Faster Mutation Analysis with MeMu

Ali Ghanbari
alig@iastate.edu
Iowa State University
Ames, Iowa, USA

Andrian Marcus
amarcus@utdallas.edu
University of Texas at Dallas
Richardson, Texas, USA

ABSTRACT

Mutation analysis is a program analysis method with applications in assessing the quality of test cases, fault localization, test input generation, security analysis, *etc*. The method involves repeated running of test suites against a large number of program mutants, often leading to poor scalability. A large body of research is aimed at accelerating mutation analysis *via* a variety of approaches such as, reducing the number of mutants, reducing the number of test cases to run, or reducing the execution time of individual mutants.

This paper presents the implementation of a novel technique, named MeMu, for reducing mutant execution time, through memoizing the most expensive methods in the system. Memoization is a program optimization technique that allows bypassing the execution of expensive methods and reusing pre-calculated results, when repeated inputs are detected. MeMu can be used on its own or alongside existing mutation analysis acceleration techniques. The current implementation of MeMu achieves, on average, an 18.15% speed-up for PITest JVM-based mutation testing tool.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Mutation Analysis, Mutant, Memoization, Test Case, Method

ACM Reference Format:
Ali Ghanbari and Andrian Marcus. 2022. Faster Mutation Analysis with MeMu. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3533767.3543288

1 INTRODUCTION

Mutation analysis [5] is a program analysis method with a variety of applications. The method is mainly used for assessing test suite quality by computing a *mutation score*, indicating how good a test suite is in detecting bugs [2]. Mutation analysis has also been used for many other purposes, such as fault localization [17], automated program repair [4], test generation [6] and prioritization [20], program verification [7], *etc*. Mutation analysis involves generating a pool of program variants, called *mutants*, by systematically mutating program elements (*e.g.*, by flipping a comparison operator) and running the test suite against the mutants.

Despite the success of mutation analysis in some practical applications [1], the method suffers from poor scalability. This is mainly because the generated mutants must be tested against the test suite, while usually a large number of mutants are generated, making the process extremely time-consuming. So far, a large body of research has been devoted to reducing the cost of mutation analysis [19]. The developed techniques focus mainly on reducing the number of generated [21] or executed mutants [23], reducing the number of test cases to be executed [11] or reordering them [24], and directly reducing mutant execution time [14].

This paper presents the engineering details of a Java-based novel mutation analysis technique, named MeMu [10]. MeMu is based on the idea of reducing mutation analysis time through reducing the execution time for the individual mutants. MeMu optimizes the unmutated code without compromising the semantics of the mutated code, which makes MeMu a lossless mutation analysis acceleration technique, *i.e.*, MeMu does not impact the mutation score. The nature of the optimization done by MeMu also allows it to work alongside existing acceleration techniques and complement them. MeMu uses memoization [15] to accelerate the execution of the expensive methods.

MeMu focuses on reducing the execution time of unmutated *expensive* methods, *i.e.*, those that have a long 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, the expectation is that unmutated expensive methods are executed frequently with the same input as they would have received in the original program. The more frequently these methods are executed, the bigger the time savings will be. Our observation that the top 20% most expensive methods account for 43.21% of the mutant testing execution time [10], which further motivates pursuing this idea.

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 carries out a lightweight table look-up to check if a given input has already been recorded in the memo-table associated with the method. If a match for the given input is found, then it updates the system state with the pre-recorded state and bypasses the method body. Otherwise, *i.e.*, if the input is not in the memo-table and a *cache miss* occurs, the method will be as executed as normal.

MeMu is independent of mutant generation or test case selection/reordering, and it is possible to use MeMu in conjunction with existing mutation acceleration techniques and analysis tools. In this

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3543288

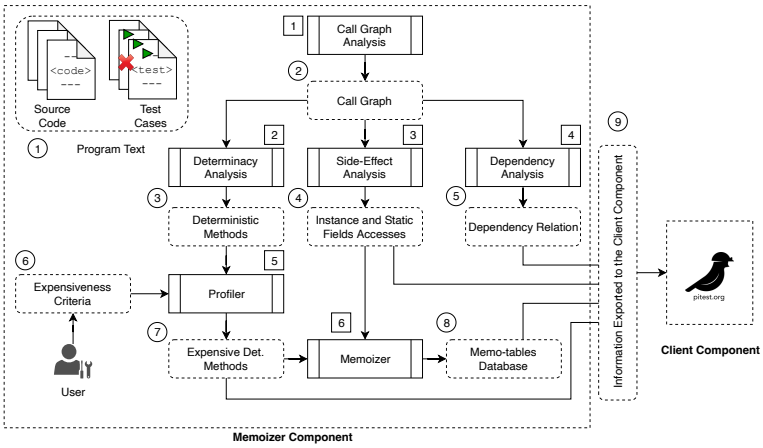

Figure 1: MeMu architecture; processes are represented as double-lined rectangles and information produced/consumed by processes using dashed rounded rectangles. Each process uses the program source code and tests as input. The two main components of the framework are memoization component and client component.

paper we link MeMu with PITest [18], a well-known JVM bytecode-level mutation analysis system. As such, the MeMu prototype is usable with JVM-based programming languages.

