CSC410 Final Project
The Report

Group Members

Plato Man Hon Leung (leungpla)

Yuzhong Liang (liangyuz)

Shao Xuan Teoh (teohshao)

Xiao Ming Zhu (zhuxia19)

2017

CSC410 Fall

Manual

Topic

Project 1. Detecting and analyzing loop constructs

Source Code on GitHub

`https://github.com/alexteoh/XPAK-LoopDetectionProject.git`

Project Directories

1. loop_analysis.py
   - Main utility module to collect loop information.
   - It contains several NodeVisitor classes. Each of them specializes in collecting one specific information (e.g. read/write variables, dependency vectors, etc.)
2. run_official_tests.py
   - Execute loop analysis on all the given inputs.
3. official_inputs
   - Directory that stores official inputs (c files)
4. past_checkins
   - Directory that contains python scripts from past check-ins
5. custom_inputs
   - Directory that stores custom testing inputs (c files)
6. pyminic
   - Submodule that points to Victor's repo: https://github.com/victornicolet/pyminic

Prerequisite

1. Install pycparser. (e.g. sudo pip install pycparser)

Instructions

```
$ git clone https://github.com/alexteoh/XPAK-LoopDetectionProject.git
$ cd XPAK-LoopDetectionProject
$ git submodule init
$ git submodule update
$ python run_official_tests.py
```

Overview

Given a piece of C program, we perform dataflow analysis on its loop constructs. We chose Python with the PyCParser and PyMinic libraries to parse the C program into an Abstract Syntax Tree (AST). We implemented several NodeVisitors to collect important information about loop constructs.

Here is an overview of the information that our program collects.

1. For each loop, we collect
    - the set of read/write variables
    - the set of live variables at loop entry
    - the set of reaching definitions
    - index used, guard condition, and update statements
2. For each nested loop, we analyze loop dependencies
    - Flow (True) Dependence
    - Anti Dependence
    - Output Dependence (incomplete)

Reason to consider the dependence

When analyzing dependence, we have to look at the control flow and data flow then consider the loop parallelism. To determine whether the loop can run in parallel or not, we have to consider the function dependencies.

Flow (True) Dependence occur when read-after-write (RAW) to the same memory location, Anti Dependence occur when write-after-read (WAR) to the same memory location , and Output dependence occur when write-after-write (WAW) to the same memory location. If there are any dependencies, it means iterations interfere with each other and we cannot make sure the loop will run in parallel. In the other words we want to avoid race condition in the program.

Difficulties

1. **Start**. At first, it took us quite some time to get familiarize with the functionality that pycparser and pyminic provide. The source code was overwhelming. After playing around with it for the first check-in project, it becomes clearer on how to manipulate NodeVisitor and all the statement objects. Overall, we find these objects are very well-structured.

2. **Dependence Analysis**. To implementing the dependence vector, we need to access the statements that are nested in multiple levels of loops. It is challenging to obtain the correct iteration. We need to check if the loop is nested, and look of its children loop if so, until we get to the statement. Once we access correct statement in the loops, we have to compute the three types of function dependence. when we have the right indices from the statements, we have to make sure we compute the dependencies accurately. In general, to compute flow dependence and anti dependence are more straightforward than to compute output dependence.
3. **Debugging**. As the size of our code grows, it becomes very difficult to trace the content of the data structures we used. Sometimes these data structures are updated by AST traversal; sometimes they are updated in recursion. A few of the data structures we used are perhaps too complex. For example, the dictionary, that we used to create a dependency mapping, has multiple dictionaries inside it. It's not trivial to reason about these data structures.
4. **Code Cleaning**. During development, we created many inefficient and unnecessary complex functions. It is hard to trim them down at the end. Tens of commits were submitted to clean up the code.
5. **Collaboration**. Although we have 4 people on the team, it was not very easy to break down tasks and assign them to individuals every week. When someone skips one or two weeks of works, it is hard for that person to catch up. However, later in the term, we collaborate better. We often discuss issues and solve difficult problems together. We also review and give feedback on each other's work routinely. This helps each other understand the code more and eventually write better code.