2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Toward Harnessing High-level Language Virtual Machines for Further Speeding up Weak Mutation Testing

Vinicius H. S. Durelli
Computer Systems Department
Universidade de São Paulo
São Carlos, SP, Brazil
durelli@icmc.usp.br

Jeff Offutt
Software Engineering
George Mason University
Fairfax, VA, USA
offutt@gmu.edu

Marcio E. Delamaro
Computer Systems Department
Universidade de São Paulo
São Carlos, SP, Brazil
delamaro@icmc.usp.br

***Abstract*—High-level language virtual machines (HLL VMs) are now widely used to implement high-level programming languages. To a certain extent, their widespread adoption is due to the software engineering benefits provided by these managed execution environments, for example, garbage collection (GC) and cross-platform portability. Although HLL VMs are widely used, most research has concentrated on high-end optimizations such as dynamic compilation and advanced GC techniques. Few efforts have focused on introducing features that automate or facilitate certain software engineering activities, including software testing. This paper suggests that HLL VMs provide a reasonable basis for building an integrated software testing environment. As a proof-of-concept, we have augmented a Java virtual machine (JVM) to support weak mutation analysis. Our mutation-aware HLL VM capitalizes on the relationship between a program execution and the underlying managed execution environment, thereby speeding up the execution of the program under test and its associated mutants. To provide some evidence of the performance of our implementation, we conducted an experiment to compare the efficiency of our VM-based implementation with a strong mutation testing tool (muJava). Experimental results show that the VM-based implementation achieves speedups of as much as 89% in some cases.**

***Index Terms*—Mutation Analysis; Java Virtual Machine; Software Testing; Maxine VM; muJava.**

I. Introduction

High-level language virtual machines (HLL VMs) have been used to implement high-level programming languages for more than thirty years. Many high-level programming languages have relied on these architecture-neutral code execution environments. HLL VMs are widely used because of their software engineering benefits, including portable program representation, security sandboxing, program isolation, and garbage collection (GC). Likewise, their advantages over statically compiled binaries have led to widespread adoption on platforms ranging from web servers to low-end embedded systems.

Given that HLL VMs are widely used, lots of papers have been written about how to improve them. Most research has concentrated on high-end optimizations such as just-in-time (JIT) compilation and advanced GC techniques [7]. Few efforts have tried to exploit the control that HLL VMs exert over running programs to facilitate and speedup software engineering activities. Thus, this research suggests that software testing activities can benefit from HLL VMs support.

Test tools are usually built on top of HLL VMs. However, they often end up tampering with the emergent computation. Using features within the HLL VMs can avoid such problems. Moreover, embedding testing tools within HLL VMs can significantly speedup computationally expensive techniques such as mutation testing [6].

To evaluate this idea, we augmented a managed execution environment by adding support for mutation testing activities. We used the HLL VM that is by far the most widely used within academic settings [7], the Java virtual machine (JVM) [17]. The implementation used is Maxine VM [21]. Maxine VM is a mature, research-oriented, high-performance HLL VM that has been used in several studies [3, 8, 25]. Instead of the more expensive strong variant of mutation, we use the cheaper approximation, weak mutation.

This paper presents a proof-of-concept, VM-integrated mutation testing environment. The novelty of this approach is not the supporting technologies and tools, but rather their integration into a cutting-edge execution environment. Moreover, it also uses a novel approach to implement weak mutation testing. The experiment shows that major savings in computational cost can be achieved. In the experiment, speedups of over 89% were obtained in comparison to muJava [18].

The remainder of this paper is organized as follows. Section II describes the objectives and rationale behind integrating mutation analysis facilities into an HLL VM. Section II also highlights why we settled on using a JVM implementation as our target HLL VM. Section III provides background on mutation testing. Section IV gives an overview of the internal organization of a JVM. Then the paper gives particulars of the chosen JVM implementation in Section V. Section VI describes the VM-based mutation analysis implementation and Section VII summarizes the experimental results. Section VIII discusses related work, and Section IX suggests future work and makes concluding remarks.

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.158

II. Objectives and Rationale

As already stated, the conventional approach to automating mutation testing is through tools built atop HLL VMs. However, these tools often interfere with the emergent computation by introducing an additional layer in between the program under test and the execution environment. Moreover, during run time[1], when tools need to carry out computations about themselves (or the underlying program under test) they must turn to costly metaprogramming operations (e.g., reflection). To examine a running program, for example, a tool has to perform introspection operations (i.e., inspecting state and structure). Likewise, to change the behavior or structure of the program under test during run time, tools have to resort to intercession [16].

Apart from the overhead incurred by reflective operations, tools that implement weak mutation rely heavily on state storage and retrieval. By storing state information these tools factor out the expense of running all mutants from the beginning. Nevertheless, such computational savings are only possible at the expense of a significantly larger memory footprint.

The rationale behind arguing that HLL VMs provide a sound basis for building an integrated mutation testing environment is that they bear a repertoire of runtime data structures suitable for being tailored to accommodate the semantics of mutation testing. First, by capitalizing on existing runtime structures is possible to decrease the amount of storage space required to implement weak mutation: from within the execution environment is easier to determine what needs to be copied, narrowing the scope down and thus reducing storage requirements. Second, there is no need to resort to costly reflective operations since runtime information are readily available at HLL VM level. Third, by reifying mutation analysis concepts (i.e., turning them into first-class citizens within the scope of HLL VMs) it is easier to take advantage of high-end optimization and memory management features which are usually shipped with mainstream HLL VM implementations, e.g., JIT compilation and GC. Fourth, a further advantage of building on HLL VMs data structures is that they make it possible to exert greater control over the execution of mutants.

