muJava Performance Enhancements

Abhiroop Kodandapursanjeeva

Department of Electrical and Computer Engineering, The University of Texas at Austin

***Abstract*— Mutation testing is a software testing technique that quantifies the effectiveness of a test suite in detecting artificially placed bugs in software. A major drawback of mutation testing is the high computational cost, which often leads to extremely long times required to execute certain mutation testing tools. This paper presents two performance upgrades for muJava, a mutation testing tool for Java classes. Evaluation of the upgraded tool is performed and presented to show that these two new features resulted in significant reduction of total execution time.**

I. Introduction

Software testing is a widely used technique that attempts to validate software by ensuring code operates as expected when utilized in various scenarios. Essentially, the purpose of testing is to reveal defects or unacceptable limitations of software before deployment [1]. The simplest way of testing software is by using unit tests. Unit tests are used to validate the correctness of smaller modules of code. For example, unit tests can be written to validate the correctness of individual methods in a Java class. Such tests should be designed to detect any possible defects of the code, which may not be straightforward when code is extremely complex. Test suites (collection of test cases for a common purpose) are therefore only as effective as the thoroughness of the test cases they run. It is thus important for developers to write thorough tests, to maximize their confidence in validated code and limit the number of possible undetected bugs.

Mutation testing is a technique that goes beyond standard software testing. The goal of mutation testing is to quantify the effectiveness of a test suite in detecting code defects. To do this, artificial bugs are placed into the code to mimic the type of bugs that a real developer might write. These bugs are often small syntactic differences that most developers would not catch by eye, and would thus rely on the test suite to detect. The process of mutation testing is often separated into two phases: mutant generation and mutant testing. Note that the term ”mutant” refers to a version of the original code with a single bug. Mutant generation is the process of changing or adding minor syntactic elements of code to the original code. This can be accomplished either by parsing and modifying the source code, or even by modifying the compiled code directly (for example, Java bytecode). Mutant testing is the process of running the test suite such that it utilizes a mutant instead of the original code. If a single test case fails when using the mutant, the mutant is termed ”killed” to signify it was successfully detected by the test suite. If no test cases fail, the mutant is termed ”live”. From this information, the mutation score can be calculated as $\frac{\text{killed mutants}}{\text{total mutants}}$. A higher mutation score generally means the test suite being examined is more effective at detecting syntactic bugs. Live mutants are useful in determining how the test suite can be extended to ensure a similar bug can be detected in the future, if developers determine the mutant to be realistic.

A major drawback of mutation testing is its possible high computational cost. Large numbers of mutants can be

generated from relatively small code sizes, and testing these mutants against a large number of test cases can lead to extremely high execution times. Various techniques to reduce these costs are discussed by Jia and Harmon [2]. Among these include the concept of ”do fewer”, which proposes that it is often sufficient to generate only a subset of all possible mutants, since many mutants are unlikely to be written by developers in practice. Other techniques in this category include eliminating equivalent mutants, and intelligently selecting test cases for a mutant to minimize wasteful execution. For example, if a mutant exists in a particular line of the code, then running a test case that does not cover this line in the original code may be wasteful, as this bug will not be executed by the test. Simpler techniques involve utilizing more advanced computer hardware to speed up execution times.

muJava is a mutation testing tool that was made for mutating Java classes and running JUnit test cases [3]. It is considered a relatively slow tool, as it does not have any ”intelligent” features to minimize execution times. For example, muJava needs to parse Java source code in order to generate mutants. This necessitates the compilation of the mutant source codes as an additional time-costly step before running tests. Faster tools can directly mutate the compiled code and skip the compilation step. In addition, mutant source code and bytecode data is stored in the file system following generation/compilation. Bytecode would then need to be read from the file system during the testing phase. Reading and writing from the file system for a large number of mutants carries large time overhead and further contributes to the high time cost of mutation testing.

