An Experimental Study of Methods for Executing Test Suites in Memory Constrained Environments

Suvarshi Bhadra
Milcord LLC
sbhadra@milcord.com

Alexander Conrad, Charles Hurkes,
Brian Kirklin, Gregory M. Kapfhammer
Allegheny College
gkapfham@allegheny.edu

Abstract

Software for memory constrained mobile devices is often implemented in the Java programming language because the Java compiler and virtual machine (JVM) provide enhanced safety, portability, and the potential for run-time optimization. However, testing time may increase substantially when memory is limited and the JVM employs a compiler to create native code bodies. This paper furnishes an empirical study that identifies the fundamental trade-offs associated with a method that uses adaptive native code unloading to perform memory constrained testing. The experimental results demonstrate that code unloading can reduce testing time by 17% *and the code size of the test suite and application under test by* 68% *while maintaining the overall size of the JVM. We also find that the goal of reducing the space overhead of an automated testing technique is often at odds with the objective of decreasing the time required to test. Additional experiments reveal that using a complete record of test suite behavior, in contrast to a sample-based profile, does not enable the code unloader to make decisions that markedly reduce testing time. Finally, we identify test suite and application behaviors that may limit the effectiveness of our method for memory constrained test execution and we suggest ways to mitigate these challenges.*

1 Introduction

The Java compiler and virtual machine provide enhanced safety, portability, and the opportunity to perform run-time optimization. Therefore, the Java programming language is now a popular choice for implementing the software applications that execute on resource constrained mobile and embedded devices [11, 14, 23]. In fact, Java is currently being used in resource constrained embedded environments to implement ad hoc and sensor networks [16], robots [1], XML processors [2], HTTP servers [3] and numeric expression evaluation and function graphing applications [4]. However, using Java in mobile and embedded devices is challenging because of the memory constraints inherent in these execution environments. In an attempt to balance the competing requirements of ample memory and battery lifetime, engineers often restrict the size of physical memory since it is often the "dominant energy consumer" [20]. For instance, the recently developed SmartControl CS-210/211 single-board computers by Snijder Micro Systems only contain 64 MB of RAM [2] while the T-Mobile G1 with Google Android has 192 MB of RAM [5].

Even in light of these challenges, it is still important to test a program in the setting in which it will run since handhelds and cell phones operate in a variety of complex execution environments [7, 13]. Yet, the automated testing of Java programs on mobile devices is challenging because embedded Java virtual machines (JVMs) commonly limit the size of the heap [11] and use a "just in time" (JIT) compiler to transform Java bytecode into time efficient native code [2, 10, 19]. Unless additional steps are taken, small JVM heap sizes may prevent the testing of a program from either starting or running to completion [9]. While JVMs that use a JIT can reduce the execution time of programs by avoiding bytecode interpretation and exploiting the potential for run-time optimization, dynamic compilation also increases space overhead because the native code representation is larger than the corresponding bytecodes [23]. Furthermore, if the JVM's heap cannot continuously store a significant part of the native code and data associated with the test executor, test cases, and program under test, then freeing memory with frequent garbage collector (GC) invocations will markedly increase the time overhead of testing.

Methods for executing tests in memory constrained environments are necessary since even mobile devices with sizable memories place limitations on the heap space (e.g., the T-Mobile G1 limits a program and a test suite to no more than 16 MB of JVM heap space [11]). In fact, our preliminary empirical study revealed that the testing of a Java program on a JVM with severely constrained heap resources caused testing time to increase by 600% when compared to a configuration with adequate heap resources [13]. Yet, if the execution of all or a portion of the tests is omitted in an

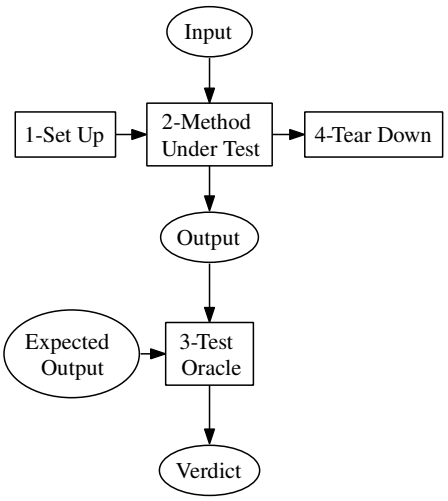


Figure 1. Executing a Test Case.

attempt to reduce the time required to test, then the quality of the program could be compromised. Alternatively, if the test suite is not executed in the intended execution setting, then the tests are less likely to reveal defects related to the program's interaction with the embedded environment. It may also be prohibitively expensive to avoid excessive heap consumption by repeatedly starting the JVM, running the next test case, and then shutting down the JVM.

In light of theses concerns, our previous work described a testing technique that uses adaptive code unloading to monitor the behavior of the test automation tool and unload certain native code bodies [13]. Building upon methods designed by Zhang and Krintz [23], our testing approach monitors the execution of a Java program and its test suite and produces either a sample-based or exhaustive profile of program behavior. Using this behavior model, the JVM identifies which native code bodies are used the least during the current execution of the test suite and are thus unlikely to be needed during the remainder of testing. Finally, the code unloader can remove these infrequently used native code bodies from the JVM's heap, thereby making room for more data, reducing the amount of time spent performing garbage collection tasks intended to free memory, and subsequently decreasing the time overhead of automated testing.