The rationale behind settling on using a JVM realization to implement our VM-based mutation analysis environment is that, apart from being by far the most used HLL VM implementation within academic circles [7], implementations of such execution environment have built-in multi-thread support. Therefore, rather than executing each mutant to completion in turn, we set out to take advantage of such infrastructure by forking off the execution of mutants in their own threads, thereby further speeding up their execution. Additional detail on how we explored the fork-and-join model to speed up mutants execution is presented in Section VI.

[1]Following the convention proposed by Smith and Nair [22], in this paper the single-word form *runtime* refers to the virtualizing runtime software in an HLL VM, *run time* refers to the time during which a program is running and *run-time* refers to the amount of time it takes to execute a program.

It also bears mentioning that by integrating mutation analysis facilities into the runtime environment it is possible to take advantage of ahead-of-time compilation. That is, the set of classes that belongs to the core execution environment is usually compiled in an ahead-of-time fashion to generate a bootstrap image. Thus, adding mutation analysis facilities to the core of HLL VMs would make them self contained execution environments that require no dynamic class loading until later stages of execution, which would further boost start-up performance.

III. Weak Mutation Testing

Mutation testing is a fault-based technique [11] that helps testers design test cases to uncover specific types of faults. Faults are small individual syntactic changes induced into a given program. The changed programs, called mutants [19], mimic typical mistakes that programmers make or encourage good tests [6]. Test data are then designed to execute the mutants with the goal of fulfilling the following conditions [2]:

- **Reachability:** The mutated location in the mutant program must be executed;
- **Infection:** The state of the mutant program must be incorrect after the faulty location is executed;
- **Propagation:** The infected state must be propagated to some part of the output.

This is called the *RIP* model. When a test case satisfies the RIP model by causing the output of the mutant to be different from the output of the original program, the test case is said to *kill* the mutant. New tests are run against all live mutants, but not dead mutants. If a mutant has the same output as the original program for all possible test cases, it is said to be *equivalent*. Tests are added and run against live mutants until all mutants are killed or the tester decides the test set is good enough. The mutation score is the percentage of nonequivalent mutants that have been killed [19].

Although powerful, mutation testing requires good automated tools to manage execution of the hundreds of mutants generated by moderate-sized classes. Reducing this computation cost is the goal of several research ideas, including weak mutation [9].

In weak mutation, execution is stopped after the mutation and if infection has occurred, the mutant is marked dead. In other words, the infected state does not need to propagate to the output. This also allows for the possibility of killing more than one mutant in a single execution.

IV. A Primer on the Structure of the JVM

This section describes the structure of the JVM according to its specification [17]. We describe the JVM as an abstract computing machine in terms of its relevant runtime data areas and virtual instruction set architecture. The specific JVM used in this experiment is described in Section V.

The JVM global storage area for holding objects is managed as a *heap*. The heap is created when the JVM initializes and several other data storage structures are allocated from it. One runtime data structure that is allocated from the heap

is the *method area*. The method area stores per-class and per-interface structures such as code for methods and constructors, and is shared among all threads [17]. The method area also stores other per-type runtime representations, that is, *runtime constant pools*. When a class or interface is loaded into the JVM, a runtime constant pool is allocated in the method area. This pool contains several constants and is similar to a symbol table in a conventional programming language [5].

In addition to the global and per-type structures, JVM also contains per-thread and local runtime data structures. These drive the execution of Java programs. Many threads can be spawned at run time. When a new thread is created, it is associated with a *JVM stack* [17], which manages method invocation and returns, and stores local variables and partial results. JVM stacks use two local structures to hold data: a *program counter* (*PC*) *register* and a *frame*. PC registers keep track of the code being executed by the underlying thread. If the current method is not native, the PC register holds the address of the method instruction currently being executed.

Frames are fundamental to method invocation. They are pushed onto JVM stacks at method invocation time. So each frame corresponds to a method invocation that has not returned yet. When control returns from a method, whether it is a normal or an uncaught exception, the frame is popped off the underlying JVM stack. Upon normal method completion, the result of the method computation, if any, is transferred from the current frame to the invoker. Frames use an *array of local variables* and an *operand stack* to pass parameters and store intermediate results.

When a method is invoked, the parameters being passed are stored in its array of local variables. The array's length is determined at compile time. The array takes two consecutive local variables to hold the value of a `long` or `double` type, other stack types (`int`, `float`, and `reference`) take up only one local variable.[2] Each entry on the local variable array is accessed by indexing. The initial index of the parameters varies depending on the type of the method. If the method is a class method, its parameters begin at the first element of the local variable array, that is, at index zero. Instance methods, have the first element of their local variable arrays reserved for the `this` pseudovariable, thus their parameters begin at index one [5].

All computations performed in the context of methods take place in the operand stack. The operand stack is empty upon frame creation, and values are pushed onto or popped from the stack during execution [17]. Similarly to the local variable array, the maximum depth of the operand stack is determined at compile time and long and double types take up two operand stack slots.

V. Maxine VM

Maxine VM [21] is an open-source JVM from Oracle Laboratories (formerly Sun Microsystems Laboratories). It is a *meta-circular* HLL VM (written in the same language that it realizes). Maxine VM also integrates with Oracle's standard Java Development Kit (JDK) packages, and relies upon an optimizing compiler that translates the HLL VM itself.

The optimizing compiler generates a boot image of Maxine VM in an "ahead-of-time" fashion [21]. The boot image is not a native executable but a binary blob targeted at the platform for which the image was generated. The boot image contains data required by the HLL VM until it can begin loading further classes and compiling methods on its own (for example, a pre-populated heap and a binary image of method compilations). However, since the boot image is not executable by itself, bootstrapping Maxine VM entails invoking a C startup routine that loads the boot image and then transfers control of the execution to the HLL VM.

