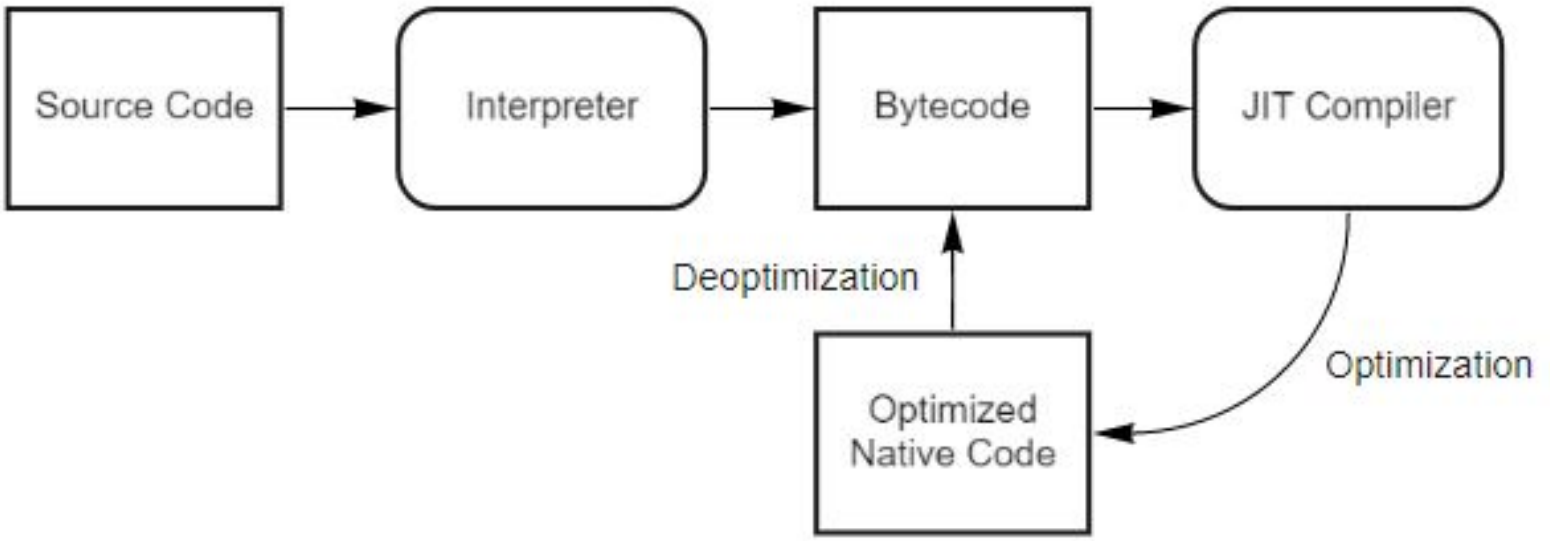
Visualizing JIT Compiler Graphs

HeuiChan Lim and Stephen Kobourov
The University of Arizona

September 16, 2021

Just-in-Time Compiler Systems

- JIT Compiler: turns bytecode into native code
- JIT compilers are needed to improve performance
- JIT compiler system: pipeline overview



Motivation

- Google Chrome, Microsoft Edge, Apple Safari, Mozilla Firefox
- Used by 2.65 billion, 600 million, 446 million, and 220 million

- JIT compiler bugs can lead to security vulnerabilities
- Such bugs can be used to hijack passwords and to navigate to other sites to execute malicious programs
- Need to quickly analyze, localize and fix JIT compiler bugs

Motivation

- Size and complexity of JIT-based systems, make it challenging to analyze and locate bugs quickly (e.g., Google Chrome has over 1 million lines of code)
- Traditional debuggers rely on text, even though the main feature of a JIT compiler is building a graph-like structure to translate bytecode into optimized machine code
- Prior work and available tools focus on the static code – not suitable for debugging JIT compilers, which generate code at run-time

Our Contributions

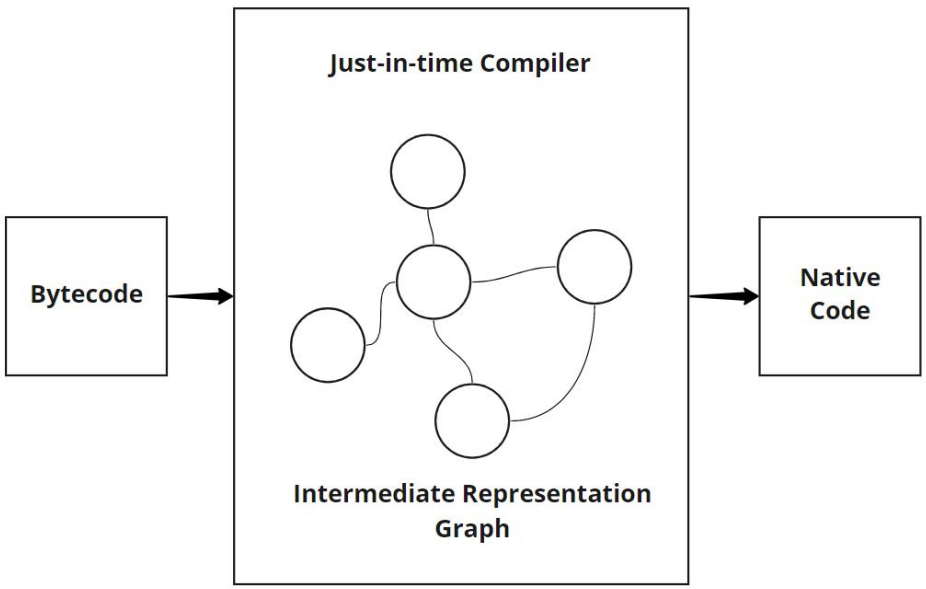
We designed, implemented and tested a new visualization tool:

- Part 1: relies on IR identification and generation techniques, described in detail by Lim and Debray [VEE'21]
- Part 2: merge multiple IR graphs into a single graph, simplify the merged graph, convert the simplified graph into a hypergraph, simplify the hypergraph, and visualize the hypergraph