Any mutation analysis acceleration approach faces two main challenges: (1) limiting the overhead costs; and (2) maintaining true value of mutation score. In a previous work [8], we highlighted the challenges and solutions in achieving these goals. Specifically, we argued that 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 introduced a novel technique, called *provisional memoization*, 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. Furthermore, we applied MeMu on 12 real-world Java projects and observed that it reduces the mutation analysis costs by 18.15%, on average, without affecting the mutation score.

The users of mutation analysis tools, test-based automated program repair systems, and regression testing tools make the target user group for MeMu. The source code of the tool is publicly available on GitHub [9].

2 RELATED WORK

Traditionally, approaches for reducing mutation analysis costs are classified into three major categories [19]: (1) *do fewer* approaches strive generating/testing as few mutants as possible with minimal adverse effect on mutation score; (2) *do faster* approaches are meant to generate and run mutants as fast as possible; (3) *do smarter* approaches are intended to distribute the workload of testing mutants into several machines or several cores of a single machine, or factor out shared state between mutant executions and avoid re-executing them. MeMu fits in the third category as it applies a semantic-preserving program optimization method (*i.e.*, memoization) 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 most related work to MeMu.

Split-stream [14], and its modern incarnations [22], avoid repeated execution of part of the code that is shared between mutants. Mutations targeting the same statement, result in many mutants that share 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.

Just *et al.* [12] propose three runtime optimizations that result in 40% speed up of their MAJOR mutation analysis system [13]: (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.

3 MEMU FRAMEWORK

MeMu [10] is a framework with two main components: the *memoization* component and the *client* component (see Figure 1). The *memoization component* 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 have implemented a client component for mutation testing by modifying PITest [18].

We describe the data produced and used (denoted by (x)) by the processes (denoted by [y]) in the memoization component and the client component. In summary, 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 of the methods, when a "cache hit" occurs during mutant execution.

We do not want to memoize all the methods, as it leads to large overhead, and in many cases the execution time of a method may actually be 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.

3.1 Finding Memoizable Methods

In order to avoid memoizing light-weight methods, MeMu uses two user-provided parameters: a `<threshold>` parameter and a `<limit>` parameter, that define the *expensiveness criterion* (6). It then attempts to memoize the `<limit>` most expensive methods, with execution time longer than `<threshold>` 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 a precise dynamic call graph construction algorithm. 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 of 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 a faster memoization.

3.2 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 through 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 while they are tested against the covering tests. This is done by post-processing the database using the execution information obtained during provisional memoization.

3.3 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.

3.4 Implementation

MeMu uses the ASM bytecode manipulation framework [16], and Java Agent technology [3], for instrumenting the program and implementing various dynamic analyses. The information collected from the dynamic analyses are processed using scripts written in Datalog, for a faster and more maintainable codebase. MeMu is implemented as a Maven plugin. MeMu's codebase is publicly available on GitHub [9].

4 USING THE TOOL

The memoizer component for MeMu is implemented as a one-click Maven plugin and is publicly available [9]. The executables for the PITest-based client component are also available as Maven plugins. The companion website [9] provides detailed instructions on how to install MeMu and its client component.

After installation, setting up MeMu is as easy as adding the following XML snippet under the `<plugins>` tag in the POM file for the target project.

```
<plugin>
    <groupId>edu.iastate</groupId>
    <artifactId>memoizer-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
</plugin>
```

This will allow invoking the memoizer component on the project. By default, the memoizer plugin will select top 15 methods that take 10 milliseconds or more as memoization candidates. To change these

Table 1: Summary of MeMu options configurable through the POM file

Option	Descriptions
<threshold>	Threshold in milliseconds; all the methods costlier than threshold shall be considered for memoization (10, by default)
<limit>	Maximum number of methods satisfying threshold to be memoized (15, by default)
<targetClasses>	Target production classes to be transformed (by default, ${groupId}*, *i.e.*, all application classes)
<excludedClasses>	Target application classes to be excluded from transformation (by default, all test cases are selected, *i.e.*, *Tests, *Test, *TestCase*)
<targetTests>	Target test classes to be included (by default, *Tests, *Test, and *TestCase*, *i.e.*, all classes that end with Tests or Test, or contain the word TestCase)
<excludedTests>	Target test classes to be excluded (empty, by default).

default values, the user can set the values of parameters <limit> and <threshold>, under the tag <configuration> of the plugin. Other parameters of the plugin can also be changed. Table 1, lists all the parameters of the memoizer plugin and their default values.

