A GSoC 2013 Final Report

Enhancing Giri: Dynamic Slicing in LLVM

Mingliang Liu, Swarup Kumar Sahoo
http://github.com/liuml07/giri

Dynamic program slicing has been used in many applications. Giri was a research project from UIUC, which implemented the dynamic backward slicing in LLVM. It was selected as the Google Summer of Code 2013. We achieved several improvements to Giri during GSoC 2013: *1*) Update the code to LLVM mainline and make it robust, *2*) Reduce the trace size, *3*) Make the Giri thread-aware (pthread only), *4*) Improve the performance of run-time.

1 Introduction

Program slice contains all statements in a program that directly or indirectly affect the value of a variable or instruction [7]. We can further narrow down the notion of the *slice* to *dynamic program slice* [1]. Unlike traditional slicing named static slicing which computes all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program, the dynamic program slicing computes all statements that actually affect the value of a variable/instruction at a program point for a particular execution of the program with given input. There are many applications from research and industry organizations which use (or could benefit from) dynamic slicing. For example, it has long been used in software debugging [2, 4] and testing [3]. However, to the best of our knowledge, there is no publicly available dynamic slicing tool in either GCC or Open64.

Sahoo et. al. use dynamic program slicing to filter/remove false positives from the candidate root causes for automated software fault localization [5]. They implemented the dynamic program slicing code, named Giri, in LLVM compiler infrastructure (version 3.1) for research purpose. It collects the trace of a user program execution and finally reports the dynamic slice of a given statement. It also maps LLVM intermediate representation (IR) statements to source code using the debug meta-data to report the dynamic slice in terms of source line numbers. To make the Giri up-to-date and robust for general use, we enhanced the code in several ways.

The report is organized as following. Section 2 illustrates the dynamic slicing and shows the internal architecture of Giri. Section 3 lists the progress we made during the GSoC 2013. Section 4 lists the future work to improve Giri further. Section 5 concludes our work.

2 Overview

2.1 Dynamic Program Slicing

The following is an example program.

```
1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 typedef struct result
6 {
7     int result;
8     int count;
9 } result_t;
```

```
10
11 void calc(int x, result_t *y, result_t *z)
12 {
13     if (x < 0)
14     {
15         y->result = sqrt(x);
16         y->count++;
17         z->result = pow(2, x);
18         z->count++;
19     } else {
20         if (x == 0)
21         {
22             y->result = sqrt(x * 2);
23             y->count++;
24             z->result = pow(3, x);
25             z->count++;
26         } else {
27             y->result = sqrt(x * 3);
28             y->count++;
29             z->result = pow(4, x);
30             z->count++;
31         }
32     }
33 }
34
35 int main(int argc, char *argv[])
36 {
37     int x, ret;
38     result_t y = {0, 0}, z = {0, 0};
39
40     x = atoi(argv[1]);
41
42     calc(x, &y, &z);
43
44     ret = printf("%d\n", y.result);
45     printf("%d\n", z.result);
46
47     return ret;
48 }
```

Given the input `10` to this program from command line, the source line numbers affect the return value are: *<13, 20, 27, 40, 42, 44, 47>*. The line numbers *<13, 20>* are included due to control dependence, while line numbers *<27, 40, 42, 44, 47>* are included due to data dependence.

The default starting point of slicing is the `return` instruction at the `main` function. Note that there is only one `return` when the main executes. Giri also supports two more ways to specify the starting criterion of the slice, which are source code line number and LLVM instruction number respectively. See the `test/UnitTests/test4` for more information.

2.2 The Design of Giri

Giri handles both data-flow and control-flow dependences (optional) when computing the dynamic backwards slice [5]. It has two phases. In the first pass named *tracing* pass, it instruments the code to record various runtime information in a trace file at runtime for all threads. In the second pass named *slicing* pass, it uses the execution trace to create a program dependence graph for computing dynamic slice. Giri takes advantage of the LLVM intermediate representation (IR), which is static single assignment (SSA) form, to reduce the size of the trace file.

The tracing pass instruments code to record three different pieces of information: *1*) basic block exits; *2*) memory accesses and their addresses; and *3*) function calls and returns. The slicing pass initializes a work list with the starting instruction (also called *criterion*) of slicing. For each dynamic value in the work list, the slicing pass computes both data-flow and control-flow dependences for it using program SSA form and trace. First, we compute all the dynamic instructions which influence the current dynamic value under consideration through direct flow of data values. Next, static control-dependence analysis is employed to determine which values or instructions can directly force the execution of the basic block to which this dynamic value belongs. Then, the slicing pass uses the trace records to find the most recent dynamic instance of those values or instructions which forces the execution of this dynamic value. The dynamic values computed by control or data dependence are added to the work list if not processed before. The slicing pass does not stop until all dynamic values in the work list are processed. In the end, it reports the dynamic slice of the program execution from the slice starting point in terms of LLVM instructions as well as the source code line number information.