Just-in-time Compiler

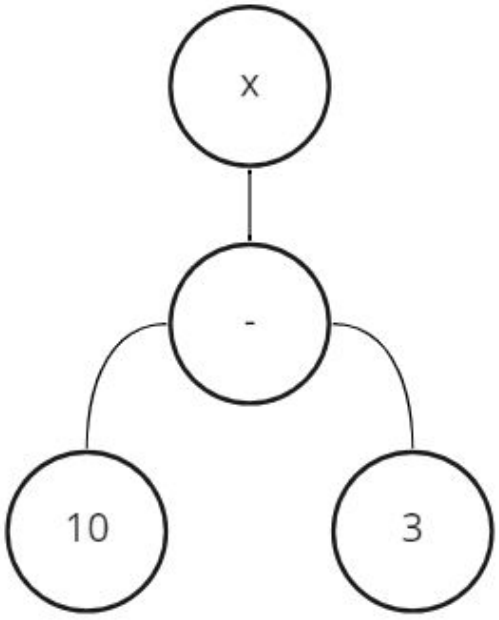
The two main jobs of a JIT compiler:

- IR Generation: convert bytecode into a graph
- IR Optimization: modify the graph

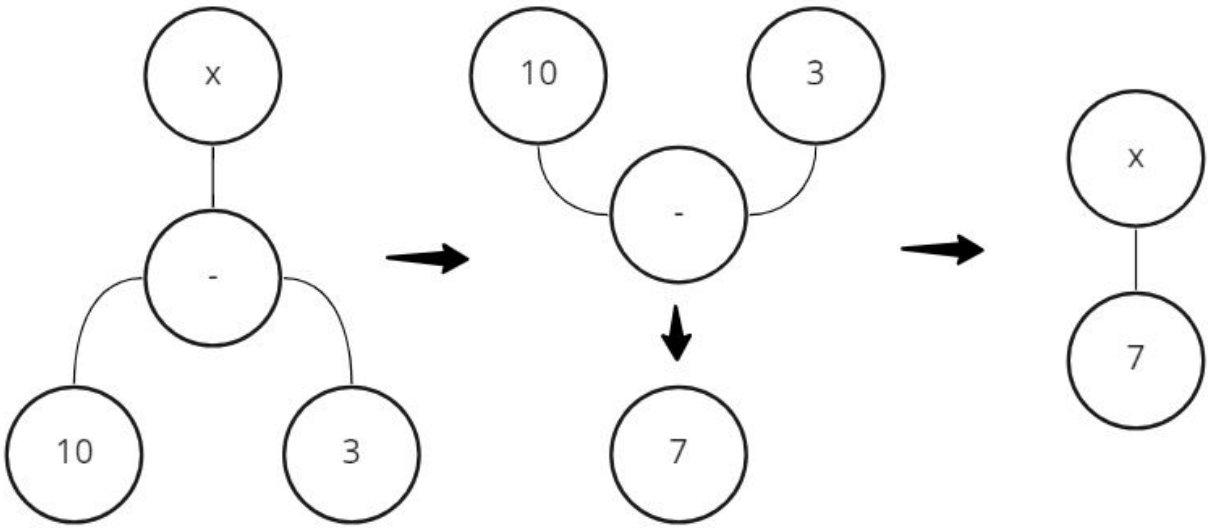


Intermediate Representation Generation

```
{
 int x = 10 - 3
}
```



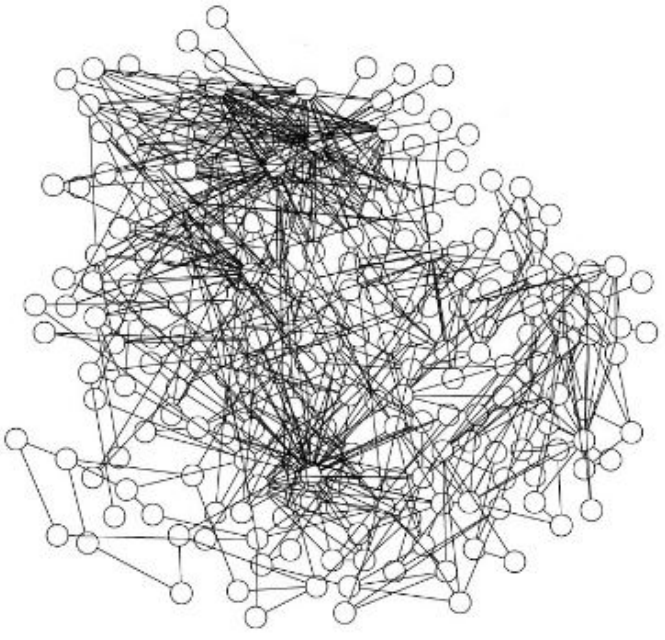
Intermediate Representation Optimization



