, dstCond);
}

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstMac</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>dstSw=sw1</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>dstSw=sw2</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>dstSw=sw2</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>dstSw=swu</td>
</tr>
</tbody>
</table>

Same state and hence should merge.
0. Route onPacketIn(Packet p) {
    L1. Switch dstSw = hostTable.get(p.dstMac());
    L2. Cond dstCond = condTable.get(p.dstMac());
    L3. Route aRoutes = AllPairCondRoutes();
    L4. return aRoutes.get(dstSw, dstCond);
}
0. Route onPacketIn(Packet p) {
    L1. Switch dstSw = hostTable.get(p.dstMac());
    L2. Cond dstCond = condTable.get(p.dstMac());
    L3. Route aRoutes = AllPairCondRoutes();
    L4. return aRoutes.get(dstSw, dstCond);
}

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstMac</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>dstCond=S</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>dstCond=C</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>dstCond=C</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>dstCond=UK</td>
</tr>
</tbody>
</table>

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

dstMac is binded, Is there an ancestor binds to dstMac=11? Yes. Blocked
Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

0. Route onPacketIn(Packet p) {
    L1. Switch dstSw = hostTable.get(p.dstMac());
    L2. Cond dstCond = condTable.get(p.dstMac());
    L3. Route aRoutes = AllPairCondRoutes();
    L4. return aRoutes.get(dstSw, dstCond);
}

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstMac</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>dstCond=S</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>dstCond=C</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>dstCond=C</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>dstCond=UK</td>
</tr>
</tbody>
</table>

Diagram:

```
L1, EMPTY

L2, dstSw=sw1, dstMac=11
  L4, dstSw=sw1, dstCond=S
    L∞, aRoutes(sw1, S)

L2, dstSw=sw2, dstMac=22
  L4, dstSw=sw2, dstCond=C
    L∞, aRoutes(sw2, C)

L2, dstSw=sw2, dstMac=33
  L2, dstSw=sw2, dstMac=33
    L∞, aRoutes(sw2, C)

L2, dstSw=swu, dstMac=*  
  L2, dstSw=swu, dstMac=*  
    L∞, aRoutes(swu, UK)
```
Q: How to obtain the flow table of each instruction?
Assignment

- Compute the reachable inputs of the complete example program
Issues of PIT Pipelines

Both problems require evaluating merging of instructions (tables).
This example needs 10 tables, but a hardware may support only 2.

Unnecessary separate matching resources can be wasteful.

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw1, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}
Set openPorts; // {22, 53, 80, 8000, 8080, 9090}

Route onPacketIn(Packet p) {
L1 if (p.dstPort < 1025) { // privileged port, SW_FW
L2: dstSw = SW_FW;
L3: dstCond = S;
L4: } else if (openPorts.contains(p.dstPort)) { // spec user
L5: dstSw = hostTable[p.dstMac];
L6: dstCond = condTable[p.dstMac];
L7: } else {
L8: return Drop; }
  // All pairs paths for all dstConds
L9: allPaths = AllPairsCond.execute( ... );
L10: srcSw = hostTable[p.srcMac];
L11: return allPaths[dstCond].get(srcSw,dstSw); }

// All pairs paths for all dstConds
L9: allPaths = AllPairsCond.execute( ... );
L10: srcSw = hostTable[p.srcMac];
L11: return allPaths[dstCond].get(srcSw,dstSw); }

// All pairs paths for all dstConds
L9: allPaths = AllPairsCond.execute( ... );
L10: srcSw = hostTable[p.srcMac];
L11: return allPaths[dstCond].get(srcSw,dstSw); }
Algorithm Framework

