Autogenerating fast packet-processing code using program synthesis

Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, Srinivas Narayana
Programmable network devices today

- Lots of programmable network devices (e.g., switches, SmartNICs, NetFPGA)

- Easy to get started with programming these devices

- But hard to write fast packet-processing code that fits within resource constraints.
Today’s approach to writing fast programs

• Handcraft the fast programs
  • Requires familiarity with the underlying hardware architecture
  • Engineering a compiler takes a long time and large teams.
  • Difficult for new, niche, and evolving packet-processing hardware.

<table>
<thead>
<tr>
<th>Input A</th>
<th>Input B</th>
<th>Output C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Cache and memory hierarchy for SmartNICs and CPUs
Arithmetic logic unit (ALU) for switches
Lookup Table for FPGAs
Writing and compiling fast programs is hard

- Compilers must “fit” program to hardware. Example from Domino:

```c
void my_fun() {
    pkt.x = count;
    count += 1;
}
```

- The Domino compiler produces different outcomes (fit/not fit) on semantically-equivalent programs

```c
if (cond1 != 0) {
    pkt.member = 1;
    cond1 = 1;
} else {
    cond1 = 1;
}
```

- Such “butterfly effects” happen with P4 compilers as well.

- Choice of opcode
- ALU inputs
- Muxes, etc.
Proposal: Program synthesis for code generation

Program synthesis: Auto-generate a program from a specification

Search space represented by partial program

Completed program that satisfies specification

Specification

High-level language program

Return infeasible if there is none

Program synthesis as an enabling technology for packet-processing compilers
Example: Using SKETCH to do synthesis

<table>
<thead>
<tr>
<th>Specification</th>
<th>Partial program</th>
<th>Infeasible partial program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int spec(int x) { return x + x + x; }</code></td>
<td><code>int sketch1(int x) implements spec { return x * ??; }</code></td>
<td>No hole assignment</td>
</tr>
<tr>
<td></td>
<td><code>int sketch1(int x) implements spec { return x * 3; }</code></td>
<td>Hole representing unknown constant</td>
</tr>
</tbody>
</table>

Completed program

`int sketch2(int x) implements spec { return x + ??; }`
Why use synthesis for generating packet-processing code?

• For fast packet processing, there is value to generating high-quality code using exhaustive search based on synthesis:
  • RMT/PISA-style pipelined architectures: Optimize code for latency
  • NPU-style processor architectures: Optimize code for throughput

• Synthesis can potentially discover these high-quality assembly implementations
  • With lower engineering effort relative to handcrafted code generation rules
But, synthesis can be very time consuming

• The challenge: search space might be very large.

• Can leverage domain structure in packet processing:
  • Small (e.g., RCP) and simple (e.g., no loops/pointers) programs
  • Packet-processing programs don’t change that often.

• And improvements in synthesis technology
  • Many mature open-source synthesis tools (e.g., SKETCH, Rosette)
  • Synthesis benchmarks and competitions
Chipmunk: synthesis for switch code generation

Program written in Domino language (SKETCH specification) → SKETCH program Synthesis engine → Completed program output by SKETCH → PISA simulator machine code

Programmer

Compiler developer

Partial program representing PISA simulator’s capabilities
if (count == 10):
count = 0
pkt.sample = 1
else:
count++
pkt.sample = 0

Chipmunk in more detail

Program as a packet transaction in Domino

State Variable

Packet Field

Chipmunk

Machine code for PISA simulator
Evaluation: Baseline compiler

• We compare Chipmunk against Domino’s existing compiler
  • Uses handcrafted rules for translating Domino programs to simulator machine code

• Alternatives:
  • P4-14/P4-16 compilers from chip vendors (e.g., Barefoot)
    • Don’t yet support Domino’s transactional programming model
  • Open source p4c compiler
    • Supports software packet processing targets with a different architecture
Evaluation: Metrics and benchmarks

• Metrics: compilation result and resource usage.