Maxine VM's design was tailored to support HLL VM research. Its modular design allows for replacing entities with different implementations, making it a flexible testbed for trying out new ideas and prototyping HLL VM technology. Maxine VM is also a compelling platform because of its high performance. Maxine VM does not use an interpreter; instead it uses a JIT-only strategy. In effect, a tiered compilation strategy takes place during run time. Methods are compiled with a lightweight non-optimizing JIT compiler, which in a single forward pass translates Java bytecodes into pre-assembled native instructions. Afterwards, the JIT optimizing compiler may be triggered to further optimize methods that are frequently invoked (*hotspots*) [3].

Another advantage of Maxine VM is that it has a debugger and object browser, called Inspector [25]. This tool is integrated with Maxine VM, allowing the perusal of runtime information at multiple levels of abstraction. For example, objects can be inspected either abstractly or in terms of the platform-dependent memory layout. Likewise, methods can be viewed as source code, bytecodes, or native instructions.

VI. Proof-of-concept Implementation

We extended Maxine VM to demonstrate the feasibility of developing a mutation-aware JVM. Our proof-of-concept implementation automates three key steps of mutation: execution of the original program, mutant execution, and mutant results comparison. This tool implements weak mutation testing and emphasizes individual methods. Java methods are the fundamental unit for encapsulating behavior and most JVMs are built to optimize their execution. This paper calls methods in the program under test *originals*, and methods that have been modified for mutation *mutants*.

When the implementation reaches an original method invocation it prepares to execute all mutant methods of the original method. The program state is first saved, then the original method is executed. Then, each mutant method is called as a separate thread with a copy of the program state. The main program thread is held until all mutants are run. After finishing, the results of the mutant method are compared with the results of the original method. This research focuses on unit testing, so the results of method execution are defined to be the return value, which return location was used, instance

[2]The types `boolean`, `byte`, `short`, and `char` are treated as `int`.

variables of the class that the method referenced, and other local variables in the method. Details about saving the state are given in the following subsection.

When an original method is first invoked, it is pre-processed before being compiled and run by the JIT compiler. The pre-processing instruments the method to obtain a snapshot of the context before and after invocation. The instrumentation also transfers control to the entity responsible for triggering the execution, handling unexpected behavior, and analyzing mutant methods. Original methods are instrumented at the beginning and at each of their return locations. Figure 1 outlines the pre-processing that takes place before an original method is JIT-compiled: the instrumented sections and code that is invoked from within the inserted instrumented code are shown in gray shade. After the pre-processing, the modified original method is JIT-compiled and executed.

As can be seen in Figure 1, in the presence of an already instrumented original method, the program's *calling sequence* (code executed immediately before and after a method call) is modified to invoke all mutant versions of the original method. Mutant methods are only instrumented at return locations, as shown in Figure 1, to capture the result of running the mutant.

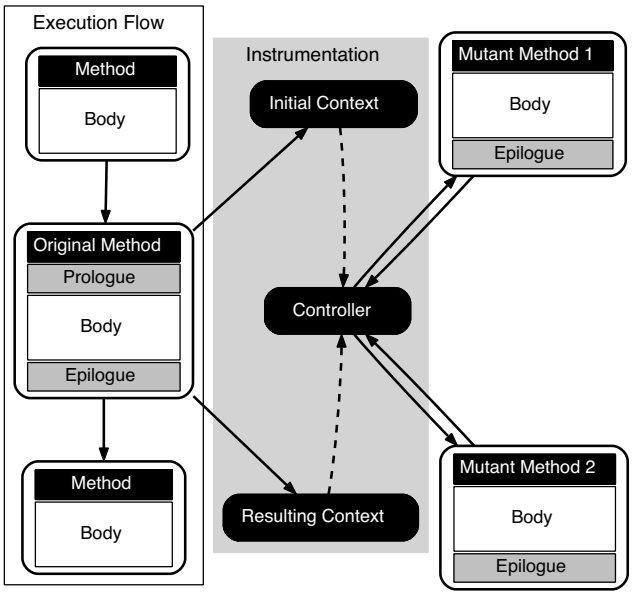


Fig. 1: Overview of the pre-processing through which original methods undergo before being JIT-compiled.

Mutant methods are run in separate threads and results are compared with results from the original method immediately. If the mutant method has results different from the original method, the mutant is marked dead. The program under test resumes execution after all the mutant methods finish. Comparing results at the end of the methods makes this a weak mutation approach.

A benefit of forking new threads is that the program up until a method call does not need to be repeated for every mutant. This implements the split-stream execution in King and Offutt's paper [15]. This is a significant execution savings. As with all other mutation systems, dead mutants are not re-executed. Also, the mutant method look-up is performed only once for each method. The look-up takes place just after the first execution of the original method. After this, the controller keeps track of the mutant methods, invoking the ones that are still alive at the end of executions of their respective original methods.

A. Prologues

As described in the preceding section, original methods undergo pre-processing before being invoked for the first time. The pre-processing instruments the methods' *prologues*, which is located before their bodies. The instrumentation extracts a copy of the *initial context* as illustrated in Figure 1. What specifically is saved depends on the method. A class method's context is just the parameters passed to it, if any. An instance method always has at least one parameter since the index 0 is reserved for the `this` pseudo variable. So the contexts of instance methods contain the receiver and all parameters.

Algorithm 1 gives a high-level overview of the instrumentation code for copying the initial context. Line 2 gathers the initial context into an array with a conservative approach: instead of computing the minimal depth required for the method's operand stack, we increase its size by 5. Additionally, an internal representation for the original method, a string, is generated and added to the runtime constant pool as shown in line 3. Next, if the method is an instance one, the underlying object is deep copied into the initial context array (lines 5–9).

Next, an array is initialized with information on the method's parameters (line 10). Since each Java primitive type has specific instructions that operate on them, parameter information is used to decide which load opcode needs to be instrumented to copy each parameter in the context array. Although not shown in Algorithm 1, primitive types are converted to object wrapper classes before being added to the initial context array. In addition, type information is used to find the index of the next variable to be included in the initial context array. This is needed because, as mentioned in Section IV, some primitive types take up two slots, and attempts to load double types using inappropriate opcodes would lead to type checking problems. Reference types are deep copied as shown in line 15 before being added to the array.