This paper is distinguished from our prior work because it extends the preliminary empirical study with new visualizations and analyses that enhance the understanding of the trade-offs associated with memory constrained testing. Using six case study applications, we measure the time overhead of test suite execution and the size of both the JVM process and the native code bodies. After reporting on baseline experiments that justify the need for methods to support testing in memory constrained environments, we compare six different test execution techniques that use code unloading. Leveraging graphs that pinpoint the variation in code

$$\{S, X\} - \{GC, TM\} = \langle C, UC, U, H \rangle$$

Parameter	Meaning
C	initial GC period (GC cycles or secs)
UC	initial unload freq (GC cycles or secs)
U	non-initial unload freq (GC cycles or secs)
H	heap residency threshold (%)

(a)

$$\{S, X\} - CS = \langle Z_{init}, Z_{incr}, U_{CS} \rangle$$

Parameter	Meaning
Z_{init}	initial code cache size (bytes)
Z_{incr}	code cache increment size (bytes)
U_{CS}	unload session resize trigger (count)

(b)

Figure 2. Code Unloading Configurations.

size and tables that summarize the general trends, this paper (i) identifies the system configurations that are best able to reduce the time and space overhead of testing and (ii) highlights the limitations of the presented approach.

2 Automated Testing with Code Unloading

In order to ensure that this paper is self contained, this section reviews the basics of automated testing with code unloading (see [13, 23] for more details). As shown by the boxes in Figure 1, the execution of each test case involves the use of "set up" and "tear down" methods that respectively run before and after the testing of the desired method. Beyond storing the native code bodies associated with both these functions and the test automation framework, the JVM must also use its heap to store various data structures (e.g., the circles representing method input and output and the expected output and verdict of each test oracle). Since the garbage collector already controls the data and the unmanaged native code bodies may consume a considerable amount of memory, our testing tool unloads methods that may not be needed during the remainder of testing.

Any memory constrained testing technique that uses code unloading must address the questions *what code must be unloaded?* and *when should code be unloaded?* [23]. As the test suite runs, the JVM's native code unloader creates either sample-based (denoted *S*) or exhaustive (denoted *X*) profiles of application behavior. Upon invocation by the JVM, the unloader consults these profiles and identifies the least frequently used native code bodies. The testing framework's JVM decides when to invoke an unloading technique by using timers (denoted *TM*) or triggers that focus on either garbage collection behavior (denoted *GC*) or the variation in code cache size (denoted *CS*) [23].

The *TM* strategy unloads code at a regular interval that is specified when program testing first begins. Alternatively, the *GC* technique performs code unloading when garbage collection occurs and the heap residency exceeds a specified threshold. The *CS* approach stores the native code bodies in a fixed size code cache and it clears and resizes this storage buffer when its space is exhausted. As in our previous work,

Name	*GC*	*CS*	*TM*
UniqueBoundedStack (UBS)	$\langle 4, 1, 1, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
Library (L)	$\langle 5, 1, 3, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
ShoppingCart (SC)	$\langle 3, 1, 1, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 2, .5, 1, 0.0\rangle$
Stack (S)	$\langle 4, 1, 1, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
JDepend (JD)	$\langle 8, 1, 4, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
IDTable (ID)	$\langle 1, 1, 3, 0.0\rangle$	$\langle 65536, 8192, 5\rangle$	$\langle 2, .5, 1, 0.0\rangle$

Figure 3. Code Unloading Configurations for the Jikes RVM.

Name	*Min* Size (MB)	# Tests	# Failures	Console Output?	NCSS
UniqueBoundedStack (UBS)	8	24	1	Yes	362
Library (L)	8	53	14	Yes	551
ShoppingCart (SC)	8	20	0	Yes	229
Stack (S)	8	58	0	Yes	624
JDepend (JD)	10	53	0	No	2124
IDTable (ID)	11	24	0	Yes	315

Figure 4. Characteristics of the Case Study Applications and Test Suites.

this paper studies the potential of six code unloading techniques (i.e., *S-GC*, *X-GC*, *S-CS*, *X-CS*, *S-TM*, and *X-TM*) to make testing more time and space efficient. For example, *S-GC* indicates that the JVM will use a sampled behavior profile to unload code according to the GC trigger.

Figure 2(a) shows that the *GC* and *TM* methods have configuration tuples where the first three parameters (C, *UC*, and U) denote GC cycles for both of the *GC* techniques and seconds for the *TM* methods. A configuration of *S-GC* or *X-GC* is described by the tuple $\langle C, UC, U, H\rangle$ where C indicates how many garbage collection cycles are assumed to occur throughout a test suite's initialization phase. During the first C cycles, code unloading will happen every *UC* cycles as long as the heap residency is above the threshold specified by H. Once C cycles have occurred, code unloading will take place every U cycles. For instance, $\langle 4, 1, 2, 0.0\rangle$ would configure *S-GC* or *X-GC* so that the first four GC cycles are assumed to occur during program startup and thus code unloading happens every cycle regardless of the residency of the JVM heap. After four cycles of garbage collection, code unloading will take place every two cycles. The *S-TM* and *X-TM* code unloading techniques could be described in an analogous fashion. For the *TM* method, the tuple $\langle 2, 1, 5, 0.2\rangle$ indicates that the first two seconds of testing are seen as part of the initialization phase during which code will be unloaded every second if the heap residency is greater than 20%. After the first two seconds, the JVM will perform code unloading every five seconds.

Figure 2(b) shows that a configuration of *S-CS* or *X-CS* is described by the tuple $\langle Z_{init}, Z_{incr}, U_{cs}\rangle$ where Z_{init} is the initial size of the code cache, Z_{incr} is the amount by which the code cache grows, and U_{cs} is the number of unloading sessions that must occur before the code cache is increased in size. As an example, the tuple $\langle 49370, 512, 5\rangle$ describes a code unloading strategy where the initial size of the code cache is $49,370$ bytes. When space in the code cache is exhausted and code unloading occurs five times, this JVM will increase the size of the entire cache by 512 bytes.

