Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis

Poster · May 2016

DOI: 10.1145/2884781.2884787

3 authors:

Rahul Gopinath
Oregon State University
13 PUBLICATIONS **25** CITATIONS
SEE PROFILE

Carlos Jensen
Oregon State University
58 PUBLICATIONS **852** CITATIONS
SEE PROFILE

Alex Groce
Oregon State University
83 PUBLICATIONS **2,014** CITATIONS
SEE PROFILE

Topsy-Turvy: A Smarter and Faster Parallelization of Mutation Analysis

Rahul Gopinath
Oregon State University
gopinath@eecs.orst.edu

Carlos Jensen
Oregon State University
cjensen@eecs.orst.edu

Alex Groce
Oregon State University
agroce@gmail.com

ABSTRACT

Mutation analysis is an effective, if computationally expensive, technique that allows practitioners to accurately evaluate the quality of their test suites. To reduce the time and cost of mutation analysis, researchers have looked at parallelizing mutation runs — running multiple mutated versions of the program in parallel, and running through the tests in sequence on each mutated program until a bug is found. While an improvement over sequential execution of mutants and tests, this technique carries a significant overhead cost due to its redundant execution of unchanged code paths. In this paper we propose a novel technique (and its implementation) which parallelizes the test runs rather than the mutants, forking mutants from a single program execution at the point of invocation, which reduces redundancy. We show that our technique can lead to significant efficiency improvements and cost reductions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging Testing Tools

Keywords

software testing, mutation analysis, parallelization

1. INTRODUCTION

Mutation analysis involves the systematic injection of syntactic faults (mutants) into a piece of software, and the quality of a test suite is judged by its ability to find these mutants. The mutation adequacy score characterizes the ability of a test suite to find real bugs [15].

The biggest problem with mutation analysis is the high latency and computational requirements [16]. We propose a solution. When cheap forking system call is available (such as in Unix), postpone mutant creation until the execution of the specific point of code where the mutation has taken place. A simple source code transformation can allow each encounter of a mutation point to result in a fork. The parent process continues execution of the test case, and marks that particular point as having already been mutated. The forked child replaces the particular mutation point with the mutation, and ignores all other mutation points. Thus the parent process simply spawns the mutants that it comes across, but not until the mutant is executed, reducing redundant execution. One advantage of such a procedure is that it can amortize test setup costs. The setup costs of unit tests can be very high. Bell et al. [2] found, in a study of 1,200 large open source projects, that, for most (81% of projects with more than 1000 tests or 71% of projects with more than 1 million LOC) large applications, initialization steps alone are on average 7.18 times as costly as the test execution alone.

Our approach removes the need for separate compilation of mutants, and achieve finer grained coverage-based filtering of mutants than what is available for most tools.

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884787

2. RELATED WORK

According to Mathur [27], the idea of mutation analysis was first proposed by Richard Lipton, and formalized by DeMillo et al. [9] A practical implementation of mutation analysis was produced by Budd et al. [3] in 1980. While mutation analysis belongs to the class of embarrassingly parallel problems [11, 13], the actual parallel and distributed implementation of mutation analysis tools has lagged in practice [8]. Delahaye et al. [8] found that of the eight available mutation tools for Java, Jester [30], Jumble [14], MAJOR [17], and MuJava [22] did not provide any parallelization; Bacterio [24], Javalanche [34], Judy [23], and PIT [7] can parallelize on the same computer, and only Judy has support for distributed evaluation of mutants.

Previous efforts at parallelizing mutation analysis include early work by Krauser et al. [20] who investigated parallelization in an NCube, mutant unification based approaches [28, 29, 32, 33] on vector processing (SIMD) systems (limited to mutations that can be vectorized), and work using MIMD machines [4, 5, 19, 31] that allowed concurrent execution of mutants. Fleyshgakker et al. [10, 36], Just et al. [18], and Ma et al. [21] showed that the state information after execution of mutation used intelligently (lazy mutant analysis) can lead to savings in executions by using only representative mutants. Zapf [37] provided the first implementation for a distributed interpreter for the Mothra mutation system — MedusaMothra. Mateo et al. [25] found that parallel execution with dynamic ranking is the best distribution algorithm.

A related time-reduction technique is the mutant schemata [1,

```
def avg(a, b)
  return (a + b)/2
end
```

```
def avg(x, y)
  return μ(:a, μ(:b, x, y, +), 2, /)
end
```

Figure 1: The original and mutated program *average*

```
def test_avg_eq()
  ⊢ avg(1, 1) = 1
end
```

```
def test_avg_neq()
  ⊢ avg(2, 0) = 1
end
```

Figure 2: The test suite for program *average*

24, 26, 35] approach, where all mutants are encoded into a single *meta-mutant*, and the mutant is chosen by a runtime flag.