Finally, in line 29, after collecting the initial context into an array, the instrumentation code transfers both the method's internal representation and the context array to the controller. The controller uses copies of the information to invoke each mutant method. Thus, mutant methods are invoked with the same context that was passed to the original method.

It is worth mentioning that copying only a method's parameters sometimes does not encompass all the context needed to executed mutant methods. However, by not copying a large portion of the heap for every mutated method, our approach decreases the amount of storage space required to implement weak mutation.

B. Epilogues

Inserting epilogue instrumentation code is more complex. It entails obtaining information on both the types of variables and their liveness. Each method has only one prologue, but can have multiple epilogues; one at each return site. For example, the method shown in Listing 1 has three return locations (i.e., ❶, ❷, and ❸) so three epilogues are needed. Although `void` methods might not have explicit `return` statements, when they are rendered into bytecodes, opcodes of the family *xreturn* are inserted at their return locations. We use this to implement epilogues.

Algorithm 1 Prologue instrumentation.

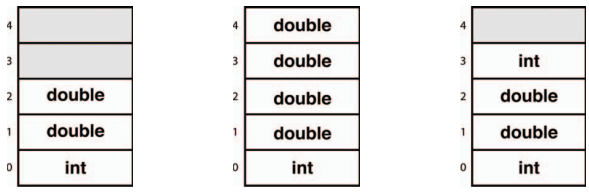
```
1:  procedure PROLOGUE(method)
2:      method.opStack.size ← method.opStack.size + 5
3:      add method.ID to runtimeConstantPool
4:      i ← 0
5:      if method.isInstance? then
6:          load deep copy of method.receiver
7:          add copy to context[i]
8:          i ← i + 1
9:      end if
10:     parameters ← method.parameters
11:     if parameters.length ≠ 0 then
12:         for each p in parameters do
13:             if parameters[i] is category 1 then
14:                 if parameters[i] is reference then
15:                     load deep copy of parameters[i]
16:                     add copy to context[i]
17:                 else
18:                     load parameters[i]
19:                     add value to context[i]
20:                 end if
21:                 i ← i + 1
22:             else
23:                 load parameters[i], parameters[i + 1]
24:                 add value to context[i], context[i + 1]
25:                 i ← i + 2
26:             end if
27:         end for
28:     end if
29:     transferToController (method.ID, context)
30: end procedure
```

Our implementation discovers return locations by overriding all *xreturn* methods in Maxine VM bytecode pre-processor. These methods are invoked whenever a return opcode is found in the bytecode. Another technical hurdle was that while it is possible to know how large the local variable array is at a return location, knowing which variables are live and what their types are required further analysis[3]. We coped with this issue by modifying Maxine VM bytecode verifier to capture the state at return locations.

To illustrate the dataflow analysis used to uncover which variables make up the resulting context, consider the example method shown in Listing 1. Before line 3 is executed, the set of local variables comprises just the method's parameters as shown in Figure 2(a). If the condition in line 3 evaluates to true, the statements in lines 4 and 5 are executed. During their execution, a variable of type double is created and stored in the local variables set, using two slots as depicted in Figure 2(b). Therefore, this variable is *live* at the return location ❶. If the condition in line 3 evaluates to false and the one in line 6 evaluates to true, an integer is created and stored in the local variables set. As a result, the array of local variable would be as depicted in Figure 2(c) at return location ❷. Finally, if neither of the previous conditions is true, the else in line 10 (return location ❸) is executed and none of the previously mentioned variables are created. In such case, the set of local variables would be as depicted in Figure 2(a); the context has only two variables, `seasonCode` and `amount`. If `charge` were an instance method rather than static, the resulting context would also contain a reference to the instance upon which the method operates.

Listing 1: Example method with multiple return locations.

```
1  public static double charge(int seasonCode,
2                              double amount) {
3    if (seasonCode == SUMMER) {
4      double summerRate = getSummerRate()-10.9;
5      return amount + summerRate; ❶
6    } else if (seasonCode == NEW_YEARS_DAY) {
7      int multiplyBy = 2;
8      return amount * multiplyBy; ❷
9    } else {
10     return amount;              ❸
11   }
12 }
```



(a) (b) (c)

Fig. 2: The three possible layouts of the array of local variables at each return location. Initially, only three slots are taken up by the method's parameters as depicted in (a). Afterwards, depending on which condition evaluates to true, the array of local variables may be either completely (b) or partially taken up as shown in (c) and (a).

An additional ability is that when a mutant method makes an abnormal return by throwing an exception that is not handled in its body nor indicated in its signature, the exception is

[3]Java files compiled with the `-g` option have debug data that could have been used to determine the liveness and type of variables. For flexibility, we decided not to impose such a constraint.

handled by the controller (shown in Figure 1). Since in this case the exception is thrown by a method inside a different thread, the thread running the controller has no access to the stack associated with the terminated thread. Therefore, our implementation performs no *"postmortem examination"* of the contents of the terminated thread and the resulting context is the exception in question.

VII. Experiment Setup

This section describes the experiment setup used to gauge the performance of the VM-integrated mutation testing system. Specifically, we investigated the following research question:

RQ_1: How much of the computational cost of mutation testing can be reduced by implementing it as an integral part of a managed execution environment and weakly killing mutants?

To evaluate this question, we compared our implementation with a conventional mutation testing tool for Java, muJava [18]. A difference is that muJava implements strong mutation testing, but it is an easy to use Java mutation tool that has been widely used [10].

A. Goal Definition

We use the organization proposed by the Goal/Question/-Metric (GQM) paradigm [24]. GQM presents experimental goals in five parts: object of study, purpose, perspective, quality focus, and context.