3 Experiment Goals and Design

We only report a few relevant points about the experiment design since it is similar to the one from our preliminary experiments [13]. For a program P and test suite T, we respectively measured space overhead before and after the use of code unloading, $\mathcal{S}_B(P,T)$ and $\mathcal{S}_A(P,T)$, and computed the reduction and percent reduction in the size of both the native code and JVM process, as defined in Equations (1) and (2). We also measured the time overhead reduction, $\mathcal{T}_R(P,T)$, and the percent reduction of time overhead, $\mathcal{T}_R^{\%}(P,T)$. The experiments use a Jikes Research Virtual Machine (RVM) x86 version 2.2.1 that includes the code unloading extensions from [23]. In conformance to our prior experiments, Figure 3 describes the different Jikes RVM configurations that we selected to run the tests. While we picked these tuples after performing a manual, yet systematic, parameter study it is possible that they do not represent the best configurations for the chosen applications.

$$\mathcal{S}_R(P,T) = \mathcal{S}_B(P,T) - \mathcal{S}_A(P,T) \quad (1)$$

$$\mathcal{S}_R^{\%}(P,T) = \frac{\mathcal{S}_R(P,T)}{\mathcal{S}_B(P,T)} \times 100 \quad (2)$$

As shown in Figures 4 and 5, the Jikes RVM ran in two separate memory configurations called *Min* and *Full*. The *Min* configuration was empirically identified to be the smallest heap size that would allow the program's tests to execute without producing out of memory errors and the *Full* configuration was fixed at a 32 MB maximum heap size. The *Min* RVM is meaningful because it represents the type of memory constrained environment that is common when testing occurs on an embedded device. For each case study application and JUnit 3.8.1 test suite, Figure 4 also records the number of failing tests and whether or not the test suite produces console output since these behaviors could contribute to the time overhead required to execute the tests. Finally, we used JavaNCSS 21.41 to calculate the number of non-commented source statements (NCSS) for the combined source code of the tests and the applications.

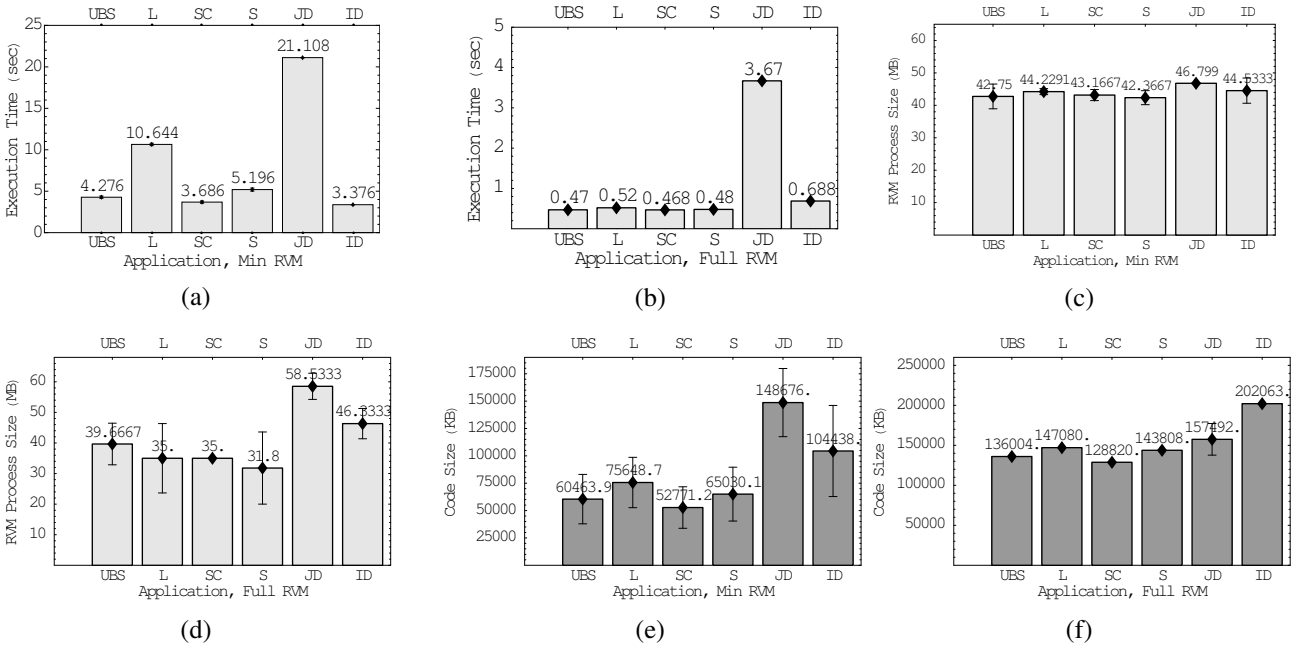


Figure 5. Baseline Measurements for: Average Execution Time - (a) *Min* and (b) *Full*; Average RVM Process Size - (c) *Min* and (d) *Full*; Average Method Code Body Size - (e) *Min* and (f) *Full*.