3. ALGORITHM

Given a program, we first generate the AST of the subject program and replace each instance of an operator to be mutated with a call to a mutation decision procedure. This procedure takes a unique id that identifies the mutant instance, the operands, and the operator. For instance, Figure 1 shows a simple program for computing the average of two numbers. The *dynamic-mutant* we produce after modifying the AST is shown in Figure 1. Note that unlike a traditional *meta-mutant*, there is little extra compilation required here. Each operator is translated directly into a method call. The tests are all executed separately. That is, for the tests given in Figure 2, each test in the test suite is executed independently (possibly on different machines — our approach does not interfere with traditional test suite parallelization). The call to *avg* from each test calls $\mu(id, a, b, op)$ in turn. On a call to μ, the procedure μ checks if the current process is the parent process (procedure *parent?*). If it is, it checks if the current mutant (variable *id*) has already been spawned (procedure *has?*), in which case, the dynamic-mutant returns the original result for the operator. If not, it iterates through all valid first-order mutations of that operator (e.g. for $+$, we iterate through $-, *, /, \%$ etc.). For each mutation, we fork off a child, mark the particular mutant id as serviced, and return the result. When the child is forked for the first time, the mutant id is associated with the mutant chosen for that child, which is never changed.

```
def μ(id, a, b, op)
  if parent?
    return op(a, b) if has?(id)
    mutations(op).each do |o|
      fork
      if child?
        set(id, o)
        return o(a, b)
      end
    end
    set(id, op)
    return op(a, b)
  else
    o = get(id) || op
    return o(a, b)
  end
end
```

Figure 3: The forking mutants algorithm. Variable *mutations* is a key map of all valid operator replacements.

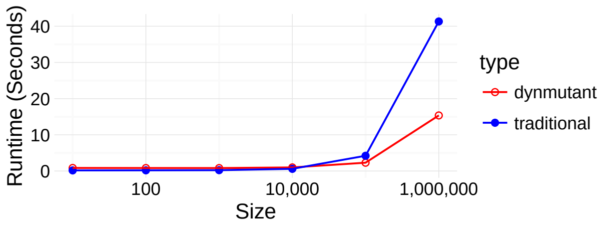

Figure 4: Comparison of traditional and dynamic-mutants *total* runtime against the problem size (x axis).

4. EVALUATION

For the initial evaluation of our technique, we tested the implementation of a simple primality checker. The unit tests for this program require first verifying results for the trivial conditions $(0, 1, 2)$, and then making sure that the program does not fail for large prime and composite numbers. The setup of the program requires creating two large numbers, one prime and other composite. The cost of creating numbers with these specific properties is high, simulating the complex overhead of test setup typical in large real-world projects. The total run time (without considering parallelization) of the traditional mutation technique and for our approach, plotted against the size of primes checked in unit tests, are shown in Figure 4. The figure shows that while initially the traditional mutation technique holds some advantage, that advantage vanishes as the size of the numbers to be checked increases (and with it, setup time). The advantage of our approach is that, once the threshold is crossed, the small penalty in runtime is compensated by the reduction of redundancy in mutant execution. We expect that our approach can, obtain some advantage over traditional mutation analysis in many cases, *even ignoring opportunities for parallelization*, due to the high cost of test setup.

5. CONCLUSION

We propose a novel mutation analysis strategy of evaluating individual test cases separately, with mutants being forked off only when the execution hits the particular mutated expression. Our strategy is extremely simple, and can be implemented easily. Next, like the mutant schemata method, the strategy is only dependent on a source transform, and hence directly applicable to any language, by providing the μ call in a library, and is particularly trivial to implement in languages that support macros or dynamic reflection. Due to the trivial AST modification required, the compilation time required is much less than the traditional schemata method, and comparable to the time taken for the original source. Since mutants are forked off only when their corresponding original instruction is in the execution path, test selection is obtained for free. This is potentially important, and as Delahaye et al. [8] and Coles [6] note, very few mutation tools provide automatic test selection using coverage (note that we do not even require prior coverage information). Note that forking, where it is available, can be very cheap with copy-on-write semantics, and no extra memory is required unless a particular variable is modified. Hence our technique will always be cheaper than the traditional parallelization technique. Finally, our strategy lends itself to distributed execution of individual tests, and with a trivial modification, distributed execution of sets of mutants. The parallelization of mutant evaluation is automatic. Our implementation and tests are available [12].