- **Object of study:** The objects of study are our VM-integrated mutation testing implementation and muJava.
- **Purpose:** the purpose of this experiment is to evaluate two approaches to automating mutation testing with respect to their computational costs. Specifically, we investigate whether VM-integrated mutation testing is a significant improvement over the conventional approach (muJava). The experiment provides insight into how much speedup can be obtained by implementing mutation analysis within the managed runtime environment. It is also expected that the experimental results can be used to evaluate the impact of forking the execution of mutant methods in their own threads and weakly killing them have on the performance.
- **Perspective:** this experiment is run from the standpoint of a researcher.
- **Quality focus:** the primary effect under investigation is the computational cost as measured by execution time. The execution time of a program under test is defined as the time spent by the mutation testing system executing the program and all of its mutants against a test case.
- **Context:** this experiment was carried out using Maxine VM on a 2.1GHz Intel Core 2 Duo with 4GB of physical memory running Mac OS X 10.6.6. To reduce potential confounding variables, muJava was run within our modified version of Maxine VM. Since the current implementation is an academic prototype and the subject programs are not of industrial significance, this experiment is intended to give evidence of the efficiency and applicability of the technique solely in academic settings.

Our experiment can be summarized using Wohlin et al.'s template [24] as follows:

Analyze *our VM-integrated implementation and muJava*
for the purpose of *comparing*
with respect to their *computational cost*
from the point of view of *the researcher*
in the context of *heterogeneous subject programs ranging from 11 to 103 lines of code.*

B. Hypothesis Formulation

Our research question ($\mathbf{RQ}_1$) was formalized into hypotheses so that statistical tests can be carried out:

Null hypothesis, $\mathbf{H}_0$: there is no difference in efficiency between the two implementations (measured in terms of execution time) which can be formalized as:

$$\mathbf{H}_0\text{: } \mu_{\text{muJava}} = \mu_{\text{VM-integrated implementation}}$$

Alternative hypothesis, $\mathbf{H}_1$: there is a significant difference in efficiency between the two implementations (measured in terms of execution time):

$$\mathbf{H}_1\text{: } \mu_{\text{muJava}} \neq \mu_{\text{VM-integrated implementation}}$$

C. Experiment Design

To verify our conjecture, we applied a standard design with one factor and two treatments [24]. The main factor of the underlying experiment, an independent variable, is mutation testing. The treatments or levels of this factor are the two approaches to automating mutation testing: muJava and our VM-integrated implementation. In this experiment setup, the main dependent variable is execution time, which is defined as the time spent by a treatment to run a program and all of its mutants against a test case.

We used six subject Java programs ranging from 11 to 103 lines of code. During the selection of these programs we focused on covering a broad class of applications, rather than placing too much emphasis on their complexity and size. Also, several of the subject programs have been studied elsewhere, making this study comparable with earlier studies. For example, `Triangle` implements the triangle classification algorithm, which has been broadly used as an example in the software testing literature [2]. Table I gives the number of executable Java statements and the number of mutants generated from each subject program.

TABLE I: Subject programs used in the experiment.

Subject Program	Experimental Subject Programs	
	Lines of Code*	Number of Mutants
`Fibonacci`	11	49
`ArrayMean`	13	38
`InsertionSort`	14	63
`ArrayIterator`	35	46
`KMP`	53	140
`Triangle`	103	316

*Physical lines of code (non-comment and non-blank lines).

All mutants were generated using muJava, which implements class-level and method-level (i.e., traditional) mutation operators. Because our interest is on testing methods, and the class-level mutation operators focus on the integration of classes, this experiment only used the method-level operators. The method-level operators are *selective* [20], that is, only the mutation operators that are most effective. The mutants were run on both treatments, which required minor modifications since muJava generates new .class (bytecode) and Java files for each mutant, whereas our implementation requires that all faulty versions of a method be placed in the same file as their original method. To adapt muJava mutants to the VM-integrated implementation, we collected all mutants of a method into the same class file, and gave them unique names corresponding to the mutant type (i.e., operator name) and a number. For example, ROR mutants for method `mean()` were named `mean$ROR_1()`, `mean$ROR_2()`, etc. These operations were mostly performed by a Ruby script.

For each program, we randomly generated 100 test cases. These tests were then executed against the mutants under both treatments. `ArrayMean` and `InsertionSort`, for instance, were executed using tests containing arrays of randomly-varying sizes (ranging from 0 to 1000) filled with random double values. Since we were interested solely in investigating the run-time of the two execution models, we did not try to generate mutation-adequate test sets. The execution time of each test case was calculated based on the mean of three executions. To deal with mutants that go into infinite loops, we set both treatments with a three-second timeout interval.

D. Analysis of Data

This section presents the experimental findings. The analysis is divided into two subsections: (*1*) descriptive statistics and (*2*) hypothesis testing.

1) Descriptive Statistics: This subsection provides descriptive statistics of the experimental data. Figure 3 charts the mean execution times for the subjects against the test sets. From Figure 3, it can be seen that the Maxine VM implementation outperforms muJava on all subjects, and significantly so for some. The speedup is less for some subjects, and it seems that the larger differences are when the subject program uses more computation (`InsertionSort` and `KMP`) and when the tests make several method calls (for example, `Triangle`, whose test cases comprise method calls to classify the underlying triangle and compute its perimeter and area).

Since the execution times diverge so much, particularly with muJava, we consider the median to be more a useful measure of central tendency than the mean. The median execution times are in Table II, which shows a maximum difference of 89% for `InsertionSort`.