We executed each test suite on every application five times in order to compute arithmetic means and standard deviations. During experiments with the Jikes RVM that did not perform code unloading, the sequence of native code body sizes did not change across the five executions. For the code size measurements produced by this Jikes RVM, we report the standard deviation from the average memory consumption during a single trial. Whenever necessary, the bar charts in this paper use error bars to represent standard deviations. A diamond that appears at the top of a vertical bar without an error bar indicates that the standard deviation from the average was very small. For the results depicted in Figures 5 and 10, we adopt the convention that dark shaded bars represent a standard deviation from the average of a single trial and light shaded bars stand for a standard deviation from the average of all the trials. Finally, when one code unloading technique produces empirical results similar to another approach, we select the best strategy according to the following criteria: (i) the smallest potential time and space overhead (i.e., prefer S to X), (ii) the smallest variability (i.e., favor small standard deviations) and (iii) the smallest average native code body size.

4 Experimental Results

Baseline Performance. Figure 5 presents the baseline performance measurements for all of the evaluation metrics and a Jikes RVM that runs in both the *Min* and *Full* configurations without the benefit of code unloading. As expected, the *Full* RVM's execution time is significantly less than the *Min* RVM. This is due to the fact that the *Min* RVM must perform a considerable amount of memory management in order to operate in the resource constrained environment. For the application with the largest NCSS, the process size of the *Full* RVM is larger than the size of the RVM that executes in the *Min* configuration. As such, JD causes the *Full* RVM to consume on average 58.5 MB and the *Min* RVM to use 46.7 MB. We attribute this outcome to the fact that the *Full* configuration allows the Jikes RVM to use the 32 MB heap to store more of JD's code bodies and data structures. Yet, for the smaller NCSS programs (e.g., UBS, L, SC, S, and ID), there is less need for the RVM to take advantage of the larger heap and thus the *Min* and *Full* process sizes are relatively similar when we consider the error bars.

The bar charts in Figure 5 also reveal that the average size of the native code stored by the RVM is larger in the *Full* configuration. For example, JD exhibits an average code size of $157,492$ KB in the *Full* RVM and $148,676$ KB in the *Min* RVM. Since [23] notes that the code bodies in the Jikes RVM are stored within the GC-managed heap, the smaller maximum heap in the *Min* RVM means that less code can exist at any point during testing. In the *Min* configuration, the results indicate that across a single trial, the code size for JD varies from a minimum of $24,935$ KB to a maximum of $170,363$ KB and demonstrates a standard deviation of $31,142$ KB. Yet, when JD is executed with the *Full* RVM, the code size varies from $128,025$ KB to the same maximum of $170,363$ KB with a standard deviation of $19,775$ KB. We attribute this phenomenon to the fact

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	12.1	78.5 ✓
X-GC	11.1	61.4
S-TM	12.5	62.9
X-TM	12.3	44.9
S-CS	16.8 ✓	56.4
X-CS	11.6	52.4

(a)

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	32.7	78.8 ✓
X-GC	32.1	65.0
S-TM	32.0	72.8
X-TM	31.5	62.3
S-CS	34.3 ✓	61.4
X-CS	33.4	59.8

(b)

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	8.6	55.0 ✓
X-GC	8.5	39.2
S-TM	14.7 ✓	56.3
X-TM	8.6	30.5
S-CS	9.4	45.0
X-CS	6.3	35.2

(c)

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	24.3	79.0 ✓
X-GC	25.4	63.4
S-TM	25.0 ✓	64.9
X-TM	24.6	47.8
S-CS	24.7	61.6
X-CS	20.9	46.9

(d)

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	20.3	76.6
X-GC	21.1	60.8
S-TM	20.0	74.0
X-TM	21.5 ✓	60.9
S-CS	21.0	76.7 ✓
X-CS	20.8	73.0

(e)

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	-1.1	42.5
X-GC	-1.1	26.7
S-TM	-1.2	44.5
X-TM	-.29 ✓	28.8
S-CS	-.77	51.4
X-CS	-1.4	61.4 ✓

(f)

Figure 6. Reductions for (a) UBS, (b) L, (c) SC, (d) S, (e) JD, and (f) ID.

that the *Min* RVM slowly grows the native code space over time while the *Full* RVM's ample memory resources enable it to quickly allocate large regions of the heap to support native code storage. In summary, these baseline experiments demonstrate the need for a technique that can (i) reduce the execution time of testing when the RVM operates in a *Min* configuration, (ii) decrease or maintain the size of the Jikes RVM process, (iii) reduce the average code size during testing, and (iv) strategically load and unload native code while avoiding any undesirable fluctuations in native code size.

Time and Space Reductions. Figure 6 summarizes the time and space reduction percentages when all of the code unloading strategies and all of the case study applications are executed on the *Min* RVM (these average values are computed using the arithmetic mean from the five experiment trials). In this figure, $\mathcal{S}_R^{\%}(P,T)$ refers to the percent reduction in the size of the code bodies and a checkmark (i.e., "✓") indicates the technique that produced the greatest percent reduction of time or space overhead. These results show that the code unloading technique that produces the most noticeable space reduction does not always lead to the best time reduction. In the UBS, L, and S applications, *S-GC* yields the largest $\mathcal{S}_R^{\%}(P,T)$ value while *S-CS* or *S-TM* creates the greatest value for $\mathcal{T}_R^{\%}(P,T)$. This is due to the fact that the *S-GC* often unloads code too aggressively, thus decreasing space overhead at the cost of requiring additional time to reload previously unloaded code [23].