Node Merge

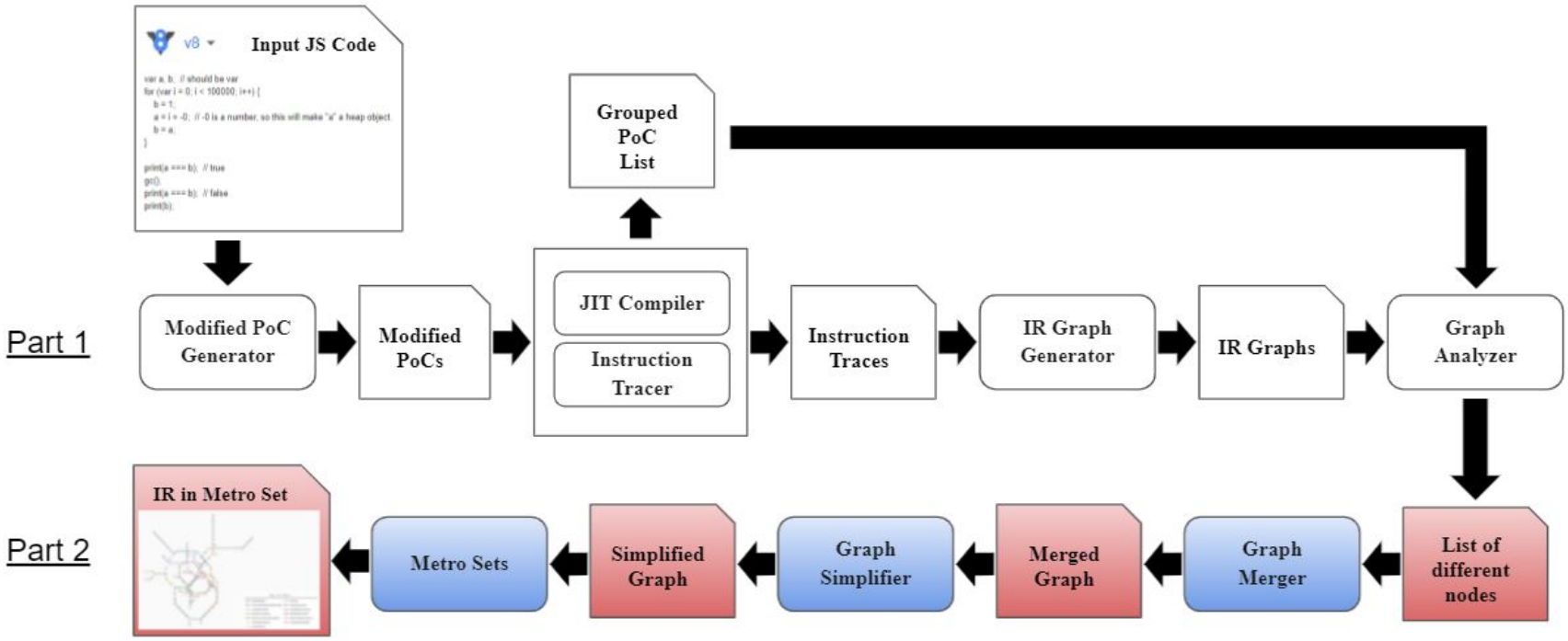
Theory vs. Practice

```
1 function foo($a,$b) {
2       $a = $a|0;
3       $b = $b|0;
4       var $sub = $a - $b;
5       return ($sub|0) < 0;
6 }
7
8 var result;
9 for (var i = 0; i < 10000; i++) {
10      result = foo(2147483647, -1);
11 }
```



Pipeline Overview

Part 1: Fault Localization (see Lim and Debray [VEE'21])



Part 2: Visualization of the IR

Overview of Fault Localization (Part 1)

Fault Localization Idea

- Randomly modify original input code to generate many similar IRs
- Some will still exhibit the buggy behavior; others won't
- The objective is to localize the buggy node, i.e., *v4*

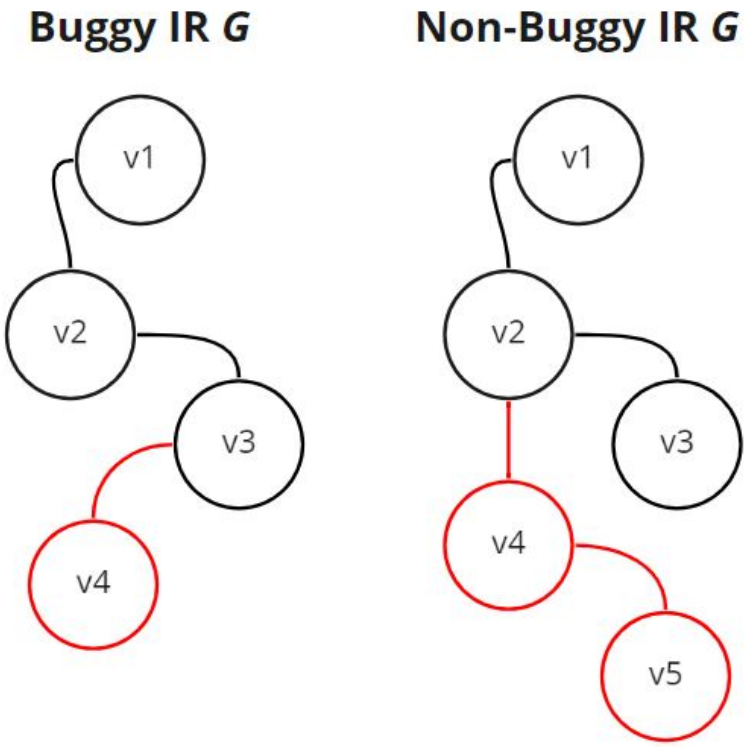
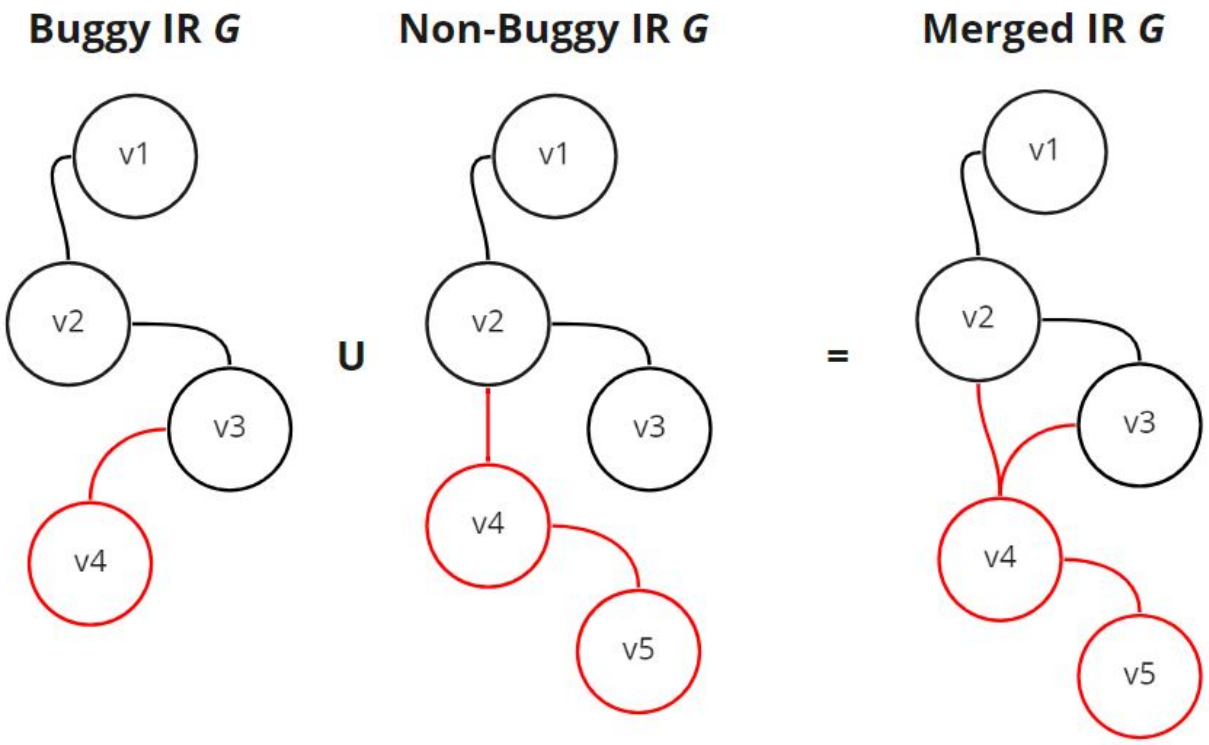