The data dispersion is shown in Table III, which shows the standard deviations. These data show that muJava had larger standard deviation values than Maxine VM. Presumably, the VM-integrated implementation is more consistent because it requires only one execution per test case to evaluate all mutants (the forking, or split-stream technique). muJava, on the other hand, separately executes the original program and each live mutant for each test. That is, in addition to the overhead of repeatedly loading new versions of the program, muJava also has to execute chunks of code that have already been executed until it reaches mutation points. Another advantage of the VM-integrated implementation is that its performance is not adversely affected when most of the mutants are killed due to timeout: each mutant executes in its own thread, thus mutants that end up in an endless loop do not affect the execution of other mutants. They are eventually preempted and killed by the controller (Figure 1). This feature resulted in significant savings for `InsertionSort`.

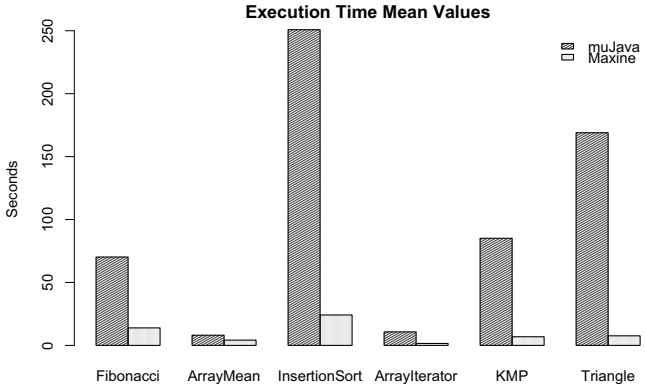


Fig. 3: Execution time mean values for each subject program under each of the two treatments.

TABLE II: Median of the execution times for each subject program under both treatments.

MEDIAN		
SUBJECT PROGRAM	TREATMENT	
	muJava	Maxine
`Fibonacci`	63.49s	13.77s
`ArrayMean`	7.48s	4.16s
`InsertionSort`	225.90s	23.97s
`ArrayIterator`	10.94s	1.56s
`KMP`	58.85s	6.77s
`Triangle`	162.90s	7.35s

TABLE III: Standard deviations of the execution time for each subject program under each of the two treatments.

STANDARD DEVIATION		
SUBJECT PROGRAM	TREATMENT	
	muJava	Maxine
`Fibonacci`	34.26s	0.50s
`ArrayMean`	3.36s	0.24s
`InsertionSort`	124.67s	0.48s
`ArrayIterator`	1.66s	0.31s
`KMP`	62.82s	0.42s
`Triangle`	105.62s	1.29s

Figure 4 summarizes the sampled run-time data, providing an overview of the significant savings in execution that can be achieved by our VM-integrated implementation.

2) Hypothesis Testing: Since some statistical tests only apply if the population follows a normal distribution, before

choosing a statistical test we examined whether our sample data departs from linearity. We used Q-Q plots as shown in Figure 5, which show that most sets of data depart from linearity, indicating the non-normality of the samples. These plots also show the presence of outliers. Thus, we used a non-parametric test, the Wilcoxon signed-rank test [12].

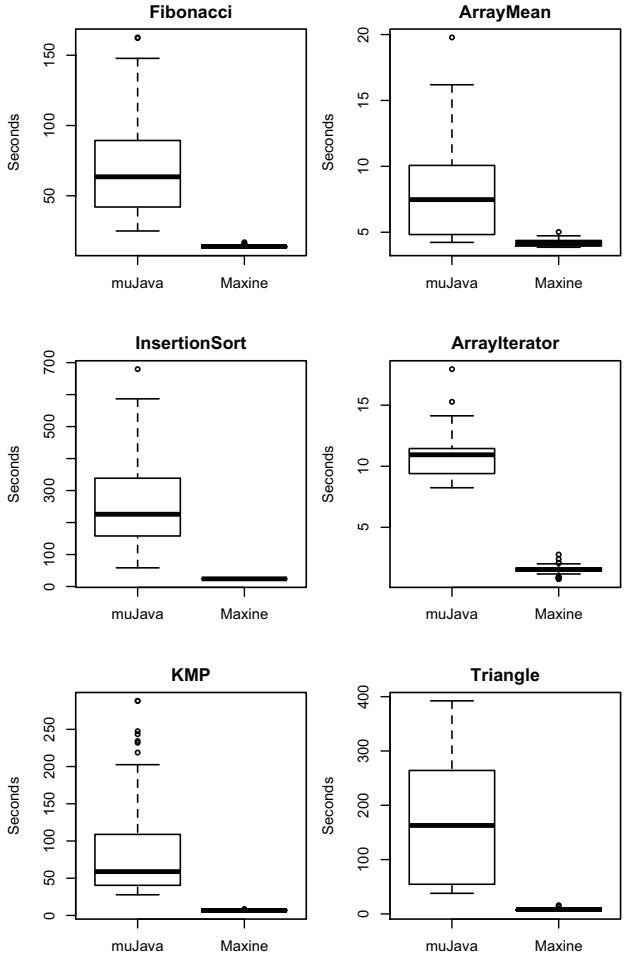


Fig. 4: Boxplots of the execution times for each subject program under each treatment.

Table IV shows the results of hypothesis testing with a significance level of 1%. Based on these data, we conclude there is considerable difference between the means of the two treatments. We were able to reject $\mathbf{H}_0$ at 1% significance level in all cases. All the p-values are very close to zero, as shown in Table IV, which further emphasizes that the VM-integrated implementation performs significantly better than muJava.

From observing the confidence intervals shown in Table IV it can be seen that the VM-integrated implementation led to substantial savings in execution time in most cases. For instance, savings of approximately 187.53 seconds for `InsertionSort` and 134.14 seconds for `Triangle` were achieved. The worst-case was `ArrayMean`, for which speedups of only about 2.74 seconds were achieved.

