Toward Speeding up Mutation Analysis by Memoizing Expensive Methods

Ali Ghanbari Andrian Marcus
University of Texas at Dallas, TX 75080, USA
{ali.ghanbari,amarcus}@utdallas.edu

***Abstract*—Mutation analysis has many applications, such as assessing the quality of test cases, fault localization, test input generation, security analysis, etc. Such applications involve running test suite against a large number of program mutants leading to poor scalability. Much research has been aimed at speeding up this process, focusing on reducing the number of mutants, the number of executed tests, or the execution time of the mutants.**

This paper presents a novel approach, named MeMu, for reducing the execution time of the mutants, by memoizing the most expensive methods in the system. Memoization is an optimization technique that allows bypassing the execution of expensive methods, when repeated inputs are detected. MeMu can be used in conjunction with existing acceleration techniques. We implemented MeMu on top of PITest, a well-known JVM bytecode-level mutation analysis system, and obtained, on average, an 18.15% speed-up over PITest, in the execution time of the mutants for 12 real-world programs.

These promising results and the fact that MeMu could also be used for other applications that involve repeated execution of tests (*e.g.*, automatic program repair and regression testing), strongly support future research for improving its efficiency.

***Index Terms*—Memoization, Mutation Analysis, Test Case, JVM**

I. Introduction

Mutation analysis/testing [1] is a program analysis technique that involves generating a pool of program variants, called *mutants*, by systematically mutating (*e.g.*, replacing an arithmetic operator with another) program elements and running the test suite against the mutants. Mutation analysis has been mainly used for assessing test adequacy by computing a *mutation score*, which indicates how good a test suite is for detecting bugs [1]–[3]. In addition, mutation analysis has been used for other purposes, such as fault localization [4]–[6], automatic program repair [7]–[11], test generation [12]–[14] and prioritization [15], program verification [16], [17], *etc.*

Despite its success in some practical use cases [18], [19], mutation analysis suffers from poor scalability. One main reason behind this problem is that the generated mutants must be tested against the test suite, and usually a large number of mutants are generated, making the process lengthy. Much research has been devoted to reducing the cost of mutation analysis [20], [21], focusing primarily on: (1) reducing the number of generated [22]–[24] or executed mutants [25]–[27]; (2) reducing the number of tests [28]–[30] or reordering them [31]; (3) reducing mutant execution time [32]–[35].

In this paper, we focus on reducing mutant execution time. The common aspect of most of the approaches in this category is that they focus, one way or another, on the mutated code. We contend that the execution time during mutation analysis can also be reduced by reducing the execution time of unmutated code. Such a speed-up technique will complement existing acceleration techniques, as they are orthogonal to each other. Specifically, we focus on reducing the execution time of (unmutated) *expensive* methods, *i.e.*, those that have a longer execution time relative to other methods. Given that mutation analysis requires many repeated test executions, and a mutation involves small (usually single-pointed) changes to the program, we expect that unmutated expensive methods are executed frequently. The more frequently these methods are executed, the bigger the time savings will be. In support of our idea, an empirical study (see §II) revealed that the execution of the top 20% most expensive methods account for 43.21% of the mutant testing execution time (in average).

This paper investigates the use of memoization [36] for speeding-up the execution the of expensive methods, in the context of mutation analysis. Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again, and it has been successfully used for speeding up recursive functions [36], [37], optimizing functional programs [38], [39], and eliminating performance bottlenecks [40], [41]. We introduce and evaluate a technique, named MeMu (**Me**moized **Mu**tation analysis), for reducing the execution time of expensive methods during mutation analysis *via* memoization.

Specifically, after identifying the expensive methods, MeMu records a snapshot of the state of the unmutated program at the entry and exit point(s) of the those methods, in the form of input-output pairs and stores them in a *memo-table*. When testing the mutants, upon the invocation of an expensive method, MeMu does a light-weight table look-up to check if a given input has already been recorded in the memo-table. If a match for the given input is found, then it updates the system state with the pre-recorded state, without executing the expensive method. Otherwise, if the input is not in the memo-table (*i.e.*, a *cache miss* occurs), the method is executed.

MeMu is independent of mutant generation or test case selection/reordering and it is meant to be used in conjunction with any existing mutation analysis tool. We implemented and evaluated an instance of MeMu, built on top of the PITest mutation analysis system [42]. As such, the MeMu prototype is usable with JVM-based programming languages.