Assume $P$ is the set of pipelines eval'd

$p_0 = \text{empty};$ resource$_0 = \text{INF}$

foreach $p$ in $P$

    if hw_oracle($p$) && resource($p$) < resource$_0$

        $p_0 = p;$ resource$_0 = \text{resource}(p)$

Q: How many pipelines in $P$ if $P$ is generated from full enumeration. Assume $N$ instructions and $\leq M$ tables?
\[ g_1 = \text{dstPort} < 1025 \]
\[ \text{dstSw} = \text{SW_FW} \]
\[ \text{dstCond} = \text{V} \]
\[ \text{egress} = \text{Drop} \]
\[ \text{dstSw} = \text{hostTable} (\text{dstMac}) \]
\[ \text{dstCond} = \text{condTable} (\text{dstMac}) \]
\[ \phi (\text{dstSw}) \]
\[ \phi (\text{dstCond}) \]
\[ \text{srcSw} = \text{hostTable} (\text{srcMac}) \]
\[ \text{egress} = f (\text{dstCond}, \text{dstSw}, \text{srcSw}) \]
Insight: Suppose we already have a table $T$ matching on $(\text{dstPort}, \text{dstMac})$. What should be handled by $T$?
Discussion and Next Step

- The compiler approach
  - Pros?
  - Cons?

- For those who are looking for a project, introduce flow algebra (vs relational algebra) to compute propagation and optimal design (a possible NSDI’20 submission)
Outline

- Admin and recap
- Datapath programming
- Controller framework supporting programmable networks
  - often referred to as network OS (NOS)
Discussion

- What components do we include in a software architecture for network OS (NOS) to support programmable networks?
ONOS System Architecture

Northbound - Application Intent Framework
(policy enforcement, conflict resolution)

Distributed Core
 scalability, availability, performance, persistence

Southbound
(discover, observe, program, configure)

OpenFlow
(pluggable, extensible)

More API in future release

Source: ON.LAB
OpenDaylight System Architecture

OpenDaylight Software Architecture

Network Devices
- Protocol Plugin
- Protocol Plugin
- Config Subsystem

Applications
- App/Service Plugin
- App/Service Plugin
- RESTCONF

Controller
- Model-Driven SAL (MD-SAL)
- Messaging
- Data Store

Remote Controller Instance

Remote Controller Instance
A More Abstract Architecture

Key goal: provide applications with high-level views and make the views highly available (e.g., 99.99%), scalable.

Key component - data store:
- Data model
- Data access model
- Data store availability

logically centralized
data store

Program

Network View
Service/Policy

NE Datapath
NE Datapath
Outline

- Admin and recap
- Datapath programming
- Controller framework supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
Data Model

- Not a fixed model/view, but models described by Yang and loaded into the system by OSGi
- Yang: a modeling language describing
  - data as a tree
  - RPCs and notifications
  - Based on RFC 6020
Yang Data Tree

Data Tree: Details

- Whole data store as a tree
- Two Logical Data Stores
  - config
  - operational
- InstanceIdentifier:
  - pointer to a node
Data Store Operations

- Operations must be included in transactions

```java
ReadWriteTransaction transaction = dataBroker.newReadWriteTransaction();
Optional<Node> nodeOptional;
nodeOptional = transaction.read(
    LogicalDataStore.OPERATIONAL,
    n1InstanceIdentifier);
transaction.put(
    LogicalDataStore.CONFIG,
    n2InstanceIdentifier,
    topologyNodeBuilder.build());
transaction.delete(
    LogicalDataStore.CONFIG,
    n3InstanceIdentifier);
CheckedFuture future;
future = transaction.submit();
```
Outline

- Admin and recap
- Datapath programming
- Controller framework supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed network OS
Setting

Discussion: How may a distributed NetOS look like?
A Common Approach: Replicated Log

- Replicated operations log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
- Failure model: fail-stop (not Byzantine), delayed/lost messages
Two general approaches to consensus:

- **Symmetric, leader-less:**
  - All servers have equal roles
  - Clients can contact any server
  - We will see Basic Paxos as an example

- **Asymmetric, leader-based:**
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
  - We will see Raft, which uses a leader
  - OpenDaylight is based on Raft
Warning

- Leslie Lamport, 1989
- Nearly synonymous with consensus

“The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-).” —NSDI reviewer

“There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system...the final system will be based on an unproven protocol.” —Chubby authors
Prepare for Next Class

- Stare at Paxos and play with examples
- Read Raft
- Start to think about projects