Simulator-Based Diff-Time Performance Testing

Ivan Postolski, Victor Braberman, Diego Garbervetsky
UBA, ICC, CONICET. Argentina
ivan.postolski@gmail.com, {vbraber, diegog}@dc.uba.ar

Sebastian Uchitel
UBA, ICC, CONICET. Argentina
Imperial College London. UK.
suchitel@dc.uba.ar

***Abstract*—We propose an approach for rapid detection of performance regressions using a simulator built from the original program by dynamic slicing and a certificate built using static analysis that generalizes its correctness. We discuss two case-studies that illustrate the potential benefits of the proposal.**

***Keywords*-Performance testing, Dynamic and Static Slicing**

I. Introduction

Software performance is a rising concern that remains an open challenge for verification approaches. In particular, performance testing is a resource intensive task that often involves heavyweight resource usage and time consuming analysis of results. This hinders providing early feedback about performance regressions. Furthermore, recent reports [1] indicate that the timeliness of a regression detection is an essential factor to alleviate developers 'context-switch', and thus to improve fix rates. Our aim is to develop lightweight techniques that can detect performance regressions and report back at diff-time (i.e. right after a commit is introduced).

A recent study [2] identifies that one of the two major reasons for performance regressions involves changes in the program logic that result in expensive code regions or calls being executed more times than expected. For instance, the MySQL regression bug #49264 reports a change that caused an expensive table lookup function to be executed unnecessarily multiple times causing an up to 80x degradation.

Our proposal for detecting these kind of performance regressions is to automatically transform a program into a faster one, a *performance simulator*, with less dependencies, that only reports the number of times expensive calls would have been hit. A performance simulator allows to run a performance test suite much faster than the original program while still detecting regressions on the number of expensive calls made. Our proposal is orthogonal to strategies such as the use of, for example, stubs and mocks.

A high level overview of this approach is shown in Figure 1(a). The approach can be thought of as a hybrid between a measurement and a model-based approach [3], where actual application code is executed, but where some code is abstracted and a proxy to performance is analyzed.

This paper contributes the following: i) a new approach to detect performance regressions based on simulators built using dynamic slicing, ii) a novel approach based on static analysis to certify the guarantees of a simulator, and iii) two case-studies that illustrate the potential benefits of the approach.

II. Dynamic Slicing Construction of Simulators

Slicing out from the original program the lines of code that are irrelevant for determining the number of expensive calls performed is key to the construction of an efficient simulator. The crucial factors of such slice are precision (degree to which irrelevant lines are removed), and soundness (universe of inputs for which the slice is correct). *Building a simulator based on a sound slice will allow correct detection of performance regressions, building it from a precise slice will allow the simulator to run efficiently*.

Given a program, an expensive call and some functional tests, it is possible to build a simulator based on an executable dynamic program slice: We introduce ghost counters at the expensive call locations and slice out the statements that do not affect the counters values. The dynamic slice will preserve the number of times the expensive call would have been executed in the given tests, but may not do so for other tests. In other words, dynamic slicing [4] will produce a simulator that in principle is guaranteed correct only for a limited number of tests. Nevertheless, a dynamic slice will usually be very precise, hence will be more efficient to run as many irrelevant computations and external dependencies will have been removed.

Static program slicing techniques, on the other hand, do not need a test to generate a program fragment and generally do better in terms of soundness, but must trade precision for scale [4]. A simulator built using static slicing, will potentially allow identifying more performance regressions. However, the simulator may be as slow as the original program.

We believe that high precision (a fast simulator) is an absolute must in order to support rapid performance regression testing. Thus, we aim to build on the strengths of dynamic slicing and mitigate as much as possible its weaknesses. Although there are no guarantees on the soundness of a dynamic slice beyond the tests that were used to generate it, *our hypothesis is that a subsequent analysis can certify a larger set of inputs for which the simulator is correct*. This will allow certifying the correctness of many performance tests run on the simulator even when they were not used to build it. An overview of the regression detection process enhanced with the certification process is shown in Figure 1(b).