We used MeMu for analyzing the tests of 12 real-world pro-

grams, resulting in 18.15% speed-up over PITest, in average (min. -0.66%, max. 51.77%), for mutant testing.

For any mutation analysis optimization technique, speed-up comes with two main challenges: (1) limiting the overhead costs; and (2) maintaining true value of mutation score. Our work highlights challenges and solutions in achieving these goals. For example, memoizing all the expensive methods results in significant runtime overhead caused by loading and deserializing large memo-table databases and a large number of cache misses. We also found that memoizing non-deterministic methods adversely affects the mutation score. We introduce a novel technique, that we call *provisional memoization* (see §III), to reduce the size of the memo-table databases and the number of cache misses. Provisional memoization also identifies certain non-memoizable methods, *e.g.*, those that involve non-determinism.

We argue that the use of memoization (with provisional memoization) for speeding-up mutation analysis is promising and we anticipate that future research will further reduce the overhead. Such research is worthwhile pursuing, as MeMu could also be used for speeding up other automated software quality assurance techniques (*e.g.*, [7], [43], [44]), which also rely on the repeated execution of a test suite on the program.

II. Motivational Empirical Study

We conducted an empirical study to understand how much time is used on the repeated execution of the expensive methods, during mutation testing. The premise of our memoization approach is that the execution of the most expensive methods amount to a significant percentage of the total mutant execution time. We used PITest [18], a state-of-the-art JVM bytecode-based mutation analysis system. It offers 29 mutation operators (including commonly-used ones [2]) and performs on-the-fly mutation generation and testing *via* ASM [45] and Java instrumentation API [46], mitigating the compilation and test isolation overhead.

As subjects, we selected 12 real-world programs (see Table I), which are widely used in mutation analysis research [25], [47], [48]. Table I lists the programs, the revisions that we used, and their sizes (number of tests and methods) .

We measured the time it took to generate and test (execute) the mutants. To measure mutant execution time, we calculated the difference between time before and after executing the mutant. By subtracting it from total mutation analysis time, we obtain an approximation of other activities (*e.g.*, mutant generation) performed by PITest. To calculate the execution time of individual methods during mutant execution, we instrumented mutants and injected *before* and *after advises* for using ASM to calculate the difference between time at the entry and exit point(s) of the methods. We used a Dell Workstation with 3.70 GHz CPU and 126 GB of RAM running Ubuntu 18.04.4 LTS. All time measurements are in seconds and are the result of the average of two executions rounded to the nearest integer.

Table I reports the execution time of the top 20% most expensive methods, the execution time of all the methods, and the total time needed by PITest to perform the mutant testing and other activities such as, mutant generation, mutation score computation, *etc.*

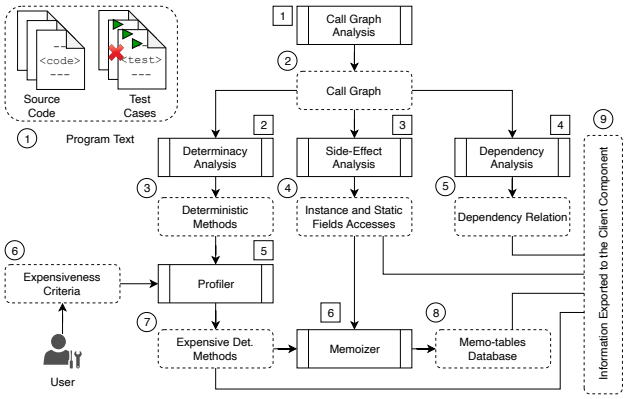

Fig. 1: The memoization component of MeMu. Processes are represented as double-lined rectangles and information produced/consumed by processes using dashed rounded rectangles. Each process uses the program source code and tests as input.

We observe that PITest spends, on average, 69.97% of its time on testing mutants. More importantly, executing the top 20% most expensive methods in the programs accounted, in average, for 43.21% of mutant test execution time (min. 10.11%, max. 78.06%). The findings imply that reducing the execution time of a relatively small fraction of methods (*i.e.*, the 20% most expensive ones) may lead to a significant reduction in the overall mutation analysis time. They serve as motivation for our approach to focus on method-level optimization for reducing the execution time.