After setting up the POM file, the memoizer plugin can be invoked from command-line using the following command.

```
mvn edu.iastate:memoizer-maven-plugin:memoize
```

Please note that all the production and test classes of the project must be compiled before invoking the above-mentioned command.

PITest's official website [18] contains details about how to configure it as a Maven plugin, as we refer the reader to the website for more details. For the purposes of using PITest as a client component for MeMu, we can use it with its default settings:

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest</artifactId>
    <version>1.3.2</version>
</plugin>
```

Invoking PITest should be proceeded as usual [18]:

```
mvn org.pitest:pitest-maven:mutationCoverage
```

5 LIMITATIONS AND FUTURE WORK

MeMu employs a combination of static and dynamic analyses that are *overhead* to the actual mutation analysis. The example that we have shipped in the companion website of the project is carefully crafted to showcase the speed-up gained by applying MeMu. However, in our previous work [10], we observed that for some Java projects the overhead outweighs the speed-up that we gained from memoized mutation analysis.

Applying MeMu on a set of real-world Java projects with many expensive methods (*e.g.*, projects involving numerous REST API invocations that cannot be mocked away) is planned as future work. We believe that with more sophisticated and efficient analyses, this overhead can be reduced, making MeMu effective on a wider range of Java projects.

6 CONCLUSIONS

This paper presents the engineering details of MeMu [10], a novel technique for accelerating mutation analysis through the memoization of expensive methods in the program. MeMu's optimization strategy is complementary to existing state-of-the-art cost reduction techniques for mutation analysis. They can be used together for further optimization. MeMu is publicly available [9].

REFERENCES

[1] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E McKenney. 2017. Applying mutation analysis on kernel test suites: an experience report. In *ICSTW*. 110–115.
[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.
[3] Oracle Corporation. 2004. Java Instrumentation API. https://bit.ly/3czmzFV Accessed: 105/22.
[4] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *ICST*. 65–74.
[5] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* (1978), 34–41.
[6] Richard A DeMillo, A Jefferson Offutt, et al. 1991. Constraint-based automatic test data generation. *TSE* (1991), 900–910.
[7] Juan P Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2015. Inferring loop invariants by mutation, dynamic analysis, and static checking. *TSE* (2015), 1019–1037.
[8] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *ISSTA*. 19–30.
[9] Ali Ghanbari and Andrian Marcus. 2020. Faster Mutation Analysis with MeMu. https://github.com/ali-ghanbari/memu-demo Accessed: 05/22.
[10] Ali Ghanbari and Andrian Marcus. 2021. Toward Speeding up Mutation Analysis by Memoizing Expensive Methods. In *ICSE-NIER*. 71–75.
[11] Milos Gligoric, Vilas Jagannath, and Darko Marinov. 2010. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *ICST*. 55–64.
[12] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA*. 315–326.
[13] Rene Just, Franz Schweiggert, and Gregory M Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE*. 612–615.
[14] Kim N King and A Jefferson Offutt. 1991. A fortran language system for mutation-based software testing. *SPE* (1991), 685–718.
[15] Donald Michie. 1968. "Memo" functions and machine learning. *Nature* (1968), 19–22.
[16] OW2 Consortium. 2020. ASM Bytecode Manipulation Framework. http://bit.ly/3fsPL2r. Accessed: 05/22.
[17] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.
[18] PIT Contributors. 2022. PIT Mutation Testing. https://bit.ly/36Xvhho Accessed: 05/22.
[19] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *JSS* 157 (2019), 110388.
[20] Donghwan Shin, Shin Yoo, Mike Papadakis, and Doo-Hwan Bae. 2019. Empirical evaluation of mutation-based test case prioritization techniques. *STVR* (2019), e1695.
[21] Roland H Untch, A Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation analysis using mutant schemata. In *ISSTA*. 139–148.
[22] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster Mutation Analysis via Equivalence modulo States. In *ISSTA*. 295–306.
[23] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive Mutation Testing. In *ISSTA*. 342–353.
[24] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*. 235–245.