III. Simulator Certification

Given a simulator built from a dynamic slice, the application source code, and an input, we define a *correctness certificate* as a program that determines whether the output of the

PRE-PRINT

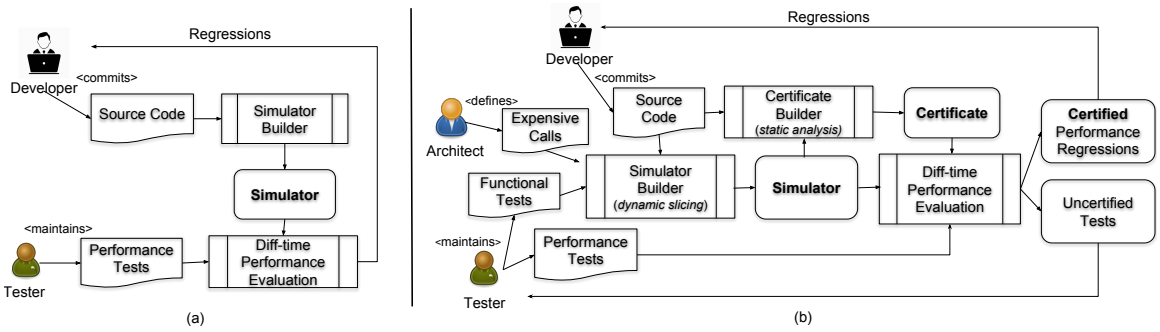

Fig. 1. (a) Simulator-based diff-time performance regression approach and (b) a refinement that assumes a dynamic sliced simulator and certificate.

simulator is correct for the input. A naive, but unacceptably slow, certificate is a program that executes the simulator and the original program, and compares their execution. We need certificates to run fast as we need to certify inputs at diff-time.

A key insight for constructing efficient correctness certificates is that the program dependencies missed by the simulator construction can be used to reason about simulator correctness.

The dependencies used to build the simulator via dynamic slicing can be thought of as determining a dependency sub-graph of those identified by a static slice. We define a *frontier dependency* as a dependency between a statement that is in the sub-graph (dynamic slice) and one that is not. It is possible to prove that if the execution of the original program under an input I never hits a frontier dependency (i.e. never escapes from the sub-graph) then all the statements that affect the number of expensive calls are present in the simulator. Therefore, the dynamic simulator output is correct for the input I (i.e. the result is the number of expensive calls performed by the original program for input I).

The above result can be exploited for a certificate construction by i) discovering the frontier dependencies (by static analysis) and then ii) transforming the original program to a certificate program that fails (reaches `assert(false)`) when a frontier would have been executed in the original program.

While this certificate might accelerate the execution for inputs that fail, safe inputs (those who never execute a frontier) will still require the complete execution of the original program. Consequently, our approach aims to further reduce the cost of certifying safe inputs by *anticipating* in the certificate execution when an input is not going to hit a frontier. Such anticipation can be achieved by adding safety conditions to the certificate program. Where a safety condition is a sufficient condition for the rest of the execution to be free of failures (simil weakest precondition).

The efficiency of our certificate programs is related to the time it takes to reach a successful safety condition or a frontier. In our setting, an ideal safety condition is one that holds for a large set of inputs and is evaluated as soon as possible. Notice that the construction of the safety conditions is directly related to the number of frontiers and their locations in the program, as the frontiers delimit the safe paths of the certificate.