6. REFERENCES

[1] O. Baruch and S. Katz. Partially interpreted schemas for csp programming. *Science of Computer Programming*, 10(1):1–18, 1988.

[2] J. Bell and G. Kaiser. Unit test virtualization with vmvm. In *International Conference on Software Engineering*, pages 550–561, New York, NY, USA, 2014. ACM.

[3] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233. ACM, 1980.

[4] C. Byoungju and A. P. Mathur. High-performance mutation testing. *The Journal of Systems and Software*, 20(2):135–152, 1993.

[5] B. Choi, A. Mathur, and B. Pattison. Architecture of pmothra: A tool for mutation baaed testing on the hypercube. In *Symposium on Software Testing, Analysis, and Verification*, 1989.

[6] H. Coles. Java mutation systems comparison. http://pitest.org/java_mutation_testing_systems/.

[7] H. Coles. Pit mutation testing. http://pitest.org/.

[8] M. Delahaye and L. du Bousquet. A Comparison of Mutation Analysis Tools for Java. *International Conference on Quality Software*, pages 187–195, July 2013.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[10] V. Fleyshgakker and S. Weiss. Efficient mutation analysis: A new approach. *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 185–195, 1994.

[11] G. C. Fox, R. D. Williams, and G. C. Messina. *Parallel computing works!* Morgan Kaufmann, 2014.

[12] R. Gopinath. Forking mutants. https://github.com/vrthra/forking-mutants.

[13] L. Inozemtseva, H. Hemmati, and R. Holmes. Using fault history to improve mutation reduction. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 639–642. ACM, 2013.

[14] S. Irvine, T. Pavlinic, L. Trigg, J. Cleary, S. Inglis, and M. Utting. Jumble java byte code to measure the effectiveness of unit tests. In *TAICPART-MUTATION*, pages 169–175, Sept 2007.

[15] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[16] N. Juristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1-2):7–44, 2004.

[17] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 433–436. ACM, 2014.

[18] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 315–326. ACM, 2014.

[19] E. Krauser, A. Mathur, and V. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, May 1991.

[20] E. Krauser and A. P. Mathur. Program testing on a massively parallel transputer based system. In *International Symposium on Mini and Microcomputers and their Applications*, pages 67–71, 1986.

[21] Y.-S. Ma and S.-W. Kim. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software*, 2016.

[22] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[23] L. Madeyski and N. Radyk. Judy - a mutation testing tool for java. *Software, IET*, 4(1):32–42, Feb 2010.

[24] P. R. Mateo and M. P. Usaola. Mutant execution cost reduction: Through music (mutant schema improved with extra code). In *International Conference on Software Testing, Verification and Validation*, pages 664–672. IEEE, 2012.

[25] P. R. Mateo and M. P. Usaola. Parallel mutation testing. *Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[26] P. R. Mateo and M. P. Usaola. Reducing mutation costs through uncovered mutants. *Software Testing, Verification and Reliability*, 2014.

[27] A. P. Mathur. *Foundations of Software Testing*. Addison-Wesley, 2012.

[28] A. P. Mathur and E. W. Krauser. Modeling mutation and a vector processor. In *International Conference on Software Engineering*, pages 154–161. IEEE, 1988.

[29] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. *Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-14-P*, 1988.

[30] I. Moore. Jester-a junit test tester. In *Second International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pages 84–87, 2001.

[31] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation testing of software using a mimd computer. In *in 1992 International Conference on Parallel Processing*. Citeseer, 1992.

[32] V. Rego and A. P. Mathur. Concurrency enhancement through program unification: a performance analysis. *Journal of Parallel and Distributed Computing*, 8(3):201–217, 1990.

[33] V. J. Rego and A. P. Mathur. Exploiting parallelism across program execution: a unification technique and its analysis. *IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on*, 1(4):399–414, 1990.

[34] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 297–298, 2009.

[35] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *ACM*

SIGSOFT International Symposium on Software Testing and Analysis, pages 139–148, New York, NY, USA, 1993. ACM.

[36] S. N. Weiss and V. N. Fleyshgakker. Improved serial algorithms for mutation analysis. *ACM SIGSOFT Software Engineering Notes*, 18(3):149–158, 1993.

[37] C. Zapf. Medusamothra: A distributed interpreter for the mothra mutation testing system. Master's thesis, Clemson Univ, 1993.