PORBS: A Parallel Observation-based Slicer

Syed Islam
School of Architecture, Computing and Engineering
University of East London, United Kingdom
syed.islam@uel.ac.uk

David Binkley
Loyola University Maryland
Baltimore, USA
binkley@cs.loyola.edu

***Abstract*—This paper presents PORBS, a parallelised observation-based slicing tool. The tool itself is written in Java making it platform independent and leverages the build chain of the system being sliced to avoid the need to replicate complex compiler analysis. The target audience of PORBS is software engineers and researchers working with and on tools and techniques for software comprehension, debugging, re-engineering, and maintenance.**

I. Introduction

Program Slicing has many applications [1], including comprehension, testing, debugging, maintenance, re-engineering, re-use, and refactoring. Despite these many applications, program slicing has not been widely adopted as part of a modern development toolkit. Two long-standing shortcomings contribute to this lack of broader use: slicing tools are typically unable to handle multi-language systems nor systems that use (unknown) binary components or libraries.

We introduced Observation Based Slicing (ORBS) [2] to address these two issues. ORBS can compute slices for heterogeneous systems. In addition to handling multiple languages, it removes the need for the slicer to replicate much of the compiler's analysis (e.g., parsing) by leveraging the existing build system. The tool used for the initial proof-of-concept (herein referred to as *serial ORBS*) was single threaded and thus computed a slice sequentially.

In this paper we present the tool PORBS, which computes *parallel observation-based slices*. The tool leverages multiple threads to perform simultaneous observations as opposed to the single observation of serial ORBS. PORBS constructs a slice faster than the serial version and scales to better use the available hardware resources of modern multi-core processors.

II. Motivation & Related Work

Modern software systems are typically composed of two to fifteen programming languages [3, pp. 504-505]. As the number of languages involved increases, the challenges faced by those who must understand and maintain these systems also increases. A slicer for multi-language systems provides a stronger incentive for industrial adaptation as part of a development, comprehension, and maintenance toolkit.

Of the many tools aimed at aiding a software engineer, the two most closely related to PORBS, Critical Slicing [4] and Delta-Debugging [5], both employ a similar deletion-oriented approach. However comparison studies have shown that these two can produce incorrect slices and can also become prohibitively expensive [2].

The tool most similar to PORBS is serial ORBS. Initial experiments find that even when using only moderate parallisation on an eight core machine, PORBS takes up to 82% less time to construct slices. Given the abundance of hardware available in the cloud, for instance, Microsoft Azure[1] provides virtual machines with 32 cores while Amazon Web Services[2] provides up to 40, PORBS stands to benefit as it can leverage these enormous hardware resources to reduce slice computation time.

III. PORBS Use Case

This section presents a PORBS use case. Consider the code shown in Figure 1, which consists of three components: a Java program, a C program, and some connecting logic written in Python. Assume an engineer needs to understand the computation of the variable `dots` at Line 12 of `checker.java` (this line-variable pair is referred to as a *slicing criterion*). The resulting PORBS slice, shown in Figure 2, includes only those statements that influence the computation of `dots`.

IV. PORBS Design and Configuration

This section considers the design and configuration of the PORBS tool. For a complete description of the configuration options, log files, and output file formats please refer to the online distribution website.[3]

A. Design

The underlying ORBS approach finds its origin in Weiser's original motivation for slicing [6]: uninteresting statements can be deleted (sliced out) of a program to help focus attention on relevant statements (i.e., the slice). Operationally, PORBS achieves this by tentatively deleting one or more lines and then observing the behavior of the resulting program. In greater detail, first a candidate slice is produced by deleting one or more lines of text from a program. Then this candidate is compiled and executed. If the compilation fails or the execution fails to terminate normally, the candidate is rejected. Otherwise, its output, restricted to the slicing criterion, is compared to that of the original program. If the two differ the candidate is rejected. Only if the candidate compiles, executes, and produces the correct output is it accepted and becomes the current slice from which further line deletions are attempted. This process is repeated until no further deletions are possible. One PORBS goal is language independence. Thus the tool does not consider language specific deletions such as removing the body of a function.