III. Memoized Mutation Analysis Framework

MeMu is designed as a framework with two main components: the *memoization* component and the *client* component. The *memoization component* (see Fig. 1) is responsible for identifying and memoizing the expensive methods, and passing this information to the client component. The *client component* can be an existing mutation analysis tool that is modified to intercept the execution of expensive methods identified by the memoization component so as to check whether or not it can reuse the already computed results instead of re-executing the method. We implemented the client component for mutation testing by modifying PITest [42].

We describe the data used and produced (denoted by (x)) by the processes (denoted by [y]) in the memoization component (see Fig. 1) and the client component. In short, to memoize a method, MeMu records a snapshot of the state of the unmutated program at the entry and exit point(s) of the the expensive methods, in the form of input-output pairs and stores them in a *memo-table*. The collection of memo-tables, *i.e.*, the *memo-tables database* (8), is then passed to the client component, which uses it to bypass the execution methods, when a "cache hit" occurs during mutant execution.

It is impractical to memoize all methods, as it leads to large overhead and in many cases the execution time of a method may be actually shorter than a look-up in the memo-table.

Hence, as we discussed before, we focus on memoizing only the expensive methods. Given a *program text* (1) comprised of the source code and a test suite, the framework needs first to determine which methods to memoize.

A. Which methods to memoize?

In order to avoid memoizing non-expensive methods, MeMu uses two user-provided parameters: a *threshold*, τ, and a *limit*, ℓ, which define the *expensiveness criterion* (6). It attempts to memoize the ℓ most expensive methods, with execution time longer than τ milliseconds. However, not all of these expensive methods are memoizable.

The framework applies a *call graph analysis* [1] to obtain the *call graph* (2). In our implementation, we used the WALA program analysis infrastructure [49] to construct a 0-CFA call-graph, but, of course, other tools may also be used. The call graph is then used by additional analyses to determine which of the expensive methods should be memoized.

First, the *dependency analysis* [4] determines the reflexive, transitive closure of the call graph, which is also sent to the client. The resulting *dependency relations* (5) are used for identifying methods that should not be memoized. If the intercepted method (*i.e.*, the one to be memoized) depends on a mutated/modified method or itself undergoes a mutation/-modification, then the method shall not be memoized.

Second, the *determinacy analysis* [2] identifies the methods that depend on time and/or random generator or return values computed in such a manner. We refer to these methods as likely non-deterministic methods. MeMu does not memoize these methods as they might result in large number cache misses (due to the way their input/output is obtained) or change the semantic of programs. The set of methods that are not likely non-deterministic (*i.e.*, *deterministic methods* (3) are used in the next process.

Finally, the *profiler* [5] instruments the system to measure the execution time of the likely deterministic methods and determines the *expensive methods* (7) that will be memoized. It also records coverage information of each test case used for excluding unnecessary test cases, for faster memoization.

B. Memoization

Recording all variables in a memo-table may lead to very large tables. So, before constructing the memo-tables, MeMu filters out fields that are untouched. The *side-effect analysis* [3] determines which method may access which static/instance fields, either directly or by calling another method. To keep the size of memo-tables small and optimize table look-up within the client, the framework only uses the *accessed fields* (4). The *Memoizer* [6] constructs a minimal *memo-tables database* (8) of the methods that are deemed memoizable in the previous steps (*i.e.*, the expensive, deterministic methods). This is done by applying two filtering steps.

First, MeMu determines which methods will not result in failures when memoized. This is achieved via provisional memoization which tentatively memoizes methods and excludes non-memoizable ones. Specifically, we consider a memoization attempt on a method as failed if memoizing the method results in (new) failed tests. In this way, we can single out non-memoizable methods. Second, before passing the *memo-tables database* to the client component, MeMu removes the methods incurring cache misses when they are tested against covering tests. This is done by post-processing the database using the execution information obtained during provisional memoization.

C. Client Component

The client component for MeMu is constructed by modifying PITest such that it loads the memo-tables database in each mutant testing process that PITest forks, and we instrument the mutant code such that the memoizable methods do a light-weight check before proceeding running their bodies. The methods check if they are mutated or depend on some mutated method. If that is the case, no memoization shall take place. Otherwise, they do a light-weight table look-up based on the state of the system at their entry points and update the system state if such a state have occurred previously. Then, the method immediately returns without executing its body.