Figure 7 presents the variation of native code size that occurred during the first trial of test suite execution for each application. The "zig zags" in the code size variation graphs of Figure 7 show that *S-GC* causes the RVM to rapidly increase and decrease the size of the native code bodies.[1] For example, *S-GC* quickly reduces the code size of UBS from $25,214$ KB to $7,653$ KB only to require an immediate increase in code size to $32,043$ KB and ultimately force eight invocations of the unloader. In contrast, Figure 7 shows that the *S-CS* method only calls for three uses of the unloader during the testing of UBS. While this problem could potentially be resolved by increasing H, the heap residency parameter, it underscores the challenge of automated testing with code unloading. For test suites that exhibit the irregular reuse of "set up" and "tear down" methods or the unanticipated re-testing of a method, overly ambitious code unloading can limit the reductions in testing time.

Figure 8(a) further explains this trend by tracking the average number of code unloads produced by *S-CS*, *S-TM*, and *S-GC* across all experiment trials. If an inflection point in a graph from Figure 7 represents a single code unloading event, then the height of a bar in Figure 8(a) corresponds to the arithmetic mean of these events over the five runs of the test suite. For UBS, we found that *S-GC* (79%) creates a larger space reduction than *S-CS* (61%). Yet, *S-GC* does not reduce testing time for UBS any more than *S-CS* (24.3% - *S-GC* vs. 24.7% - *S-CS*) because on average it triggers a code unload more often than *S-CS* (11.4 - *S-GC* vs. 2.0 - *S-CS*). In fact, Figure 8(a) reveals that for four of the six case study applications, *S-GC* causes over two times the number of costly unloading events necessitated by *S-CS*.

The results in Figure 6 also indicate that *S-CS*, *S-TM*, and *X-TM* normally produce the most significant time reduction. However, it is important to observe that all of the techniques create very similar decreases in time overhead. This trend is demonstrated by fact that the time reductions for L range from a minimum of 31.5% (*X-TM*) to a maximum of 34.3% (*S-CS*). Figure 9 presents the average time and space percent reductions across all of the chosen case study applications, suggesting that *S-GC* most effectively reduces space overhead while *S-CS* is the best at decreasing time. These results indicate that, for the selected applications, it is beneficial (on average) to use unloading to lessen the time and space overhead of testing. Yet, unloading is unlikely to enable more programs and tests suites to concurrently run on the same device since the last column of Figure 10 reveals that RVM process size does not decrease after introducing

[1] In an attempt to create a clear visualization, Figure 7(e) only depicts the first three seconds of executing JD's test cases.

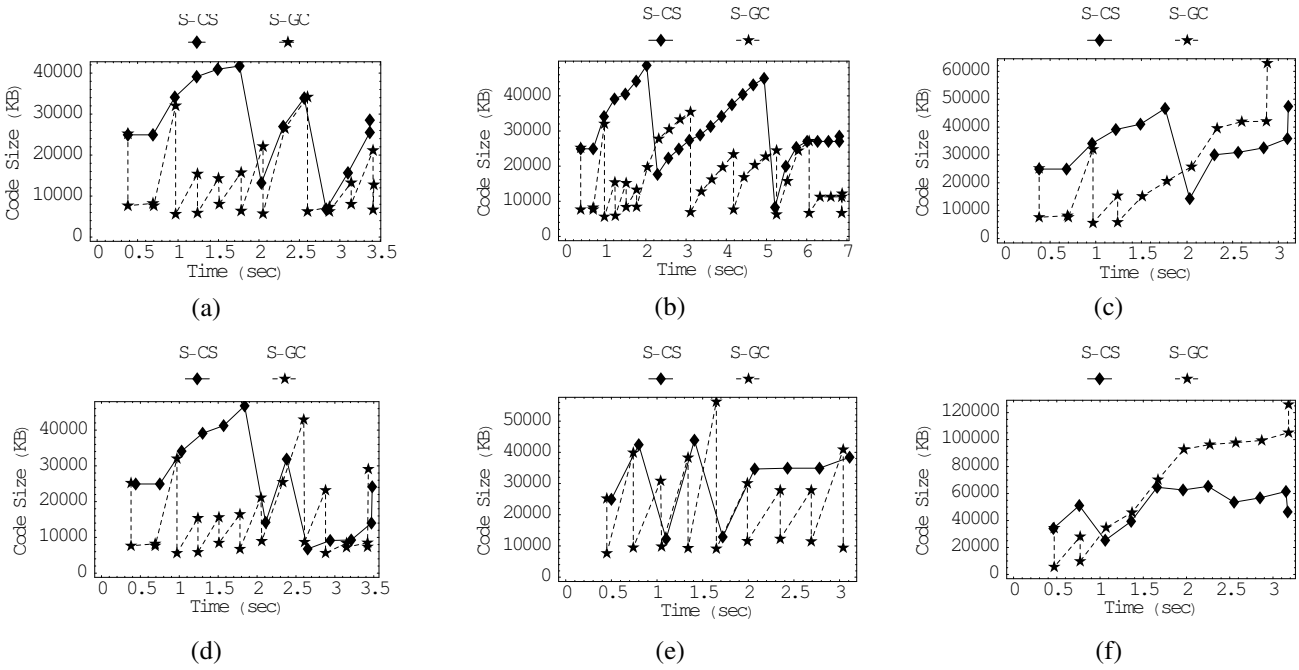


Figure 7. Code Size Variation During Testing for (a) UBS, (b) L, (c) SC, (d) S, (e) JD, and (f) ID.