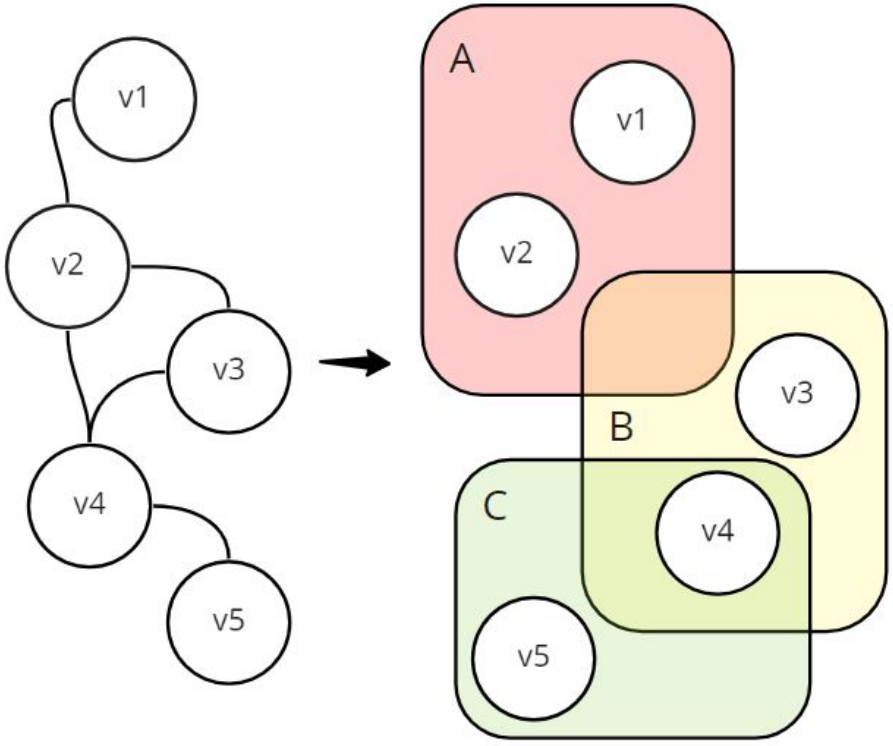
Figure 1: Graph Comparison

Merging IR Graphs (Part 2)



Interpreting the IR as Hypergraphs (Part 2)

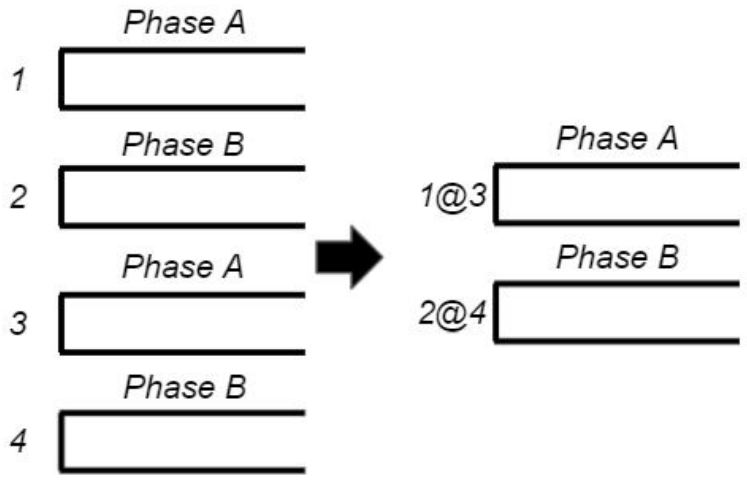
- Nodes are generated in one phase
- Hyperedges correspond to optimization phases



Hypergraph Simplification (Part 2)

2-Step Simplification

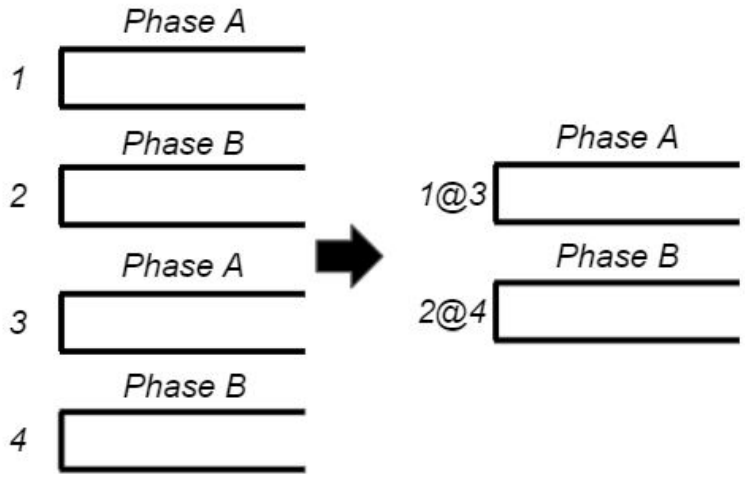
- **Hyperedge Reduction**
  - The same phases can be executed multiple times
  - Merging phases by operation
- Node Reduction