PORBS starts with an initialization phase (Figure 3), where it builds the original system and executes it n times to determine

[1] https://goo.gl/pJYshj
[2] https://aws.amazon.com/ec2/instance-types
[3] http://www.syedislam.com/orbs.html

checker.java:

```
class checker {
  public static void main(String[] args) {
    int dots = 0;
    int chars = 0;
    for (int i = 0; i < args[0].length(); ++i) {
      if (args[0].charAt(i) == '.')
        ++dots;
      else if ((args[0].charAt(i) >= '0')
                && (args[0].charAt(i) <= '9'))
        ++chars;
    }
    System.out.println(dots); // Slice here
    System.out.println(chars);
  }
}
```

reader.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>
int main(int argc, char **argv) {
  setlocale(LC_ALL, "");
  struct lconv *cur_locale = localeconv();
  if (atoi(argv[1]))
    printf("%s\n", cur_locale->decimal_point);
  else
    printf("%s\n", cur_locale->currency_symbol);
  return 0;
}
```

glue.py:

```
# Glue reader and checker together.
import commands
import sys
use_locale = True
currency = "?"
decimal = ","
if use_locale:
  currency = commands.getoutput('./reader 0')
  decimal = commands.getoutput('./reader 1')
cmd = ('java checker ' + currency
       + sys.argv[1] + decimal + sys.argv[2])
print commands.getoutput(cmd)
```

Fig. 1. Example Multi-Language Application

checker.java:

```
class checker {
  public static void main(String[] args) {
    int dots = 0;
    for (int i = 0; i < args[0].length(); ++i) {
      if (args[0].charAt(i) == '.')
        ++dots;
    }
  }
```

reader.c:

```
#include <locale.h>
int main(int argc, char **argv) {
  struct lconv *cur_locale = localeconv();
    printf("%s\n", cur_locale->decimal_point);
}
```

glue.py:

```
import commands
import sys
use_locale = True
currency = "?"
if use_locale:
  decimal = commands.getoutput('./reader 1')
cmd = ('java checker ' + currency
       + sys.argv[1] + decimal + sys.argv[2])
print commands.getoutput(cmd)
```

Fig. 2. Sliced Example from Fig. 1

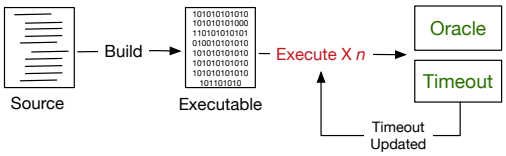

Fig. 3. PORBS initialization step

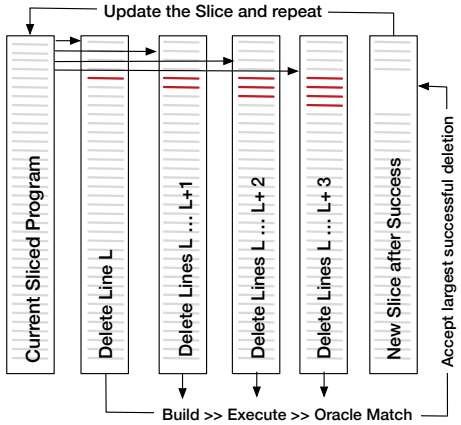

Fig. 4. PORBS parallel windowing strategy

an average execution time as well as the expected output, referred to as the *oracle output*. The average execution time is used to setup a runtime limit for each candidate execution as a guard against non-termination. The oracle is used to ensure the slice preserves the desired behavior.

While it is possible to delete a single line at a time, it is often more efficient to remove multiple lines. Sometimes, for example in the case of matching braces on consecutive lines, it is necessary to delete multiple lines to produce a syntactically valid program. PORBS controls the number of lines simultaneously deleted using the configuration parameter *window size*.

Although ORBS is sequential by nature, PORBS exploits the possibility of different deletion windows sizes to create parallel variants (Figure 4). The program resulting from the largest deletion that succeeds (i.e., compiles and produces the oracle output) becomes the current slice. The other attempts are discarded. This process continues until an iteration (a complete pass over the current slice) fails to delete any lines.

In a preliminary study, we ran PORBS against serial ORBS on several systems and found that the total number of executions performed by PORBS is similar to that of serial ORBS and often PORBS requires fewer compilations than serial ORBS. The net effect is that, user time often decreases, which can lead to a dramatic drop in wall-clock time: PORBS with a window size of 4 needs 70%-82% less time than serial ORBS. PORBS does not impose an upper bound on the number of windows that can be considered in parallel but available system resources provide a pragmatic limit.

B. PORBS Configuration

PORBS uses a standard Java properties file `OrbsFramework.Properties` for all its configuration settings. For example, which files of the subject system should be sliced, the number of parallel deletion windows to be used, slicing direction (top of file to bottom or bottom of