Thus, the challenge to achieve an efficient certificate construction process (i.e., the Certificate Builder) is to optimize (within a reasonable time budget) the quality of the generated safety conditions by improving the precision of frontier dependency discovery analysis and the backward propagation analysis used to build the safety conditions. *Our hypothesis is that is possible to efficiently construct certificate programs that are fast enough to be used at diff-time.*

IV. Use Case

We envisage the use of a performance simulator and its correctness certificate program as follows (italic terms in brackets refer to elements in Figure 1(b)). An architect and the performance testing team define i) a set of expensive calls, ii) functional tests to build the simulator via dynamic slicing, and iii) performance test inputs to determine a performance baseline chart that records a profile of hits to expensive calls.

When developer's diffs are introduced to the code, a simulator is built (*Simulator Builder*) using the provided functional tests and then a correctness certificate is built (*Certificate Builder*) from the original program and the simulator.

Subsequently, performance tests with inputs accepted by the certificate are executed with the simulator to generate a performance chart (*Diff-time Performance Evaluation*). Differences in the performance chart beyond a threshold are reported to developers (*Certified Performance Regressions*).

On the other hand, performance tests with inputs that do not pass the certificate (c.f. *Uncertified Tests*) are reported to the testing team to raise awareness that some performance tests are not being executed against a proposed diff. A tester team member could decide to run these tests later in the standard performance testing phase, or try to improve the simulator regression detection process by improving (possibly by simply extending) the functional test cases used to build the simulator.

V. Simulator and Certificate Example

We illustrate an example of our approach with the C code of Figure 2. The code has a `main()` function with an unique parameter n, a target function `expensive_op()`, and an added ghost counter `expensive_count`. If we consider a simulator that is built from a dynamic slice of the `expensive_count` value at line 4, we obtain three possible behaviours depending on the input used: If n is even, the `expensive_op` is executed `f(n)` times. If n is odd and

DRE-DEUT

less than 1024, `expensive_op` is never executed. Finally, if n is odd and greater than 1024, `expensive_op` is executed `g(n)` times.

```
1  int expensive_count = 0;
2
3  void expensive_op(){
4    expensive_count++;
5    {expensive code region};
6  }
7  int main(int argc, char *argv[]){
8    int n;
9    n = atoi(argv[1]);
10
11   if (n%2 == 0) {
12     for(int i = 0; i < f(n); i++)
13         expensive_op();
14   } else {
15     if (n > 1024){
16       for(int j = 0; j < g(n); j++)
17         expensive_op();
18     }
19   }
20   return 0;
21 }
```

n = 256
n = 1025

Fig. 2. Original Program and Dynamic Slices for $n \in \{256, 1025\}$

We first exemplify the problem of simulators that produce incorrect results: Suppose we build a simulator using 256 as input. The simulator will not include the lines 14, 15, 16, 17 and 18. As a result, the simulator output (final value of `expensive_count`) will be correct when the simulator is executed with inputs that are either even or less than 1024. However, `expensive_count` will not end with the correct result if an odd number greater than 1024 is used since it will return 0 instead of `g(n)`.

We now exemplify how a correctness certificate can generalize correctness beyond input 256. Figure 3 shows an extract of the dependency graph of the program. The dotted link refers to the frontier of the simulator generated with the input $n = 256$. The frontier is a control-dependency introduced by the `expensive_op` function invocation of line 17. Figure 4 shows the certificate built by the Trimmer [5] tool, a safety-condition propagation analysis for C. The certificate program contains an assertion failure (line 9) which is triggered when the statement in line 17 of the original program would have been executed. Thus, odd inputs greater that 1024 will fail at line 9 and not be certified.

The certificate also includes two safety conditions, i) at line 5 for even inputs, and ii) at line 7 for odd inputs below 1024. An execution reaching these lines and satisfying the conditions is guaranteed to not execute the frontier and can be stopped.

In order to exemplify a regression detection, suppose that we have a performance test suite that executes the inputs $\{101, 506, 507, 1025, 10^6\}$, and that a developer diff that accidentally duplicates `f(n)` return value when n is greater than 314. In addition, suppose that the simulator is constructed with input $n = 256$ (Figure 2) and then its corresponding correctness certificate is built (Figure 4). Evaluating the performance tests at diff-time with the exemplified simulator and its certificate will: i) certify inputs $\{101, 506, 507, 10^6\}$ (80% of the tests), ii) detect a regression for inputs $\{506, 10^6\}$ (i.e., the times `expensive_op` is hit has duplicated with respect to the baseline), and iii) report the input 1025 as not certified (20% of the tests).