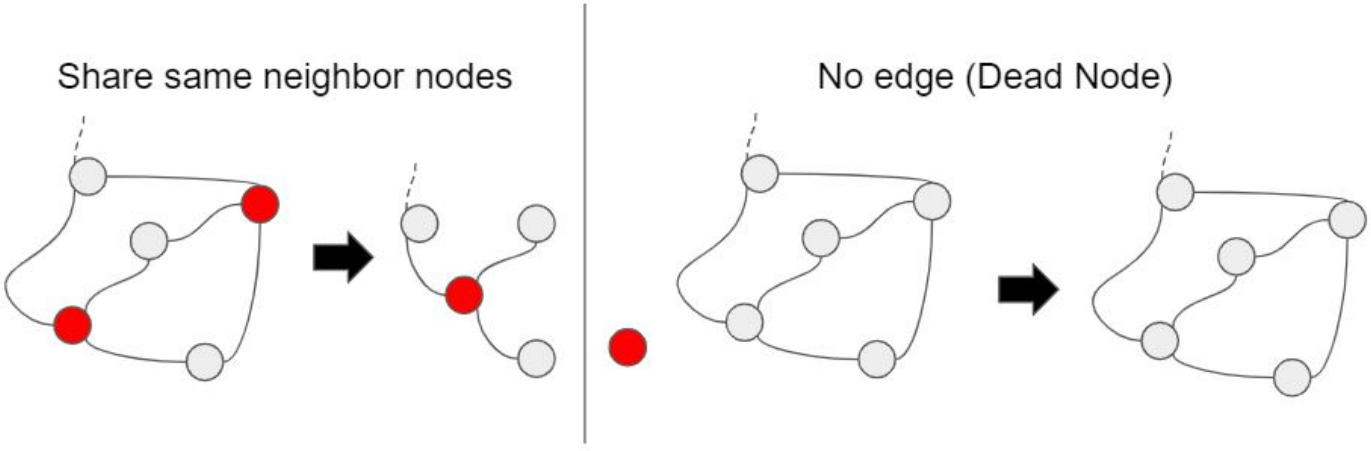


Hypergraph Simplification (Part 2)

2-Step Simplification

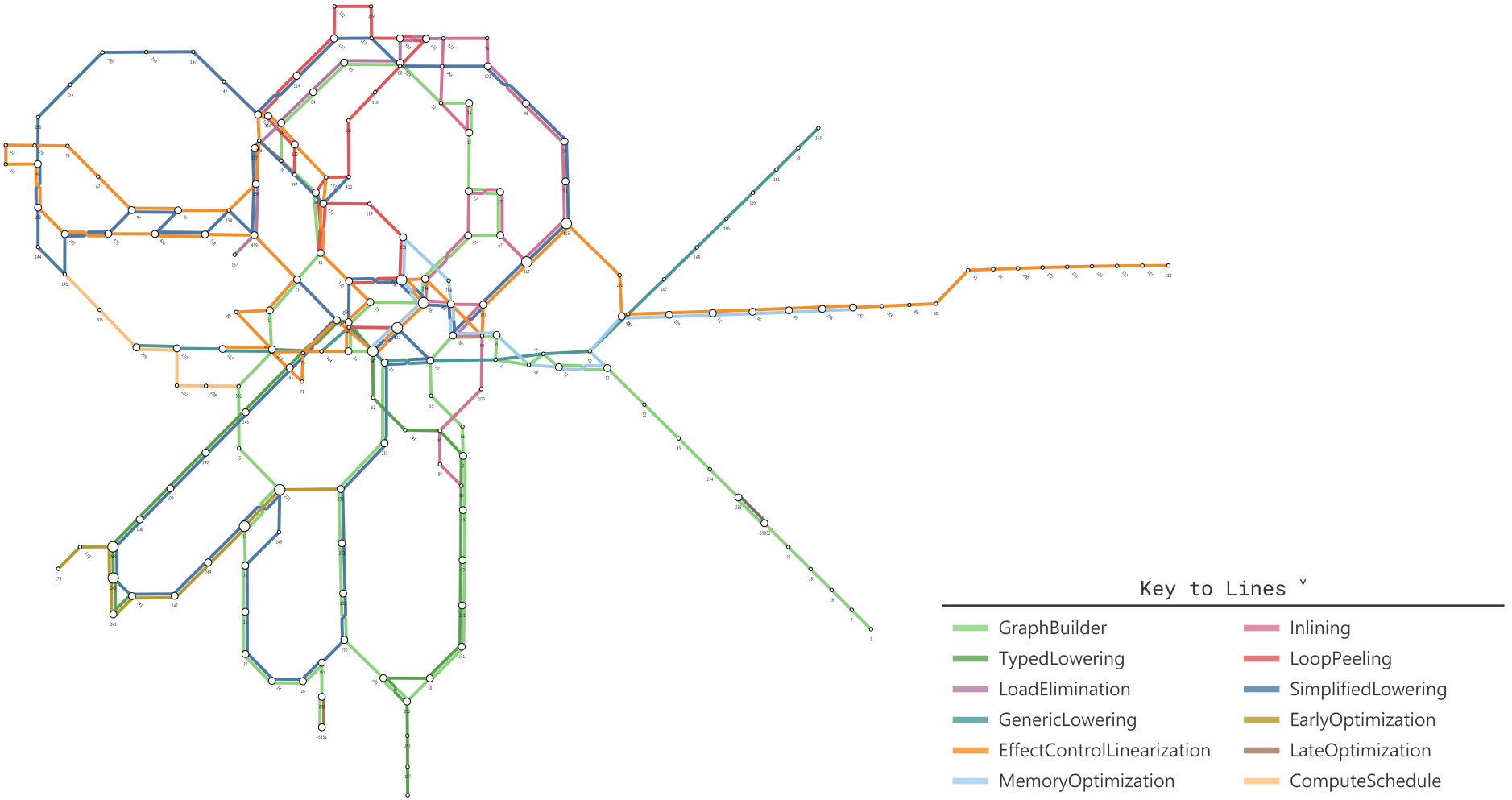
- Hyperedge Reduction
  - The same phases can be executed multiple times
  - Merging phases by operation
- **Node Reduction**