IV. Empirical Evaluation and Discussion

We conducted an empirical study to assess whether MeMu obtains any speed-up in mutant execution time compared to PITest. We used the same subjects as in the motivational study, described in §II. We set τ and ℓ to 1 ms and 20% of number of methods for each subject program, respectively.

The right hand side of Table I summarizes the information about MeMu's execution.

Comparing PITest's and MeMu's mutant testing time (*i.e.*, the two MT (s) columns in Table I), in 10 out of 12 cases, MeMu completes the execution faster. Excluding jfreechart, MeMu results in 18.15% speed-up (on average - minimum -0.66%, maximum 51.77%) over PITest.

We analyzed the two cases where MeMu did not obtain a speed-up: jfreechart and joda-time. For the jfreechart system, our implementation of MeMu fails to completely restore the system state, so provisional memoization fails to memoize any methods, so we did not perform memoized mutation testing for that subject, hence the "N/A" values the table. The reason is that jfreechart uses graphic libraries that involve system states, which are inaccessible through the Java reflection API [50] used by our framework. We expect that MeMu has the same problem with other similar systems. However, this is not a shortcoming of the idea, rather a consequence of our engineering choice to use reflection and will be addressed in future work.

Thanks to the provisional memoization algorithm, we have been able to exclude non-memoizable methods (see the #MM column). On average, the 11 system (not counting jfreechart) have 3,521 methods (20% of which is 704). MeMu memoizes, no more than 1% of the methods (min. 1, max. 50), yet it results in a considerable amount of time saving.

Provisional memoization ensured that the number of cache misses (#Miss column) is smaller than that of cache hits (#Hit

TABLE I: Result of applying PITest and MeMu on 12 systems. **MT**=mutant testing time, **Score**=mutation score, **#MM**=memoized methods.

Subject Information				PITest Execution Information				MeMu Execution Information			
Project Name	**Rev**	**#Test**	**#Method**	**Total (s)**	**MT All (s)**	**MT Top %20 (s)**	**Score**	**#MM**	**MT (s)**	**#Hit**	**#Miss**
commons-codec	5ef5	851	792	730	418	283	0.863341	21	220	933,291	45
commons-math	0da6	5,246	8,364	47,460	37,429	17,352	0.781098	41	36,280	81,788	8,638
commons-cli	0b45	390	292	140	55	20	0.897196	5	47	8,962	431
commons-csv	f368	306	189	402	282	114	0.841484	3	261	3,998	0
closure-compiler	1dfa	7,907	10,536	64,860	62,665	30,157	0.779994	5	59,493	2,246	54
commons-io	2ae0	132	1,325	2,329	1,682	170	0.805581	1	1,641	2,406	187
commons-fileupload	047f	82	306	275	232	160	0.606796	4	170	3,247	29
jfreechart	2266	2,201	9,110	3,156	1,565	164	0.346555	0	N/A	N/A	N/A
commons-imaging	fd01	93	2,439	4,740	3,954	785	0.422943	7	3,407	475	46
commons-lang	687b	2,295	2,596	1,926	761	594	0.864125	4	367	119	12
joda-time	9a62	4,043	4,366	1,839	1,362	680	0.468076	7	1,371	0	0
commons-geometry-euclidean	b36d	1,628	1,934	6,780	6,390	2,696	0.942488	50	4,595	85,709	3,350

column) for the memoized methods. However, the algorithm is not perfect; for the joda-time system, MeMu is slower than PITest, because the memoized methods do not result in any cache hits during the mutant executions.

Finally, we believe the memoization also resulted in a lossless mutation testing, because for the subject programs with constant mutation scores between runs, the mutation score before and after memoization did not change.

V. Related Work

Conventionally, we classify approaches for reducing mutation analysis costs into three major categories [20], [51]: (1) *do fewer* approaches strive generating/testing as few mutants as possible with minimal adverse effect on mutation score [24]–[28]; (2) *do faster* approaches are meant to generate and run mutants as fast as possible (without any concern about mutation score) [22], [23], [29]–[31], [52], [53]; (3) *do smarter* approaches are intended to distribute the workload of testing mutants into several machines or several cores of a single machine [18], [54], [55], or factor out shared state between mutant executions and avoid re-executing them [32]–[35].