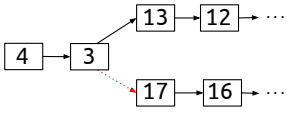

Fig. 3. Dependency graph extract for `expensive_count` field at line 4. Numbers in boxes represent the statements lines. Dotted arrow is the frontier for Simulator built from a dynamic slice ($n = 256$).

```
1  int main(int argc, char *argv[]) {
2      int n = atoi(argv[1]);
3      int t1 = n % 2;
4      int t2 = t1 == 0;
5      check(t2);
6      int t3 = n <= 1024;
7      check(t3);
8      for(int j = 0; j < g(n); j++)
9          assert(false); //line 17
10     return 0;
11 }
```

Fig. 4. Correctness certificate of a Simulator built with input $n = 256$. Assertion in line 9 anticipates hitting the frontier originally at line 17.

The execution time for the approach (T_A) is:

$$T_A = simB + certB + \sum_{t_i \in tests} cert(t_i) + \sum_{t_i \in certOK} sim(t_i)$$

where $simB$ and $certB$ refer to the build time of the simulator and certificate, $cert(t_i)$ is the certificate execution time for test input t_i, $certOK$ is the set of certified tests, and $sim(t_i)$ is the simulator execution time for input t_i.

The net time gain of our approach can be expressed as $\left(\sum_{t_i \in test} program(t_i)\right) - T_A$. To illustrate the concept, suppose that the original program code can be partitioned into the $simulatorCode$ and the $slicedCode$. Since the simulator code is essentially executed by both the original program and our approach, the net gain obtained can be approximated by: $\sum_{t_i \in certOK} slicedCode(t_i) - T_A$. In this example, $slicedCode(10^6)$ would include the execution time of the expensive code region (see Figure 2, line 5) `f`(10^6) number of times. Consequently, we define the total gain factor of the approach as the coefficient $\left(\sum_{t_i \in test} program(t_i)\right)/T_A$ and the variable gain factor as $\left(\sum_{t_i \in test} program(t_i)\right)/(T_A - (simB + certB))$ that removes the (constant) building times.

VI. Preliminary Results

Case Studies

We selected two applications to test our approaches (see Figure II): MySQL-5.1.73 (an industrial database), and Olden Benchmark baseline BH (a particle motion simulator). We selected MySQL because it is an actual performance bug reported in [2] and BH because it is a well understood relatively complex application, which has been extensively used to asses program analysis techniques.

The expensive call selected for MySQL is a function call found responsible for the #46011 performance regression [2]. This function creates a temporal table in memory and then

PREPRINT

TABLE I
CASE STUDY RESULTS. (c)/(d): AVERAGE TIME TO SIMULATE/CERTIFY THE PERFORMANCE TESTS. (e): AVERAGE EXEC. TIME FOR ORIGINAL PROGRAM.

Application	Simulator Building (a)	Certificate Building (b)	Simulation Execution (Avg.) (c)	Certificate Execution (Avg.) (d)	Standard Perf. Test Execution (Avg.) (e)	Variable Gain Factor $(\frac{e}{c+d})$	Total Gain Factor $(\frac{e}{a+b+c+d})$
Olden BH	25s	1.2s	0.07s	33.8s	18m 45s	33.21x	18.72x
MySQL-5.1.73	41.4s	55s	1.15s	0s	15m 42s	819.13x	9.65x

TABLE II
CASE STUDY APPLICATIONS.