Visualizing the Hypergraph with MetroSets

- Identify hyperedges (lines) with suspicious nodes (stations)
- Identify hyperedges that intersect with a suspicious hyperedge
- Hovering over a node shows phase, opcode, address, etc.
- The phase tells us where a node was generated
- And we can also see the phases where it was optimized

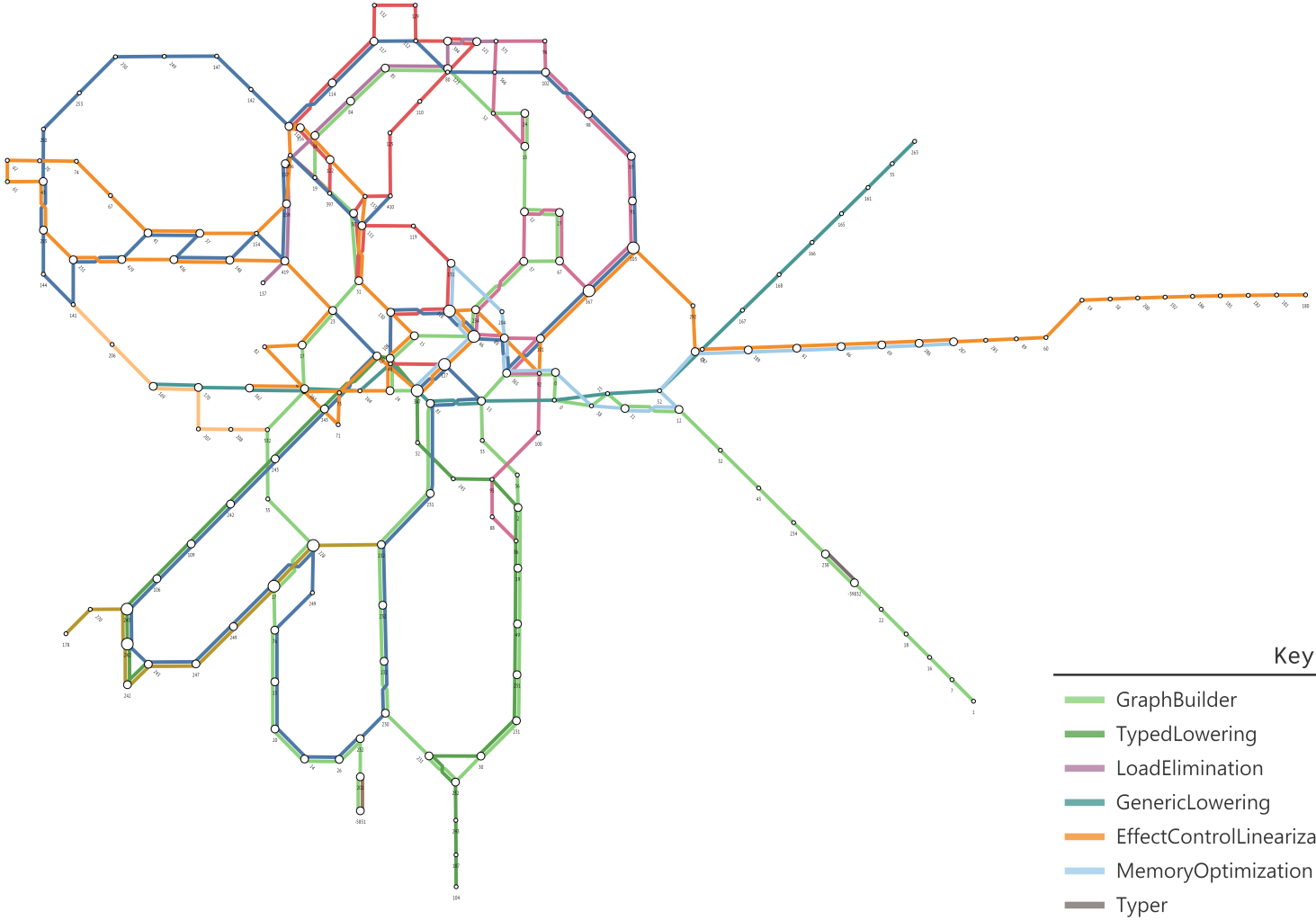


Example

- Google Chromium bug report 5129
- This compiler version has a bug in phase EarlyOptimization
- We generate 19 variants of the program and run all 20
- The instruction traces generate the IR graph below