II. Project Description

To increase the execution time of muJava, we implement two features: multithreading and storing mutant data in memory. Multithreading was used in all three phases of muJava (mutant generation, mutant compilation, and testing) to maximize usage of available computing power. The Java concurrent package makes multithreading relatively simple. Individual tasks can be created as callable objects and submitted to an ExecutorService to run asynchronously. For this project, we use the WorkStealingPool as the ExecutorService, which offers a unique way of dividing work between active threads. The WorkStealingPool will divide tasks into separate queues for each thread, and threads can pop tasks from their own queue head to execute when they are available for work. If a thread’s queue becomes empty, it will try to ”steal” a task from the tail of another thread’s queue. This minimizes contention between threads as compared to using a single task queue, and allows for free threads to accept tasks that other threads have not started. Mateo and Usaola [4] discuss many other techniques for parallelizing mutation testing, though their work focuses on distributed systems rather than multithreaded systems.

A. Multithreading

During mutant generation, each user-selected mutation operator is identified as an individual task. For this project, we focus on the method-level mutation operators, which muJava has 16 of. Therefore, maximum parallelization for this stage is limited to 16 threads. Since it is unknown how many mutants each operator will produce, each task will take an unknown amount of time to complete. The WorkStealingPool is very useful in this case to maximize processor usage when certain threads take much longer than others to finish their tasks. Each thread for this phase will use Java Reflection to create a unique Mutator object for their

operator, and complete the task of parsing the source code and generating mutants. Mutant compilation is parallelized simply by creating a task for each mutant source code created in the previous step. Maximum parallelization for this phase is now only limited by the number of threads in the WorkStealingPool.

Finally, each mutant needs to be run against the test cases and marked as killed or live. This phase was parallelized by creating a task for each mutant. Specifically, muJava will iterate over each method of a particular class and run the mutants of the methods separately. Now, the mutants of each method will be tested in parallel, limited by the number of threads in the WorkStealingPool. A complication here is the need for threads to utilize different mutant class definitions concurrently in the same JVM. This was accomplished by allowing each task to use a custom classloader to load both the mutant class **and** the JUnit test class before running tests. Now, threads can correctly run their mutant class against the JUnit tests and determine if it was killed or live. One issue of testing mutants is the possibility of a mutant containing an infinite loop. This is addressed by having each task spawn a single daemon thread to run the tests, and timeout if the test time exceeds a user-defined limit. A mutant is deemed as killed if a timeout occurs during this phase.

B. Storing data in memory

To avoid the overhead of file operations, mutant data was stored in memory rather than in the file system. It is important to note that in the original tool, the mutant generation and mutant testing phases ran from separate Java programs, making the file system necessary for storing generated mutants. We construct a CLI program that handles both generation and testing for a specified class within a single process. Doing this allows us to write mutant data (source code and bytecode) to in-memory data structures, which saves much time as compared to the file system. Threads that generate mutant source code will simply write the code to a ConcurrentHashMap, and threads that compile the source code will redirect compiler output to another ConcurrentHashMap. In the next phase, mutant testing threads will be given the bytecode of the mutant they need, allowing them to skip the process of loading the bytecode from a file.

III. Results

Evaluation of the new features was done by running the CLI tool for a single Java class and corresponding JUnit test cases. The tool will generate as many method-level mutants as possible for the class, and then run each mutant against the test cases provided. The number of threads utilized by the tool is varied to observe how the parallelism level affects tool performance. Performance of the original tool is also measured to compare with the upgraded version. We used an open source scientific Java library called jscience [5] for material to test. Specifically, a Java class for calculus functions was used for the actual mutation testing. This class contained 213 non-whitespace lines of code, and 9 public methods. A JUnit test case generator called Randoop [6] is utilized to generate 383 test cases for the entire Calculus class, utilizing all public methods of the class. Timing data was recorded for all three phases of the tool's execution, as well as the total time. The system used for testing was the UT Austin's ECE department's Kamek Linux Server. This machine has 64 total processors for use. Note that for evaluation, using 0 threads means to use the original muJava tool without any updates. Using 1 thread will use the in-memory feature but will not use any multithreading. This is to better understand how the in-memory feature alone affects execution time.