MeMu fits in the third category as it applies a semantic-preserving program optimization method (*i.e.*, memoization in this case) on the unmutated parts of the mutants to avoid re-executing (expensive) methods for which the state of the system at the entry and exit point(s) do not change from one execution to another. We discuss here the works in this category, which we consider most related to MeMu.

Split-stream [32], [34] and its modern variants are intended to avoid repeated execution of part of the code that is shared between mutants. Mutations targeting the same program element, result in many mutants that share the same code before the mutation impact point. Executing this portion of the mutants (provided that the program is deterministic) will always result the same output. Split-stream runs these portions only once and fork different processes for the each mutant *after* the mutation point of impact to test individual mutants. The modern incarnation of split-stream [33] attempts to reuse shared program states even after mutation point of impact.

Just *et al.* [35] propose three runtime optimizations that result in 40% speed up of their MAJOR mutation analysis system [48]: (1) if a mutation does not result in program state change immediately after the mutation point, it marks the corresponding mutant as *survived*, *i.e.*, not killed, and terminates the test execution; (2) even if a mutation infects the system state in an expression while the change does not propagate to the subsequent statements, it marks the corresponding mutant as survived and terminates the test execution; (3) mutants that infect the state of the system in the same way should only be executed once.

Since MeMu optimizes the execution of unmutated code and the memoization does not influence the effect of the mutation, it can complement existing cost reduction techniques. The information collection processes can be parallelized with the pre-processing done by such complementary techniques, in a non-interfering manner, to further speed-up the mutation analysis process.

VI. Conclusions and Future Work

The new idea put forward in this paper is speeding up mutation analysis by automatically memoizing expensive methods. Our optimization is orthogonal to the state-of-the-art cost reduction techniques for mutation analysis and can be used together with them to further speed up the process. An empirical study using state-of-the-art, JVM-based, mutation analysis tool PITest [18] and 12 real-world programs, revealed that 43.21% (avg.) of the mutant execution time is spent on executing the top 20% most expensive methods. This finding supports the intuition behind the memoization-based approach for speeding up mutant execution. An additional empirical study showed that memoizing a small subset of these expensive methods (1% of all methods) leads to an average of 18.15% speed-up during mutant testing. We uncovered two specific issues, important for the successful memoization: identifying non-memoizable methods and minimizing the number of cache misses during testing. Provisional memoization shows promise in tackling these issues. Future work will focus on more light-weight techniques, based on statistical models, which may be less costly than provisional memoization. The other analyses used during memoization can be optimized through parallelization.

We contend that other software quality assurance methods that rely on repeated execution of the code, such as, automatic program repair and regression testing, can also benefit from the memoization idea. Hence, the potential advantages largely exceed those reported here, supporting future work that will further optimize the memoization approach.

Data Availability

Data are available at https://bit.ly/3omErsz.

ACKNOWLEDGMENTS

This research was partially supported by the NSF grants CCF-1910976 and CCF-1955837.

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, pp. 34–41, 1978.

[2] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[3] W. Visser, "What makes killing a mutant hard," in *ASE*, 2016, pp. 39–44.

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *TSE*, pp. 707–740, 2016.

[5] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[6] ——, "Using mutants to locate" unknown" faults," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 691–700.

[7] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *CACM*, 2019.

[8] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*, 2010, pp. 65–74.

[9] A. Arcuri, "Evolutionary repair of faulty software," *ASC*, pp. 3494–3514, 2011.

[10] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*, 2019, pp. 19–30.

[11] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *FSE*, 2017, pp. 593–604.

[12] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *ESE*, pp. 783–812, 2015.

[13] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *IWSBST*, 2016, pp. 45–54.

[14] R. A. DeMillo, A. J. Offutt *et al.*, "Constraint-based automatic test data generation," *TSE*, pp. 900–910, 1991.

[15] D. Shin, S. Yoo, M. Papadakis, and D.-H. Bae, "Empirical evaluation of mutation-based test case prioritization techniques," *STVR*, p. e1695, 2019.

[16] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, and A. Zeller, "Inferring loop invariants by mutation, dynamic analysis, and static checking," *TSE*, pp. 1019–1037, 2015.