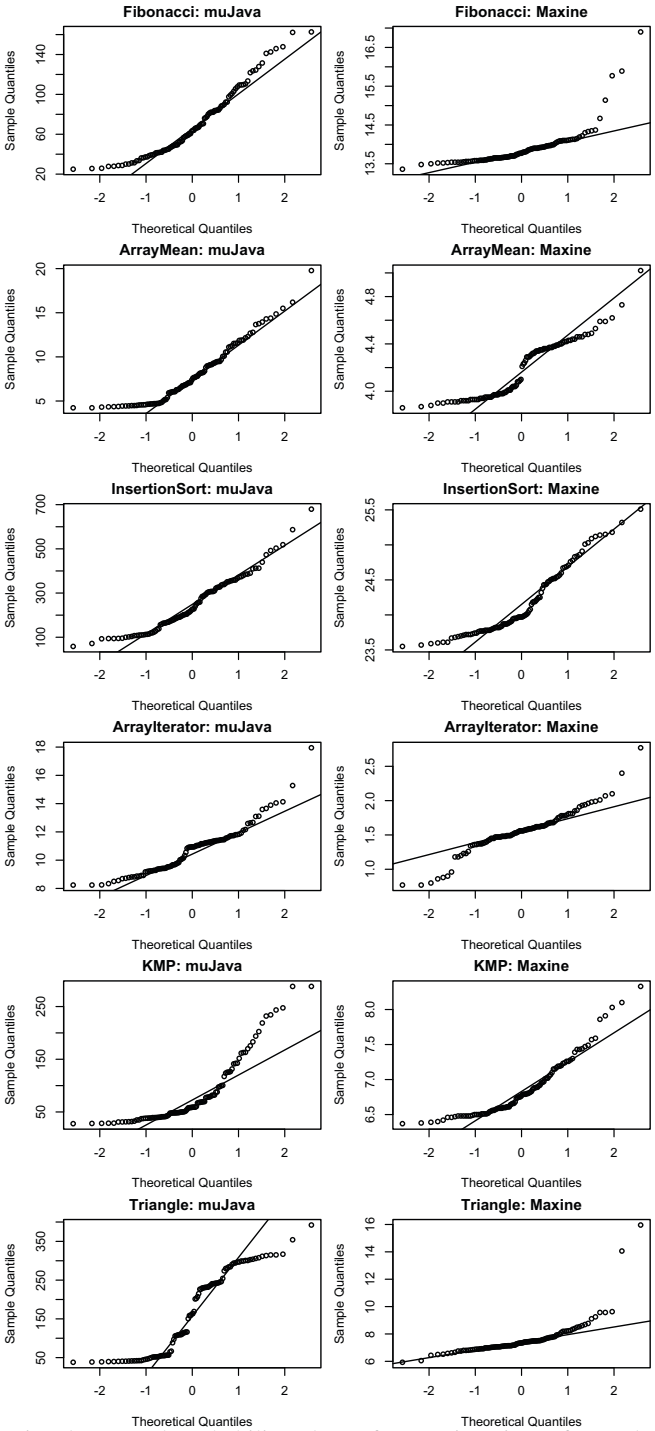


Fig. 5: Normal probability plots of execution times for each subject program under each treatment.

E. Threats to Validity

The lack of representativeness of the subject programs may pose a threat to external validity. Unfortunately, this is a problem with virtually all software engineering research, since

TABLE IV: Testing of hypotheses for each subject program.

HYPOTHESIS TESTING						
SUBJECT PROGRAM	T^+	T^-	V	P-VALUE	CONFIDENCE INTERVAL	COMMENT
`Fibonacci`	5050	0	5050	$<$ 2.2e-16*	44.13 to 63.53	We can reject $\mathbf{H}_0$ at the 1% significance level.
`ArrayMean`	5036	14	5036	$<$ 2.2e-16*	2.74 to 4.62	We can reject $\mathbf{H}_0$ at the 1% significance level.
`InsertionSort`	5050	0	5050	$<$ 2.2e-16*	187.53 to 252.69	We can reject $\mathbf{H}_0$ at the 1% significance level.
`ArrayIterator`	5050	0	5050	$<$ 2.2e-16*	8.67 to 9.50	We can reject $\mathbf{H}_0$ at the 1% significance level.
`KMP`	5050	0	5050	$<$ 2.2e-16*	50.70 to 87.51	We can reject $\mathbf{H}_0$ at the 1% significance level.
`Triangle`	5050	0	5050	$<$ 2.2e-16*	134.14 to 186.95	We can reject $\mathbf{H}_0$ at the 1% significance level.
*2.2e-16 means $2.2X10-16$ = 0.00000000000000022. This is the smallest non-zero number R can handle [23]. Hence, the p-values shown above are not very accurate, however, they are definitely small. Thus, we can confidently reject $\mathbf{H}_0$ in all cases.						

we have theory to tell us how to form a representative sample of software. Apart from not being of industrial significance, another potential threat to the external validity is that the investigated programs do not differ considerably in size and complexity. To partially ameliorate that potential threat, the subjects were chosen to cover a broad class of applications. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings. The fact that the VM-integrated approach achieved more speedup with programs that had more computation and more method calls indicates any bias could be against the VM-integrated approach. That is, it may perform even better in industrial practice. Furthermore, since our study focuses on evaluating strategies mostly used for unit testing, the size of the subject programs should not invalidate the results.

A threat to construct validity stems from possible faults in both implementations. With regard to our VM-integrated implementation, we mitigated this threat by running a carefully designed test set against several small example programs. A factor that hindered the use of such test set to perform corrective regression testing was the slow turnaround time: changes to Maxine VM files required at least a five-minute wait for recompilation. muJava, has been extensively used within academic circles, so we conjecture that this threat can be ruled out.

VIII. RELATED WORK

The goal of this research is similar to other efforts that attempt to reduce the computational cost of mutation analysis. Our approach differs from others because it embeds mutation analysis into the VM execution engine and makes mutants first-class entities. Other studies have extended HLL VMs to reduce the overhead of other testing activities: *(i)* collecting coverage data and *(ii)* storing compiled code during test suite execution.