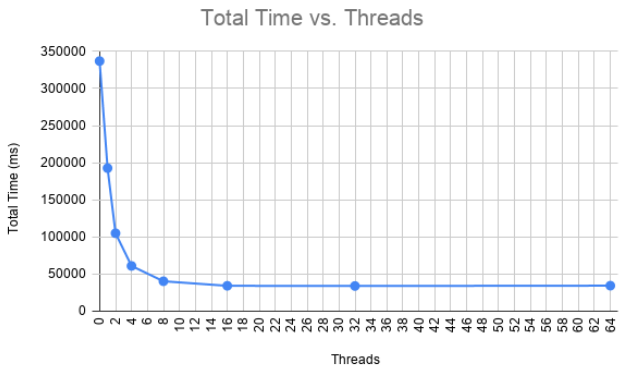

Fig. 1: Total Execution Time

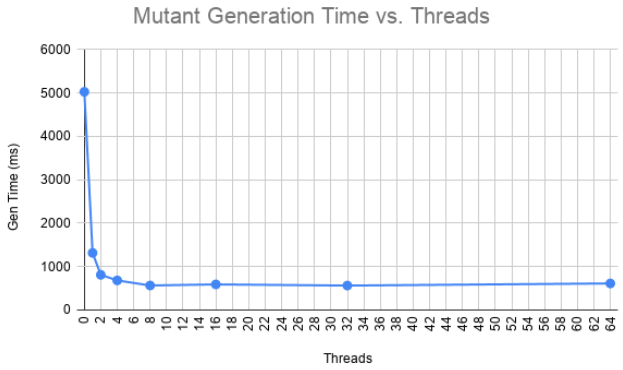

Fig. 2: Mutant Generation Time

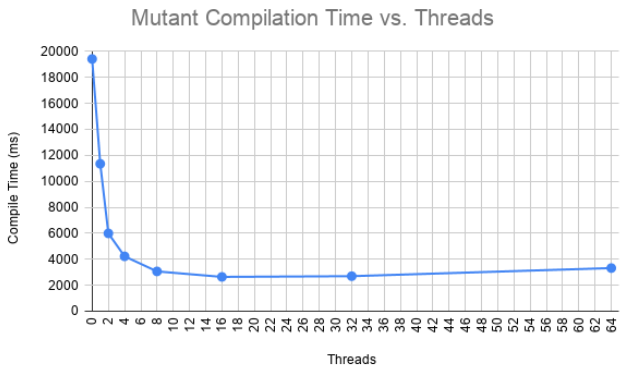

Fig. 3: Mutant Compilation Time

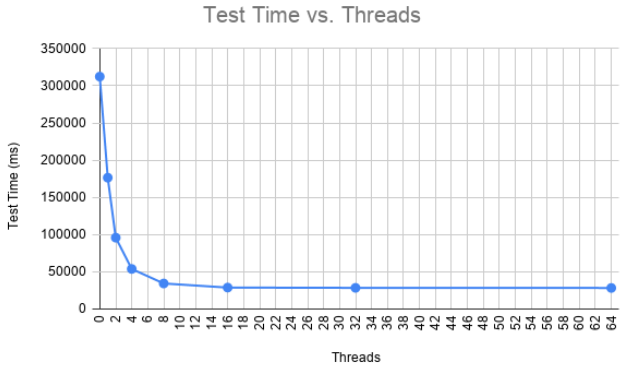

Fig. 4: Mutant Testing Time

Figure 1 shows that by using the new features, muJava's total execution time was reduced by a factor of about 10. This can be considered a significant improvement in performance. Timing data of each phase also shows a similar trend, where every phase operates at most 10x faster than the original tool. For this particular class and test set, the testing phase comprised much of the total execution time. This can be explained by the fact that 383 total tests may be considered excessive for a class of just 9 methods. In addition, Randoop's test generation created very complex test cases, many consisting of over 50 total statements. In total, the test cases comprised nearly 22000 LOC.

Figure 5 shows the speedup attained in each phase from the different number of threads used. We can observe that the generation phase benefited most from the in-memory feature, operating 3.8x faster with this feature alone. All phases show faster execution with more threads, though the testing phase certainly benefits the most from multithreading. This phase alone saw speedups of 11x by using 64 threads. Since this phase also comprises much of the total test time for this example, the total time's speedup largely followed the speedup of this phase.