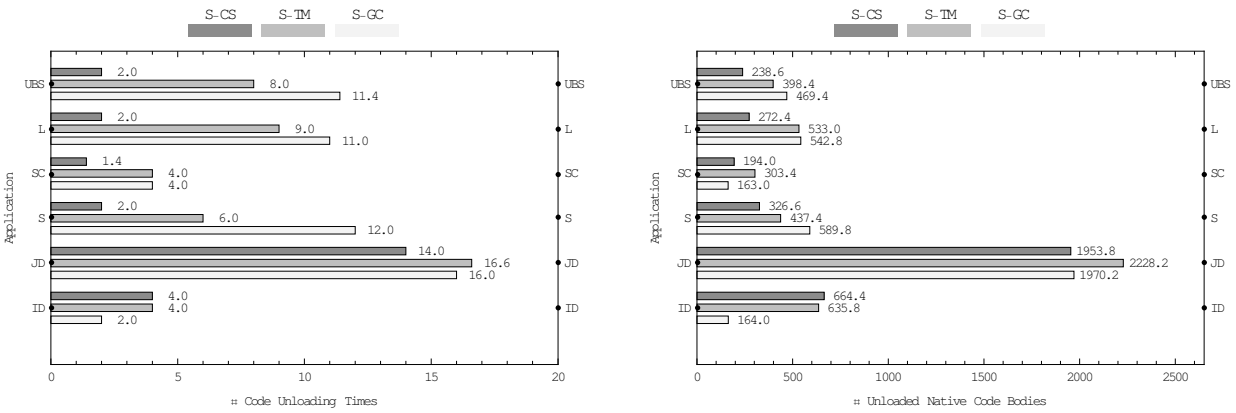
of our technique (the dashed line represents the measurement taken when the RVM did not use code unloading).

Profile Types. The results provided by Figure 10 also show that the exhaustive program behavior profile does not normally enable the reduction of time overhead noticeably more than the sampled profile. Yet, since the exhaustive profile is embedded within the code bodies [23], it frequently creates an average code size that is greater than the corresponding code size for the sample-based technique. While this greater code size does not always translate into a larger RVM process size, it does limit the potential of unloading if the addition of new tests or application features eventually increases the code size of the test suite and/or program. Interestingly, for four out of the six case study applications (UBS, L, SC, and ID) the *X-CS* technique creates a smaller Jikes RVM process size than the *S-CS* strategy. We attribute this phenomenon to the fact that the exhaustive profile ensures that the RVM does not inappropriately unload native code bodies and subsequently trigger the growth of the code cache that is stored in the RVM heap [23].

Per-Application Summary. Using Figure 10 as a reference, we observe that for UBS (row 1, Figure 10), *S-CS* produces the smallest execution time, *S-GC* creates the smallest code size, and all techniques yields similar Jikes RVM memory consumption levels. While *X-CS* creates a Jikes RVM with the smallest memory footprint for UBS, it does so with a noticeable level of variability across trials. Since *S-CS* exhibits a smaller standard deviation from the average execution time and a larger code size than *S-GC*, we judge that either *S-GC* or *S-CS* are appropriate for UBS.

For L (row 2, Figure 10), *S-CS*, *S-GC*, and *S-TM* all produce significant reductions of execution time and code size while maintaining the size of the virtual machine and creating similarly low levels of variability in the RVM process size. While *S-TM* yields the best decrease in testing time for SC, row 3 of Figure 10 reveals that all methods lead to the same modest reduction. The results for S (row 4, Figure 10) show that all techniques have similar time reductions, while both *S-CS* and *X-CS* evidence the most variability in the size of the code bodies and the RVM process. We anticipate that this outcome is due to the fact that it is possible to select better values for the Z_{init} and Z_{incr} parameters. In fact, this trend is an external confirmation of the concerns that Zhang and Krintz raise about configuring the *CS* strategy [23]. For JD (row 5, Figure 10), we see that all unloading methods reduce the execution time from 21.1 seconds to approximately 16.8 seconds while maintaining the size of the RVM and avoiding any noticeable variability across trials.

Limitations. Code unloading does not always significantly reduce the time overhead associated with the execution of a test suite in a memory constrained environment. For example, even though *S-GC* reduces SC's native code size by 55% on average, the time overhead is only reduced by 8.6%. More importantly, Figure 6 shows that the time associated with testing ID is always increased by a small factor (e.g., $\mathcal{S}_R^{\%}(P,T)$ ranges from $-.29$ to -1.4). While Figure 8(a) shows that no unloading technique causes more than 4 unloads for ID, Figure 8(b) indicates that *S-CS* and *S-TM* both require the unloading of more than 635 native code bodies. When compared with the number of unloaded code



(a) (b)

Figure 8. Average Number of (a) Code Unloading Events and (b) Unloaded Code Bodies.

bodies for the L application that has a similar number of non-commented source statements (see Figure 4), it seems likely that the RVM must be unloading the native code of an external library used by ID. In fact, ID uses the Apache log4j logging utility throughout testing and thus the logger's code bodies can only be briefly unloaded before they must be subsequently reloaded. Since the working set of ID is very large, the code unloading technique does not improve the efficiency of testing.[2] However, since all of the unloading techniques still reduced the space overhead of ID, it may be possible for code unloading to both benefit from and enable the use of heap compression [9, 22].

An additional limitation of this approach is that it may restrict the deployment of the program under test. For instance, executing a program's tests with a Jikes RVM may not directly establish a confidence in the correctness of the same program when it runs on a standard JVM. However, since Zhang and Krintz have shown that their Jikes RVM can efficiently run a wide variety of traditional programs [23], it may be reasonable for the testing RVM to also support the everyday use of the program. Furthermore, the code unloading RVM may impact the performance and/or behavior of the program, thereby invalidating certain tests and improperly inflating correctness estimates. Since the experiments suggest that many unloading configurations lead to similar time and space overheads, testers can handle this shortcoming by using a different unloader each time testing occurs. Varying the unloading method may enable the identification of subtle defects (e.g., problems related to timing and threads) that would not be detected otherwise. Testers may also decide to use a regular JVM when extra testing time is available and/or the project nears a major release.