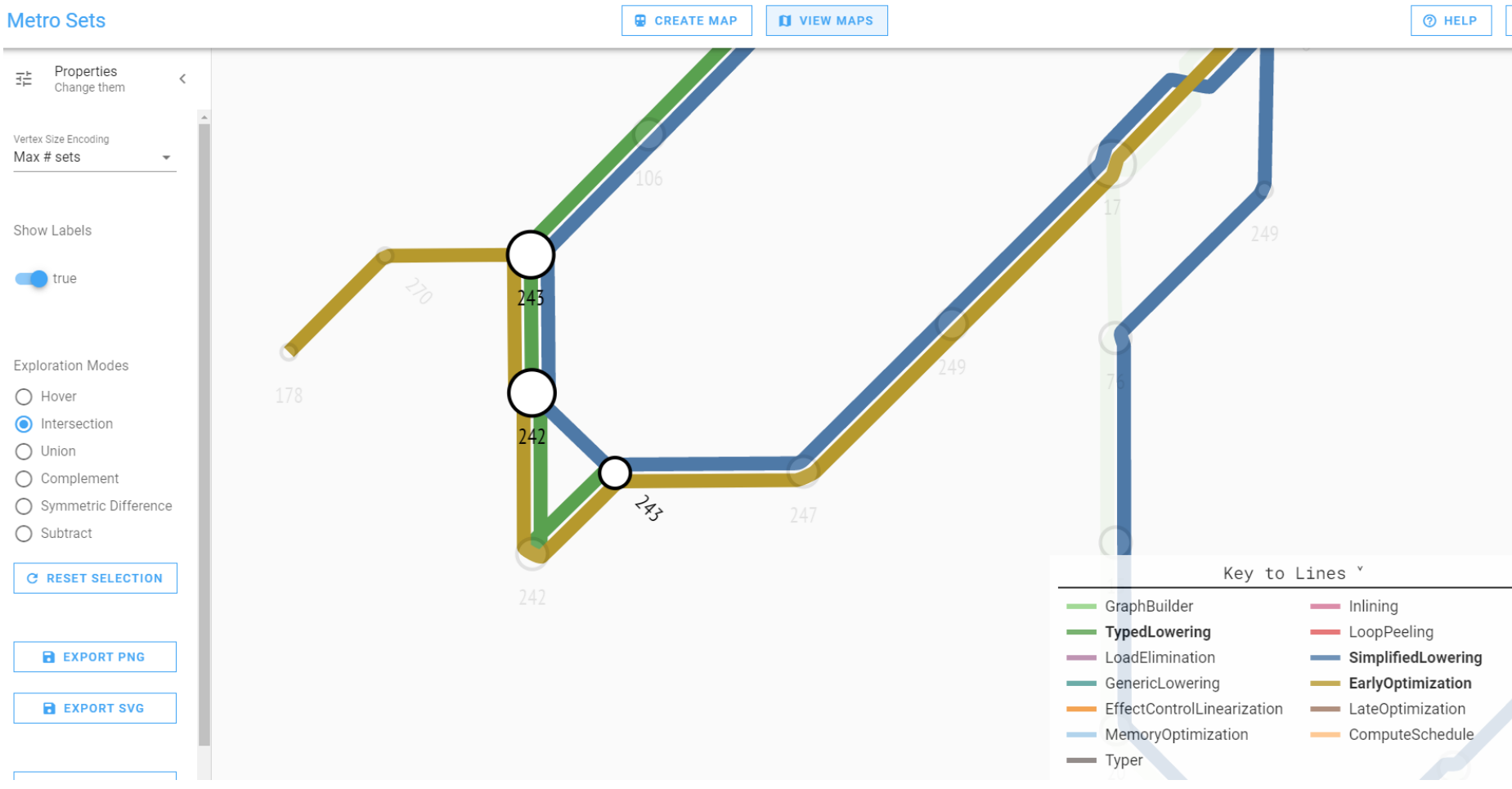
"What optimizations generated the machine code?"

The map and "Key to Lines" legend show all optimization phases



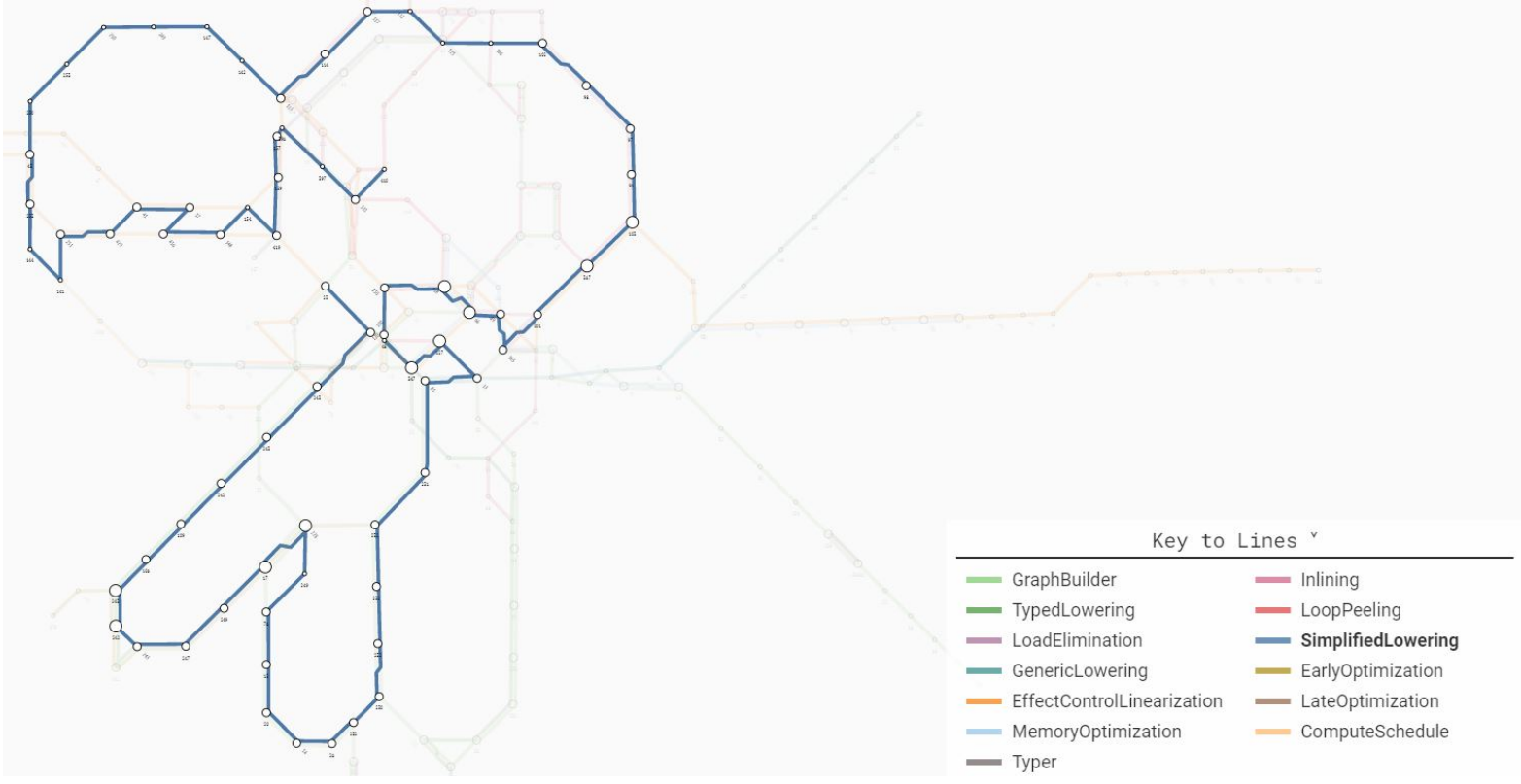
"How are optimization phases related?"