To address the former issue, Chilakamarri and Elbaum [4] proposed to remove coverage probes after they have been executed. They implemented two strategies, collectively termed disposable coverage instrumentation, in the Kaffe JVM [13]. The first strategy removes coverage probes after execution. The second strategy improve the first one by incorporating coverage data gathered from multiple instances of the modified JVM running the application under test, thereby avoiding the execution of probes that have already been covered by any of the concurrently running instances.

The second issue (storing seldom used native code during test suite execution) was approached by Kapfhammer et al. [14]. They modified the Jikes Research Virtual Machine (RVM) [1] to adaptively ascertain and unload rarely used native code from the heap.

IX. CONCLUDING REMARKS

This research capitalizes on features already present in a managed execution environment and its underlying virtual instruction set architecture to boost the performance of weak mutation analysis. Although other researchers have retrofitted software testing features into interpretive execution environments and a plethora of instrumenting systems have been implemented, to the best of our knowledge this is the first effort to incorporate the ideas of mutation testing into a full-fledged, JIT-enabled HLL VM. This paper described the basic design and components of the proof-of-concept implementation, which builds on a contemporary JVM implementation. According to our experiment results, it can be concluded that the VM-integrated approach to weak mutation outperforms a conventional strong mutation testing tool by a large margin in terms of execution time.

As far as we know, our implementation is the first to exploit multithreading to boost weak mutation performance. This novel characteristic of our implementation yields major execution savings by concurrently executing mutant methods. It proved to be particularly useful for speeding up the execution of mutant sets whose most mutants end up in infinite loops. Since our implementation executes each mutant method in its own thread, the mutants that become stuck in a loop do not affect the execution of others. Future work stemming from this first experiment needs to examine how much of the achieved speed-up is due to multithreading and how much can be attributed to weak mutation. Furthermore, the subject programs we used in the experimental study are not very representative: the largest subject program contains 103 lines of code. Thus, we intend to use more complex subject programs in a follow-up experiment so as we can obtain more conclusive results and show evidence of the scalability of our implementation.

Maxine VM compiles all methods prior to execution, thus executing a program containing a large number of mutant methods would require a considerable amount of memory. Consequently, an improvement to the current implementation

would allow the memory allocated to compiled mutant methods to be reclaimed by the garbage collector as soon as they are killed. Further experiments need to be conducted to evaluate the effects that a potentially large number of mutants would pose on the execution time.

As long term future work, we will integrate several other software testing features into our implementation so that we can evaluate which benefits the most from being incorporated into an execution environment. Besides opening up possibilities for further speeding up computationally expensive software testing techniques, we intend to delve deeper into understanding the interplay between the HLL VMs and their running programs.

ACKNOWLEDGMENTS

The authors would like to thank the financial support provided by FAPESP (grant number 2009/00632-1), CAPES (grant number 0340-11-1), and CNPq (grant number 559915/2010-1). We are also thankful to all core members of the Maxine VM project at Oracle Laboratories, specially Douglas Simon for his always enlightening and accurate answers on the Maxine VM mailing list. Simone Borges also should be thanked for helping one of the authors with some of the figures.

REFERENCES

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Systems Journal*, 44(2):399–417, 2005.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based Compilation in Execution Environments Without Interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pp. 59–68. ACM, 2010.

[4] K.-R. Chilakamarri and S. Elbaum. Reducing Coverage Collection Overhead with Disposable Instrumentation. In *15th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 233–244, November 2004.

[5] I. D. Craig. *Virtual Machines*. Springer, 2005.

[6] R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11:34–41, April 1978.

[7] V. H. S. Durelli, K. R. Felizardo, and M. E. Delamaro. Systematic Mapping Study on High-level Language Virtual Machines (VMIL). In *Virtual Machines and Intermediate Languages*, pp. 1–6. ACM, 2010.

[8] A. R. Gregersen, D. Simon, and B. N. Jørgensen. Towards a Dynamic-update-enabled JVM. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, pp. 2:1–2:7, New York, NY, USA, 2009. ACM.

[9] W. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[10] J. Hu, N. Li, and J. Offutt. An analysis of OO mutation operators. In *Seventh Workshop on Mutation Analysis (IEEE Mutation 2011)*, pp. 334–341, Berlin, Germany, March 2011.

[11] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, September 2011.

[12] R. A. Johnson and G. K. Bhattacharyya. *Statistics: Principles and Methods*. Wiley, 2009.

[13] Kaffe. Kaffe Virtual Machine. *http://www.kaffe.org/*.

[14] G. M. Kapfhammer, M. L. Soffa, and D. Mosse. Testing in Resource Constrained Execution Environments. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 418–422. ACM, 2005.

[15] K. N. King and J. Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.

[16] A. H. Lee and J. L. Zachary. Reflections on Metaprogramming. *IEEE Transactions on Software Engineering*, 21:883–893, 1995.

[17] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java™ Virtual Machine Specification Java SE 7 Edition. Available at: *http://docs.oracle.com/javase/specs/-jvms/se7/jvms7.pdf*, 2012. Accessed March 3, 2012.

[18] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava : An automated class mutation system. *Software Testing, Verification, and Reliability, Wiley*, 15(2):97–133, June 2005.

[19] J. Offutt and S. D. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[20] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[21] Oracle Corporation. Maxine Virtual Machine Project. Available at: *http://wikis.sun.com/display/MaxineVM/-Home*, 2010.

[22] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[23] P. Teetor. *R Cookbook*. O'Reilly, 2011.

[24] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsoon, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 1999.

[25] T. Würthinger, M. L. Van De Vanter, and D. Simon. Multi-level Virtual Machine Debugging Using the Java Platform Debugger Architecture. In *Perspectives of Systems Informatics*, pp. 401–412. Springer, 2010.