Threats to Validity. This paper's experiments are subject to internal threats to validity that have the potential to affect the measured variables. Since defects in our research prototype could bias our results, we regularly checked to ensure that the use of adaptive code unloading did not improperly impact the behavior and output of the test suites. By picking several moderate sized programs and test suites and using a real world test automation framework, we attempted to control the external validity threats that may limit our ability to generalize the results. As outlined in Section 6, we also intend to conduct additional empirical studies in an attempt to further mitigate concerns about external validity.

[2] The small number of average code unloading times and unloaded code bodies caused by *S-GC* for the ID application can be attributed to the fact that non-initial unload frequency (i.e., U) was set to three GC cycles and the garbage collector was only invoked twice during testing (see Figure 3 for more details).

5 Related Work

Research related to the testing technique described in this paper includes (i) approaches to testing and simulation of embedded systems and sensor networks, (ii) techniques for executing Java programs in memory constrained environments, and (iii) methods for monitoring and understanding the behavior of Java programs. This paper is primarily distinguished from prior work because of either its focus on test automation or its detailed empirical study. In category (i), Broekman and Notenboom describe a complete testing methodology for embedded software that includes recommendations for managing risk, software inspections, and applying traditional testing approaches to the embedded domain [8]. However, these authors do not specifically address the issues that are associated with the execution of test suites in a memory constrained environment. Testing approaches that are tailored for sensor networks include TOSSIM [15] and Avrora [21]. TOSSIM is a simulator for TinyOS wireless sensor networks that has been used to find real defects within the core TinyOS services [15]. Avrora is also a simulator that improves upon TOSSIM by supporting finer-grained timing of the simulated code [21]. Yet, Levis et al. and Titzer et al. both focus on simulation of resource constrained sensor networks and thus differ from our approach that supports the testing of software in the actual memory constrained execution environment [15, 21].

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	16.1	68.4 ✓
X-GC	16.4	52.8
S-TM	17.1	62.6
X-TM	16.4	45.9
S-CS	17.6 ✓	58.8
X-CS	15.3	54.8

Figure 9. Average Reductions.

In category (ii), Bacon et al. [6] and Sachindran et al. [18] both customize garbage collection algorithms to operate in memory constrained environments. Yang et al. also describe an automatic heap resizing technique that uses knowledge about the actual memory footprint of the virtual machine [22]. It is possible that these garbage collection and heap resizing algorithms can be used in conjunction with the code unloading technique that is part of our current approach to testing. Furthermore, the program partitioning scheme developed by Zhang et al. [24] could also be employed to ensure that tests are executed within acceptable bounds for time and space overhead. In category (iii), Hauswirth et al. [12] present different techniques that can be used to profile and understand the behavior of Java software applications and thus ensure that the question *what to unload?* is answered as accurately as possible.

6 Conclusions and Future Work

It is very important to test a software application in the environment in which it will execute, even if this environment is an embedded one that constrains memory resources. This paper empirically evaluates a testing technique that uses an adaptive code unloading Java virtual machine to ensure that testing is feasible when JVM heap resources are limited. The experiments described in this paper use the Jikes RVM to execute the JUnit test suites of small and moderate scale Java programs. In particular, we measure the time overhead, the average size of native code bodies, and the overall size of the virtual machine process. Our results reveal that it is possible to reduce both the time overhead of testing by 17.6% and the space overhead of the native code bodies by 68.4% while maintaining the size of the Jikes RVM process. Finally, we identify several interesting trade-offs associated with automated testing in memory constrained environments. For instance, the empirical study suggests that code unloading may not be beneficial for programs and test suites with large working sets of active methods. Yet, we judge that when used in conjunction with heap compression and other optimization methods (see below for more details), our approach may be ideal for cutting edge handsets like the T-Mobile G1 with Google Android.

Future research will investigate the impact that garbage collection [6, 18] and heap compression [9, 22] algorithms for memory constrained environments could have upon the performance of testing. We will also develop new approaches to test suite prioritization that re-order the execution of a test suite in an attempt to minimize time and/or space overhead while maximizing metrics such as structural test coverage [17]. Our preliminary work in this area suggests that offline techniques can efficiently identify test suite orderings that effectively minimize the loading and unloading of method code bodies [7]. We will also use the existing Jikes RVM framework to conduct further empirical evaluations with new case study applications. Finally, we will consider operating systems for resource constrained mobile devices (e.g., [11, 16]) in order to implement and evaluate our technique in a real embedded environment.