```
reader.c glue.py checker.java
|---a-------------|--------------|----------------|
|bb-.-a-a---eeeee-|--a--bb-a-a---|---a---ccc--bb--|
|..-.-.-.---.....-|--.--..-.-.---|---.---...--..--|
```

Fig. 5. Delete pattern file

Iteration	Lines Deleted	Compilations	Executions	Cachced Compilations	Cached Executions	Elapsed Time
0	1	0	0	0	0	00:00:00
1	20	173	73	48	49	00:00:33
2	0	120	50	4	6	00:00:13
Total:	21	293	123	52	55	00:00:46

Fig. 6. Iteration-level Statistics

Window Size	Lines Deleted	Compilations	Executions	Cachced Compilations	Cached Executions	Re-run Fail
1	6	52	26	11	12	0
2	6	47	17	11	12	0
3	3	42	18	10	10	0
4	0	38	15	8	9	0
5	5	34	14	6	6	0
6	0	30	13	4	4	0
7	0	27	9	2	2	0
8	0	23	11	0	0	0
Total:	20	293	123	52	55	0

Fig. 7. Window-level statistics

file to top), whether to include a check for non-deterministic behavior, whether to cache binaries and execution results, and the directory structure of the application being sliced. Separate scripts are used to setup the test system (`setup.sh`), build the test system (`compile.sh`), and execute the test system (`execute.sh`) using a set of inputs.

C. Running PORBS

PORBS is currently distributed as a JAR file and can be run with Java 1.8 and above. All dependencies for the system are packaged within the jar file and thus PORBS has no external system dependencies. A bash script (`orbs.sh`) is provided to simplify the process of starting PORBS. Scripts are also provided to help developers and researchers run PORBS in batch mode on several systems using a variety of window sizes.

D. PORBS Output

Every PORBS run creates a new output directory (by default under the directory `regression`), which contains the executable slice, copies of all the configuration files used, and statistics and log files. The compile and execute scripts can be used directly on the output directory to validate that the slice produces the oracle output. The remaining output files include statistics, the deletion pattern, and the execution logs.

PORBS creates two `csv` files of statistics. The first, `iterationLevelStat.csv` (see Figure 6) provides summary details regarding compilations, cache hits (when a build produces the same binary as a previous build), deletions, and run times for each iteration. The second file, `windowLevelStat.csv` (see Figure 7), provides similar information broken down by window size. These statistics also describe deterministic re-run failures for each window size.

PORBS also outputs, in the file `DeletePattern.log` (see Figure 5), the results of the deletion attempts at each line for every iteration. The pattern visually depicts the deletion process' progress. Letters in the pattern depict the size of a successful deletion. For example, the three `c`'s on the second line indicate a deletion of a window of size three. Each line in the pattern corresponds to an iteration.

Finally, PORBS produces two log files. The first, `orbs.log`, contains the configuration information, details of PORBS deletion attempts, and all the raw statistics from a run. In most cases, this log file alone provides all the information regarding the computation of a slice. The second log file, `trace.log`, contains detailed information about each compile and execution attempt for every window size. This log file is useful in debugging and in understanding the details of the deletion attempts (e.g., why a particular attempt failed, what output was actually generated, etc). The logging level is set by the `log4j2.xml` configuration file.

E. Observations and Challenges

During experiments with PORBS we made several behavioral observations that impact the slice. For example, the order of the file in which the slicing is attempted, the execution environment and the number of parallel windows all have an impact on the observed slice. By definition ORBS is guaranteed to produce a correct, executable, and minimal slice, technically the observed slice is *1-minimal*, that is, nothing can be removed from the observed slice but there may also be other minimal slices built using the same slicing criterion and input. Additionally, we also observed that in C programs removing initialization code can lead to non-determinism in the slice. PORBS can re-run accepted deletions with varying memory initialization bit patterns, in an attempt to reduce the chances of introducing non-deterministic behaviour.

V. Summary and Future work

This paper presents PORBS, a parallelised scalable observation-based program slicer that can deal with heterogeneous systems. Given the initial success in preliminary studies (where PORBS reduces slicing time by up to 82%), we are optimistic that the PORBS tool will be useful in a variety of analyses. In the future we plan to make improvements to PORBS both in terms of precision, for example by implementing token-level deletion rather than line-level deletion, and in terms of speed, for example, by allowing slicing to be carried out at the binary level forgoing the compilation step.

References

[1] Andrea De Lucia. Program slicing: Methods and applications. In 1st *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Los Alamitos, California, USA, 2001.

[2] David Binkley, Nicolas Gold, M. Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE 2014, pages 109–120, 2014.

[3] Capers Jones. *Software Engineering Best Practices*. McGraw-Hill, 2010.

[4] Richard A DeMillo, Hsin Pan, and Eugene H Spafford. Critical slicing for software fault localization. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 121–134, 1996.

[5] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[6] Mark Weiser. Program slicing. In 5th *International Conference on Software Engineering*, pages 439–449, San Diego, CA, March 1981.