Application	Size	Language	Purpose
Olden BH	2Kloc	C	Sim. of Particles
MySQL-5.1.73	2.2Mloc	C/C++	Industrial Database

sorts the rows. For Olden BH, since there are no reported performance regressions available, we selected the logging operation as the expensive call (as I/O is generally expensive).

We used a performance test for the MySQL bug taken from those defined by developers in the #46011 [2] ticket. The test creates a table with 1M rows and performs 10k queries on it. For Olden BH, we built the performance tests by picking 100 random values between 1k and 10k for the parameter that determine the program complexity. Additionally, functional tests for both applications were a subset of the performance tests but with reduced input sizes, in order to be fast. Particularly, the BH functional test suite was selected to miss an input value that is known to affect the number of logging calls, this was done for assessing the degree to which the certificate broadens the correctness input space. Experiments were run on a PC with Intel Core i5-4200U 1.60GHz and 8gb of RAM.

Simulation and Certificate Building

The simulation building process for BH was performed by a standard dynamic slicer. Its certificate was built via Program Trimming as in the example in the Section V. The simulator construction was different for MySQL due to standard dynamic slicer scale limitations. We sliced the program manually following an observational slicing [6] iteration: i) execute the application using a profiler, ii) delete the most expensive lines, and iii) compile the sliced application and compare its output against the functional test suite.

For the MySQL certificate construction we relied on a control reachability analysis (due to static slicing scale limitations) and proved that the simulator has an empty frontier set. The reasoning of the reachability analysis goes as follows: if a sliced statement S cannot reach the call E (by control), then S will never be present at the static slice of E. Thus, the simulator built by removing S will have an empty frontier set and will be correct for all inputs, executing in negligible time.

Results and Discussion

The MySQL results (Figure I) show a 9.65x total gain by a simulator constructed by the removal of just 3 lines, suggesting that higher gains could be obtained via further analysis. Moreover, the variable gain factor of 819x indicates that the total gain would likely benefit when the size (and time) of the performance test suite grows, as the building costs will be amortized. On the other hand, the certificate construction and execution for the MySQL case study illustrates the potential impact of the frontier discovery analysis to the overall approach. In terms of regression detection, we found differences in the number of expensive calls for almost all inputs, suggesting that our approach would have flagged the regression successfully.

In the case of BH the total gain factor was higher (18.72x), the difference against MySQL can be explained as the simulator and certificate building were faster and more precise due the application size. However, the variable gain in BH (33.21x) was lower than MySQL as executing a non-trivial certificate took more time in comparison. Additionally, we studied the extent to which the BH certificate deems correct a larger set of inputs. As mentioned in the subjects subsection, the BH functional tests (five tests) were designed to miss an input value, a Boolean, that affects the number of expensive call made. The certified tests for BH were in average 50% of the performance test suite (50), indicating a promising 10x broadening factor. This percentage can be explained as the probability assigned to the Boolean parameters in the suite building was 0.5 for each value.

PRE-PRINT

VII. CONCLUSIONS

In this work we present an approach for rapid detection of performance regressions caused by unexpected increase in the number of expensive calls. Our results suggest that diff-time performance regression may be achievable by further refining strategies for efficient construction of fast simulators and certificates.

ACKNOWLEDGMENTS

Work partially supported by ANPCYT PICT 2014-1656, 2015-3638, 2015-1718; CONICET PIP 2014-0688CO, 2015-0931CO; UBACYT 0419BA, 0297BA; PPL 2011-2-0004

REFERENCES

[1] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," *SCAM'18*.
[2] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *ICSE'14*.
[3] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *FOSE '07*.
[4] J. Silva, "A vocabulary of program slicing-based techniques," *ACM computing surveys (CSUR)*, vol. 44, no. 3, p. 12, 2012.
[5] K. Ferles, V. Wüstholz, M. Christakis, and I. Dillig, "Failure-directed program trimming," in *FSE'17*.
[6] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Orbs: Language-independent program slicing," in *FSE'14*.