We can examine the corresponding lines and use the interactive exploration modes (**intersection**, union, complement, etc.) to see the relationships among the phases.



"Which optimization phase was most active?"

We can visually identify the longest line, or hover over each line and see the number of nodes in it

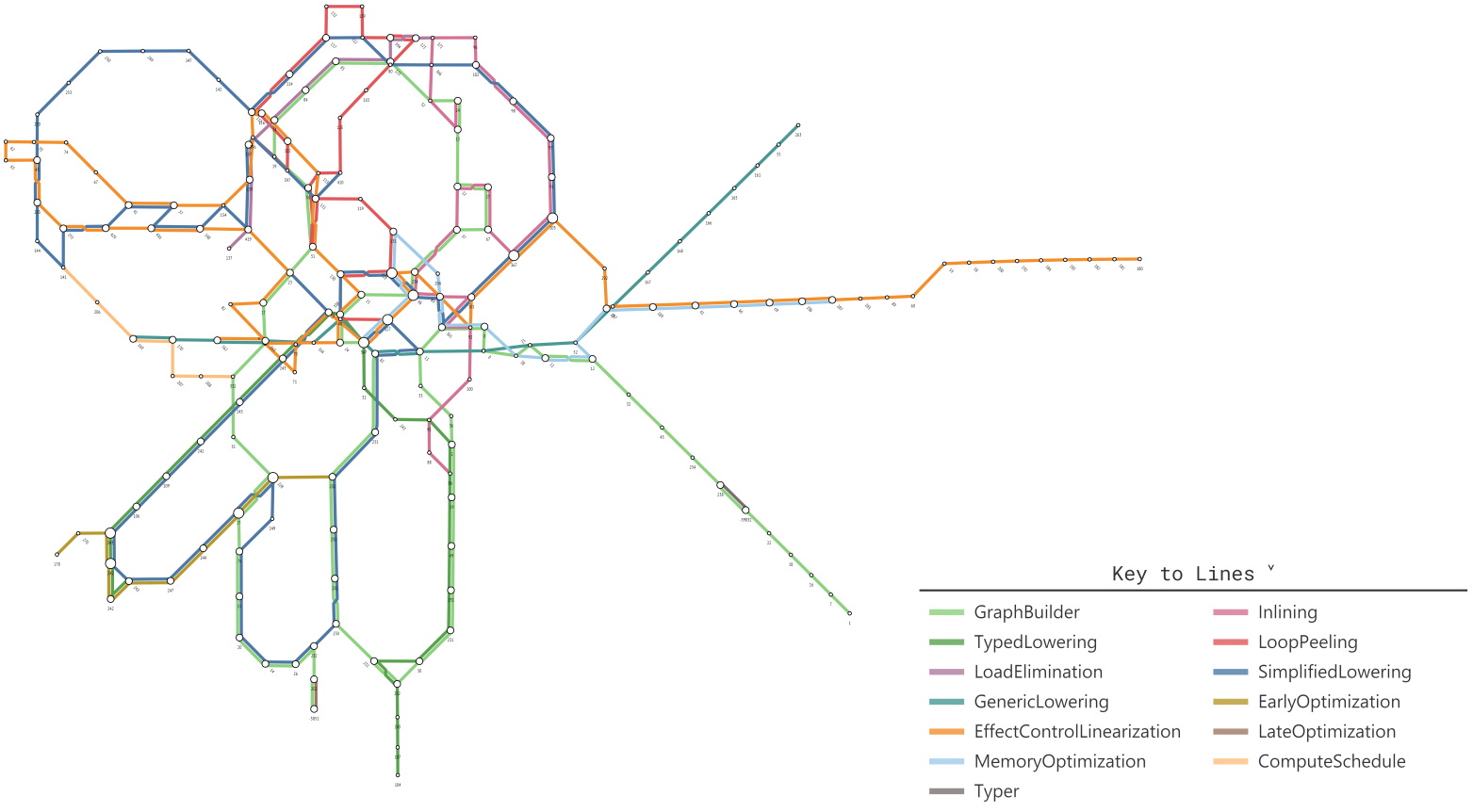


"What optimizations affected a specific node"

We can hover over the node of interest, which grays out the lines that don't contain the node. We can then examine each of the corresponding lines and look at the displayed node attributes.

"Which optimization phases are likely to be buggy?"

- Find parts that differ in the IR graphs with the bug and those without
- I.e., a program is buggy because it has extra/missing optimizations, and this information is captured in the IRs
- Any line that has many non-original IRs represents a significant difference between buggy and non-buggy programs



Discussion & Future Work

- It seems possible to represent (usually confusing) IRs using the metro map metaphor
- Functional prototype of the system: `https://hlim1.github.io/JITCompilerIRViz/`
- After simplifications (of the IR graphs, the merged graph, and the hypergraph) some useful information might be lost
- A two-level visualization which shows the simplified hypergraph as an overview but also provides all details on demand will likely be useful
- To identify suspicious phases we now hover over each line; it would be better to localize suspicious phases and highlight such lines
- More information about each node can be provided: "why is this node connected to another?", "what optimization (i.e., removed, added) created this node?", etc.