Threads	Generation	Compilation	Testing	Total
0	1	1	1	1
1	3.817767654	1.711126773	1.769781076	1.748190022
2	6.199753391	3.239326217	3.256477602	3.207898744
4	7.361639824	4.594984623	5.798923534	5.520931604
8	8.883392226	6.332898598	9.088809378	8.346191938
16	8.507614213	7.346066566	10.82701504	9.833887625
32	8.914893617	7.196369026	10.96340222	9.904389728
64	8.20228385	5.845019561	11.02180048	9.786778155

Fig. 5: Normalized Speedups by Phase

IV. Future Work

Modern mutation testing tools utilize more advanced features to further decrease execution time. For example, PIT is a popular Java mutation testing tool that operates by modifying bytecode of classes directly [7]. This allows the tool to skip the compilation phase, saving execution time. muJava would certainly benefit from this technique, though replacement of source code parsing with bytecode manipulation may be difficult to implement.

PIT also utilizes test selection to only run the test cases for each mutant that are most likely to execute the mutant line. For a given mutant, this can eliminate much wasteful execution of test cases, further decreasing execution time. PIT also isolates mutants during testing by launching child JVM processes for a given number of mutants. Each JVM is provided a set of tests and mutation identifiers to modify the mutant classes using the Java Instrumentation API. Launching JVMs is known to be an expensive process, and allowing a single JVM to handle multiple mutants can lead to errors when dealing with static members of the class [7]. This is in contrast to our tool, which utilizes custom classloaders to load the mutants in each thread, creating total isolation between threads while not needing to launch separate JVMs. It would be worthwhile to investigate how this difference in mutant isolation affects test execution time.

Various other techniques to improve performance are discussed by Jia and Harmon [2]. The most relevant to muJava is the idea that only a portion of the total possible number of mutants that can be generated are needed. The logic of this is that many generated mutants would not realistically be written by actual developers. Eliminating unrealistic mutants can be achieved by selecting only realistic mutation operators for generation, or by sampling only a fraction of all mutants generated. For example, randomly sampling 20% of the total generated mutants for testing may be sufficient in generating a meaningful mutation score, and would likely eliminate many unrealistic mutants. This technique however relies on the assumption that most mutants generated are not considered meaningful.

Petrovic and Ivankovic [8] discuss an approach to improving mutation testing performance by using developer feedback and code coverage metrics. The idea is to limit mutant generation only to testable and ”interesting” lines of source code. This technique is used in Google to allow for scalable mutation testing in real-world settings. Implementing such an approach in muJava may be quite difficult, as it would need major replacement of the mutant generation phase. Currently, muJava uses a library called OpenJava to parse source code, though this library has been long deprecated and is extremely difficult to work with. Using a modern source code parser, or preferably using a bytecode manipulator would be greatly beneficial for both performance and the ability to add new features.

V. Conclusion

Mutation testing can be an extremely expensive process both in computational costs and time. We develop and present two new features of a mutation testing tool called muJava that seek to decrease execution time. The features use in-memory data structures to eliminate the need for high-quantity file operations, and maximize computing power utilization with multithreading. Our evaluation on a simple mathematics-related Java class show that these two features allow for significant reduction in execution time, by a factor of nearly 10. Further improvements in execution time for muJava can be achieved through more advanced features, as discussed in Future Work. Implementing these features on

an outdated tool such muJava however may be redundant, as more modern and advanced mutation testing tools already exist which provide the mentioned features.

REFERENCES

[1] K. Sneha and G. M. Malle, "Research on software testing techniques and software automation testing tools," in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pp. 77–81, 2017.

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[3] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, p. 97–133, June 2005.

[4] P. R. Mateo and M. P. Usaola, "Parallel mutation testing," *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 315–350, 2013.

[5] "jscience." https://github.com/javolution/jscience, 2017.

[6] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, (New York, NY, USA), p. 815–816, Association for Computing Machinery, 2007.

[7] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java (demo)," pp. 449–452, 07 2016.

[8] G. Petrovic and M. Ivankovic, "State of mutation testing at google," in *Proceedings of the 40th International Conference on Software Engineering 2017 (SEIP)*, 2018.