Since most instructions operate on SSA scalar values, we only refer to dynamic trace whenever needed. For example, we can get all the operands of an `add` instruction from the program SSA form without accessing the trace file. However, when we try to find all the `store` instructions which feed value to a `load` instruction, we need to scan the trace file since the data read by the `load` instruction may be written by many possible `store` instructions, which can only be determined accurately at runtime.

To make the slicing pass thread aware, the runtime library writes the thread id (`thread_t`) to each record indicating current thread performing that operation. The slicing pass checks the thread id when scanning the trace file for a given dynamic value or basic block.

3 Progress

Our goal of GSoC was to make the Giri code up-to-date with the latest LLVM version, improve its

runtime performance, and/or reduce the tracing overhead. There are several things we did in this summer:

1. **Update the code to LLVM mainline.**
   - Clean up the code for version 3.1
   - Make the code up-to-date with the latest LLVM 3.4 revision (`r191529`)
2. **Make the Giri run-time library thread safe and the slicing pass thread aware**. The trace records of all threads are written to a single file. In addition to other information, trace records also include the thread id (`pthread_t`), which indicates the thread performing that particular operation. We use locks when we write trace records to the trace file in order to avoid race conditions.
3. **Improve the Giri run-time performance.** We checked every call to function `mmap` as well as the parameters, and eliminated the useless ones to improve performance. For example, at the end of the tracing, it is not needed to re-`mmap` once all the trace records are synchronized to disk file. Giri now dynamically computes the cache size of trace records to hold in memory before flushing to disk.
4. **Write dozens of unit tests and try more real programs.** Giri was able to handle few large real programs like Squid, Apache, MySQL before GSoC. We added few more real programs to this set. There is also a simple test framework which runs all the unit tests at the top level of `giri/test/` and report the results. In the future development, we shall make sure all unit tests pass locally before submitting a patch to the code repository.
5. **Reduce the trace size.** We truncate the trace file at the end of tracing according to its real size.
6. **Write documents for the project.** There are several sample pages:
   - The Wiki Page Home.
   - How to Compile Giri.
   - Hello World! Example.
   - Command Line Usage With An Example.

4 Future Work

The following is the list of future work we are going to address in the next few months. An updated list is maintained online at the wiki page of Giri at github.com.

1. Improve the performance of locking mechanism of the Giri runtime.
2. Write more unit tests with complicated direct/indirect recursive function calls.
3. Double check the `TraceFile.cpp` to make sure the thread id be checked correctly.
4. Add `tool/Tracer.cpp` to the make list. It was removed when upgrading to LLVM 3.4.
5. Try more real world programs and add them to the `test/` directory.
6. Try large test programs, e.g. *nginx*, *squid*, etc.
7. Make Giri code useful for other platforms besides Linux, e.g. Mac OS X, Cygwin, and FreeBSD.
8. Handle more special function calls, which are intrinsic LLVM instructions (like *libc* functions `printf, scanf, sprintf` etc.) in `TracingNoGiri::visitSpecialCall()`.
9. Parallelize the code which writes the entry cache containing trace records to trace file and adds trace records to the entry cache.

5 Conclusion

As the first publicly available dynamic slicing tool, Giri was improved significantly during the GSoC. We updated the code to LLVM mainline, reduced the trace size, made it thread-aware, and improved the performance of its run-time. We have published our code at https://github.com/liuml07/giri [6].

The code is still under active development. Dr. Swarup will direct Mingliang LIU in future to make the Giri code better. There are other researchers from Tsinghua University, Xi'an Jiaotong University and University of Miami, who contacted us for permission to use and contribute to Giri project. For more information, please visit the homepage above.

References

[1] H AGRAWAL. Dynamic Program Slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[2] Hiralal Agrawal, Richard A Demillo, and Eugene H Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[3] Hiralal Agrawal, Joseph R Horgan, Edward W Krauser, and Saul A London. Incremental regression testing. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*, pages 348–357. IEEE, 1993.

[4] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Software EngineeringESEC/FSE99*, pages 303–321. Springer, 1999.

[5] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. ACM.

[6] Swarup Kumar Sahoo, John Criswell, Mingliang Liu, and Vikram Adve. Giri: Dynamic Program Slicing in LLVM. `https://github.com/liuml07/giri`, 2013.

[7] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE, pages 439–449. IEEE, 1981.