Design for FAST Virtual eXecution Environment

Kai Gao
Tsinghua University

The *Virtual eXecution Environment* is a crucial part of the FAST framework. It can execute a function instance and report the fine-grained data dependencies automatically. This document describes the current design, implementation considerations, and future improvement of the VXE.

1 The Target

The target of data dependency tracking is to answer the following questions:

1. (The simpler form) Whether certain data have an effect on the output of a function instance;
2. Which data have an effect on the certain output of a function instance.

2 The Current Design

Currently we are using a relatively naive solution: after the program is actually executed, we build the dependency graph between all *data units* by simulating the whole process on the bytecode level. Here the *data units* are a general form of representing data, which include frames with registers (required to simulate stacks), global static members and arrays.

Generate dependency graph from bytecode

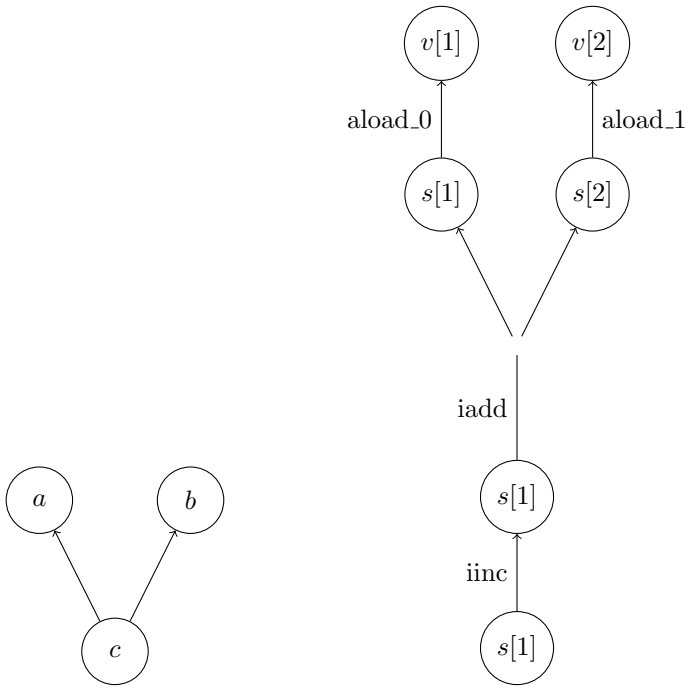
There are plenty of libraries that can be used to manipulate the bytecode, which saves us from implementing a compiler. But there are disadvantages too, such as the explosion of the dependency graph. Consider the following example:

```
public int add1(int a, int b) {
        int c = a + b + 1;
        return c;
}
```

Example 1

As demonstrated in Figure 1, we can see that the graph size for bytecode is larger than that of statements. This can be improved by running a static analysis to compress the graph.

Now we rewrite the program a little bit:



(a) The dependency graph of the statement "c = a + b + 1;"

(b) The dependency graph of the compiled statement "c = a + b;"

Figure 1

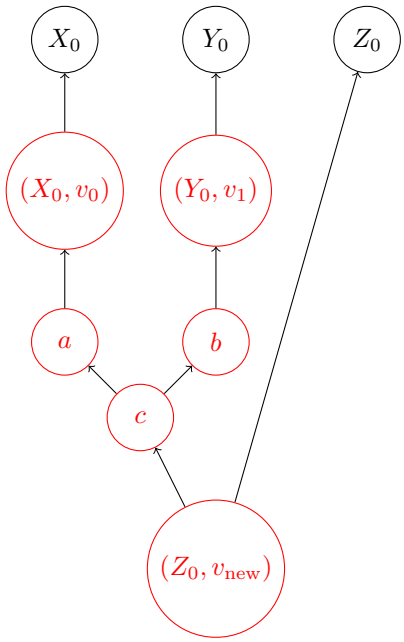
```
public void run(X, Y, Z) {
        int a = datastore.read(X);
        int b = datastore.read(Y);

        int c = a + b + 1;
        datastore.write(Z, c);
}
```

Example 2

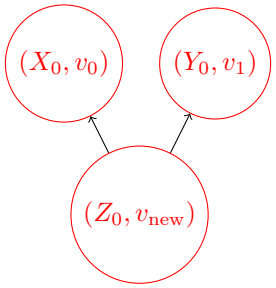
Identify VXE API and introduce the label system

For oridinary programs we will see a dependency graph with a dependening on both *datastore* and X. But in VXE, a only depends on X but the node representing a will be marked with a tuple (*key, version*) which designates the value that is read from the datastore. The value of the *key* in this tuple will be replaced with the exact value of X at runtime. If we execute the method with parameters X_0, Y_0, Z_0, the dependency graph in Figure **??** can be obtained.

Basically the graph can be interpreted as "the value of data at Z_0, whose version is v_{new}, depends on the data at X_0 and Y_0, and the corresponding versions are v_0 and v_1", and we can futher compress it to Figure **??**.



(a) Example of a complete dependency graph for VXE



(b) Example of a compressed dependency graph for VXE

```
public void run(X, Y, Z) {
        int a = datastore.read(X);
        int b = Y;

        int c = a + b + 1;
        datastore.write(Z, c);
}
```

For ordinary programs these two programs should have similar data dependency graphs except that b also depends on *datastore* in the former. In VXE, however, they are quite different.

For the label system to identify the VXE datastore API and get the correct version of the ,