[17] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney, "How verified is my code? falsification-driven verification (t)," in *ASE*, 2015, pp. 737–748.

[18] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 449–452.

[19] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney, "Applying mutation analysis on kernel test suites: an experience report," in *ICSTW*, 2017, pp. 110–115.

[20] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *JSS*, vol. 157, p. 110388, 2019.

[21] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, pp. 649–678, 2010.

[22] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ISSTA*, 1993, pp. 139–148.

[23] P. R. Mateo and M. P. Usaola, "Mutant execution cost reduction: Through music (mutant schema improved with extra code)," in *ICST*, 2012, pp. 664–672.

[24] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *JSS*, pp. 185–196, 1995.

[25] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *ISSTA*, 2016, pp. 342–353.

[26] D. Mao, L. Chen, and L. Zhang, "An extensive study on cross-project predictive mutation testing," in *ICST*, 2019, pp. 160–171.

[27] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans, "Automata language equivalence vs. simulations for model-based mutant equivalence: An empirical evaluation," in *ICST*, 2017, pp. 424–429.

[28] L. Chen and L. Zhang, "Speeding up mutation testing via regression test selection: An extensive study," in *ICST*, 2018, pp. 58–69.

[29] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *ISSTA*, 2012, pp. 331–341.

[30] M. Gligoric, V. Jagannath, and D. Marinov, "Mutmut: Efficient exploration for mutation testing of multithreaded code," in *ICST*, 2010, pp. 55–64.

[31] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *ISSTA*, 2013, pp. 235–245.

[32] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *SPE*, pp. 685–718, 1991.

[33] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *ISSTA*, 2017, p. 295–306.

[34] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden, "Muvm: Higher order mutation analysis virtual machine for c," in *ICST*, 2016, pp. 320–329.

[35] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *ISSTA*, 2014, pp. 315–326.

[36] D. Michie, ""memo" functions and machine learning," *Nature*, pp. 19–22, 1968.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[38] H. Xu, C. J. Pickett, and C. Verbrugge, "Dynamic purity analysis for java programs," in *PASTE*, 2007, pp. 75–82.

[39] A. Heydon, R. Levin, and Y. Yu, "Caching function calls using precise dependencies," in *PLDI*, 2000, pp. 311–320.

[40] L. Della Toffola, M. Pradel, and T. R. Gross, "Performance problems you can fix: A dynamic analysis of memoization opportunities," *OOPSLA*, pp. 607–622, 2015.

[41] P. J. Guo and D. Engler, "Using automatic persistent memoization to facilitate data analysis scripting," in *ISSTA*, 2011, pp. 287–297.

[42] M. Delahaye and L. Du Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *SPE*, pp. 875–891, 2015.

[43] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: an evolutionary test approach for java," in *ICST*, 2010, pp. 185–194.

[44] M. Z. Gligoric, "Regression test selection: Theory and practice," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2015.

[45] E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: a code manipulation tool to implement adaptable systems," *AECS*, 2002.

[46] O. Corporation, "Java Instrumentation API," 2004, accessed: 10/20. [Online]. Available: https://bit.ly/3czmzFV

[47] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ISSTA*, 2009, pp. 69–80.

[48] R. Just, F. Schweiggert, and G. M. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a java compiler," in *ASE*, 2011, pp. 612–615.

[49] J. Dolby, S. J. Fink, and M. Sridharan, "Tj watson libraries for analysis (wala)," 2015. [Online]. Available: https://bit.ly/3jWm8Jn

[50] O. Corporation, "Trail: The Reflection API," 2020, accessed: 10/20. [Online]. Available: https://bit.ly/37niPHJ

[51] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[52] W. E. Howden, "Weak mutation testing and completeness of test sets," *TSE*, pp. 371–379, 1982.

[53] V. H. Durelli, J. Offutt, and M. E. Delamaro, "Toward harnessing high-level language virtual machines for further speeding up weak mutation testing," in *ICST*, 2012, pp. 681–690.

[54] R. Gopinath, C. Jensen, and A. Groce, "Topsy-turvy: a smarter and faster parallelization of mutation analysis," in *ICSE-Companion*, 2016, pp. 740–743.

[55] N. Li, M. West, A. Escalona, and V. H. Durelli, "Mutation testing in practice using ruby," in *ICSTW*, 2015, pp. 1–6.