• Picked benchmark programs written in the Domino language

• However, we can’t directly use these benchmarks.
  • Reason: These benchmarks were all written to ensure successful Domino compilation.
  • And we confirmed that Chipmunk can successfully compile them as well.
  • So comparing Chipmunk vs. Domino on these benchmarks won’t tell us anything interesting.

• Instead: If we mutate these programs in semantic-preserving ways, will Domino/Chipmunk still compile them?
  • Theoretically, yes: if a program compiles, an equivalent program should also compile
  • Empirically, maybe: how good are Domino and Chipmunk in finding machine code to implement a Domino program?
**Evaluation: compilation result on mutations**

<table>
<thead>
<tr>
<th>Program</th>
<th>Domino</th>
<th>Chipmunk</th>
<th>Chipmunk Compilation time (sec) 95% confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCP</td>
<td>10/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stateful Firewall</td>
<td>9/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampling</td>
<td>0/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLUE (increase)</td>
<td>0/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLUE (decrease)</td>
<td>0/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>10/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detecting new flows</td>
<td>0/10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detecting flow reordering</td>
<td>0/10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Evaluation: compilation result on mutations

<table>
<thead>
<tr>
<th>Program</th>
<th>Domino</th>
<th>Chipmunk</th>
<th>Chipmunk Compilation time (sec) 95% confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCP</td>
<td>10/10</td>
<td>10/10</td>
<td></td>
</tr>
<tr>
<td>Stateful Firewall</td>
<td>9/10</td>
<td>9/10 (timeout)</td>
<td></td>
</tr>
<tr>
<td>Sampling</td>
<td>0/10</td>
<td>10/10</td>
<td></td>
</tr>
<tr>
<td>BLUE (increase)</td>
<td>0/10</td>
<td>10/10</td>
<td></td>
</tr>
<tr>
<td>BLUE (decrease)</td>
<td>0/10</td>
<td>10/10</td>
<td></td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>10/10</td>
<td>10/10</td>
<td></td>
</tr>
<tr>
<td>Detecting new flows</td>
<td>0/10</td>
<td>10/10</td>
<td></td>
</tr>
<tr>
<td>Detecting flow reordering</td>
<td>0/10</td>
<td>10/10</td>
<td></td>
</tr>
</tbody>
</table>
## Evaluation: compilation result on mutations

Domino compilation time: Very quick; few seconds on most benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Domino</th>
<th>Chipmunk</th>
<th>Chipmunk Compilation time (sec) 95% confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCP</td>
<td>10/10</td>
<td>10/10</td>
<td>[ 11.29, 17.36]</td>
</tr>
<tr>
<td>Stateful Firewall</td>
<td>9/10</td>
<td>9/10 (timeout)</td>
<td>[ 1365.09, 2508.60]</td>
</tr>
<tr>
<td>Sampling</td>
<td>0/10</td>
<td>10/10</td>
<td>[ 7.93, 8.21 ]</td>
</tr>
<tr>
<td>BLUE (increase)</td>
<td>0/10</td>
<td>10/10</td>
<td>[ 10.35, 10.9 ]</td>
</tr>
<tr>
<td>BLUE (decrease)</td>
<td>0/10</td>
<td>10/10</td>
<td>[ 45.97, 51.25 ]</td>
</tr>
<tr>
<td>Flowlet switching</td>
<td>10/10</td>
<td>10/10</td>
<td>[ 3475.88, 4587.41 ]</td>
</tr>
<tr>
<td>Detecting new flows</td>
<td>0/10</td>
<td>10/10</td>
<td>[ 8.16, 8.46 ]</td>
</tr>
<tr>
<td>Detecting flow reordering</td>
<td>0/10</td>
<td>10/10</td>
<td>[ 8.00, 10.91 ]</td>
</tr>
</tbody>
</table>
Evaluation: resource consumption

![Graph showing resource consumption comparison between Domino and Chipmunk for different stages and ALUs per stage.]
Conclusion

• Program synthesis can play an important role in code generation for fast packet processing

• Program synthesis can enable fast packet processing like SAT/SMT have enabled network verification.

• Future work: code generation for other programmable substrates (e.g., SmartNICs, FPGAs, DPDK, etc.)