References

[1] http://muvium.com/.
[2] http://www.embedded-web.com/.
[3] http://tynamo.qindesign.com/.
[4] https://micromatica.dev.java.net/.
[5] http://T-MobileG1.com/.
[6] D. F. Bacon et al. Garbage collection for embedded systems. In *Proc. of 4th EMSOFT*, 2004.
[7] S. Bhadra and G. M. Kapfhammer. Prioritizing test suites by finding Hamiltonian paths: Preliminary studies and initial results. In *Proc. of 3rd TAIC PART (Abstract)*, 2008.
[8] B. Broekman and E. Notenboom. *Testing Embedded Software*. Addison-Wesley, November 2002.
[9] G. Chen et al. Heap compression for memory-constrained Java environments. In *Proc. of 18th OOPSLA*, 2003.
[10] M. Chen and K. Olukotun. Targeting dynamic compilation for embedded environments. In *Proc. of 2nd JVMRTS*, 2002.
[11] R. Guy. *Avoiding Memory Leaks*. AD Blog, 2009.
[12] M. Hauswirth et al. Vertical profiling: understanding the behavior of object-oriented applications. In *Proc. of 19th OOPSLA*, 2004.
[13] G. M. Kapfhammer et al. Testing in resource constrained execution environments. In *Proc. of ASE*, 2005.
[14] R. Lehrbaum. Focus on embedded systems: Embedded Linux and Java—wave of the future? *Linux Journal*, 2002(94):13, 2002.
[15] P. Levis et al. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. of 1st SENSYS*, 2003.
[16] H. Liu et al. Design and implementation of a single system image operating system for ad hoc networks. In *Proc. of 3rd MOBISYS*, 2005.
[17] M. Rummel et al. Towards the prioritization of regression test suites with data flow information. In *Proc. of 20th SAC*, 2005.
[18] N. Sachindran et al. MC^2: high-performance garbage collection for memory-constrained environments. In *Proc. of 19th OOPSLA*, 2004.
[19] N. Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Proc. of 2nd JVMRTS*, 2002.
[20] H. Shim et al. Low-energy off-chip SDRAM memory systems for embedded applications. *TECS*, 2(1):98–130, 2003.
[21] B. L. Titzer et al. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of 4th IPSN*, 2005.
[22] T. Yang et al. Automatic heap sizing: taking real memory into account. In *Proc. of 4th ISMM*, 2004.
[23] L. Zhang and C. Krintz. The design, implementation, and evaluation of adaptive code unloading for resource-constrained devices. *ACM TACO*, 2(2), 2005.
[24] T. Zhang et al. Tamper-resistant whole program partitioning. In *Proc. of LCTES*, 2003.

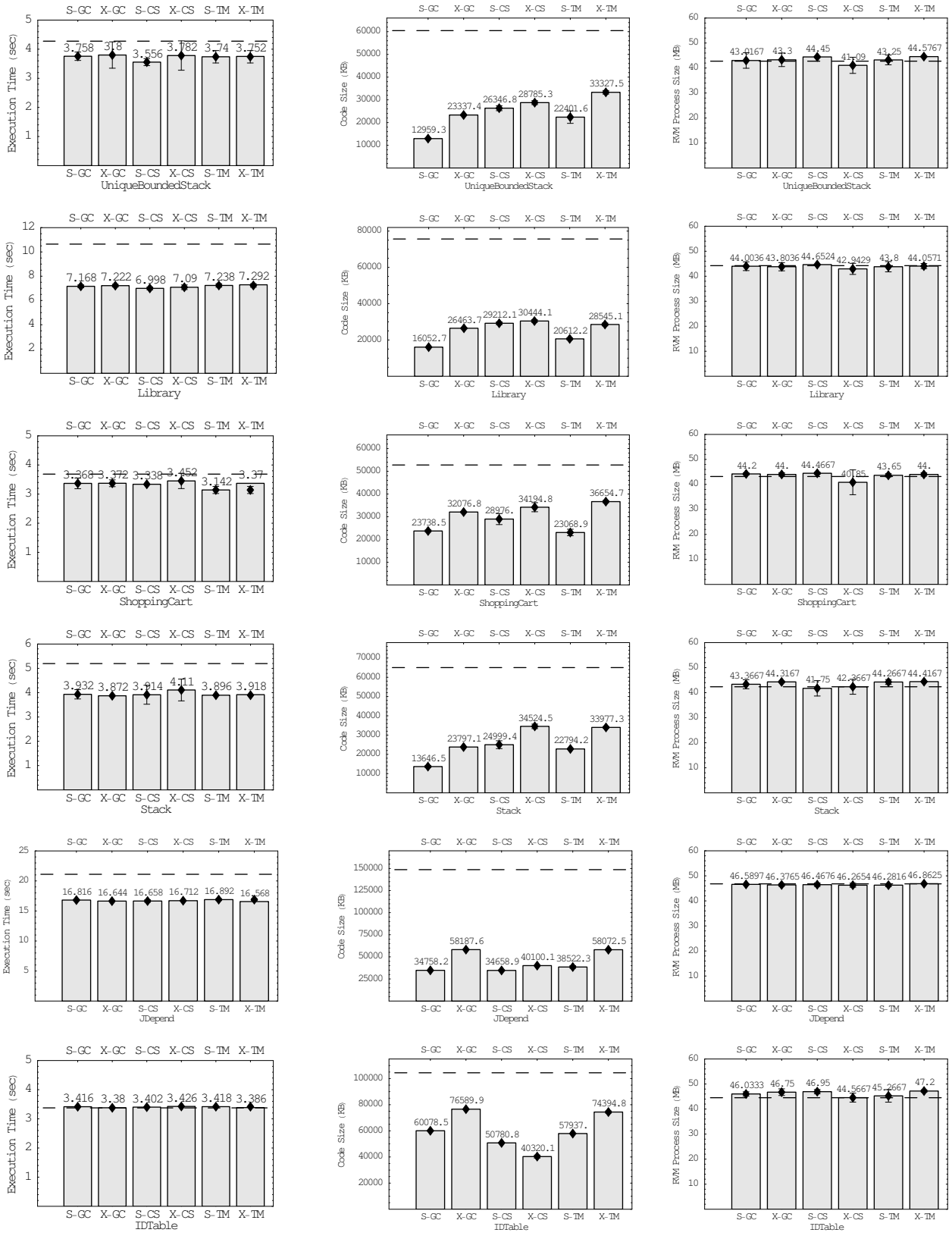


Figure 10. Changes in Test Suite Execution Time, Native Code Size, and RVM Process Size.