Testing in Resource Constrained Execution Environments

Gregory M. Kapfhammer
Dept. of Computer Science
Allegheny College
gkapfham@allegheny.edu

Mary Lou Soffa
Dept. of Computer Science
University of Virginia
soffa@cs.virginia.edu

Daniel Mosse
Dept. of Computer Science
University of Pittsburgh
mosse@cs.pitt.edu

ABSTRACT

Software for resource constrained embedded devices is often implemented in the Java programming language because the Java compiler and virtual machine provide enhanced safety, portability, and the potential for run-time optimization. It is important to verify that a software application executes correctly in the environment in which it will normally execute, even if this environment is an embedded one that severely constrains memory resources. Testing can be used to isolate defects within and establish a confidence in the correctness of a Java application that executes in a resource constrained environment. However, executing test suites with a Java virtual machine (JVM) that uses dynamic compilation to create native code bodies can introduce significant testing time overheads if memory resources are highly constrained. This paper describes an approach that uses adaptive code unloading to ensure that it is feasible to perform testing in the actual memory constrained execution environment. The experiments demonstrate that code unloading can reduce both the test suite execution time by 34% and the code size of the test suite and application under test by 78% while maintaining the overall size of the JVM.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging-*Testing tools*; D.3.4 [**Programming Languages**]: Processors-*code generation, compilers, memory management, run-time environments*

General Terms: Experimentation, Verification

Keywords: test suite execution, code unloading

1. INTRODUCTION

The Java compiler and virtual machine provide enhanced safety, portability, and the opportunity to perform dynamic optimization. The Java programming language is now a popular choice for implementing the software applications that execute on resource constrained mobile and embedded devices like hand-helds and cell phones [12, 15, 18]. In fact, Java is currently being used in resource constrained embedded environments to implement ad hoc and sensor networks [1, 13], XML processors [2], HTTP servers [3] and numeric expression evaluation and function graphing applications [4]. Since resource constrained devices can operate in a variety of complex execution environments that are often difficult and costly to simulate correctly, it is important to test a program in the setting(s) in which it will really execute. Yet, Java virtual machines (JVMs) that use dynamic compilation can create significant testing time overheads when the tests are executed in a memory constrained environment.

Many recent embedded JVMs (e.g., [2, 9, 16]) use a "just it time" (JIT) compiler to compile Java bytecode into native code in an attempt to reduce overall execution time [6]. While virtual machines 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 [18]. If the Java virtual machine's heap cannot continuously store a significant part of the native code and data associated with the test executor, the test cases, and the application under test, frequent garbage collector (GC) invocations will increase the time overhead of testing. If the execution of all or a portion of the tests is omitted in order to reduce the time required to test, the quality of the application could be compromised. Alternatively, if the test suite is not executed in the intended execution setting(s), the tests are less likely to reveal defects related to the program's interaction with the embedded environment.

In light of these concerns, this paper describes a testing technique that uses a JVM that performs adaptive code unloading [18]. The proposed 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 can identify which native code bodies were used the least during the current execution of the test suite and are thus unlikely to be used during the remainder of testing. Code unloading can remove these infrequently used native code bodies from the JVM's heap, which often reduces the amount of time spent performing memory management and subsequently decreases the time overhead associated with testing. This approach ensures that testing can be efficiently performed in the resource constrained environment in which the program will actually execute. In summary, the important contributions of this paper are as follows:

1. The application of code unloading to support efficient testing in resource constrained environments (Section 2 through Section 4).

ASE '05, November 7–11, 2005, Long Beach, California.

2. Detailed experiments that use real world applications to measure time and space reductions and identify:

   (a) The code unloading techniques that demonstrate the greatest reductions in the execution time and code size of an application and its test suite (Section 5.1).

   (b) The characteristics of software applications and their test suites that prohibit the use of code unloading techniques to make testing in memory constrained environments more efficient (Section 5.2).

   (c) The strengths and weaknesses associated with the use of both sample-based and exhaustive profiles of program testing behavior (Section 5.3).

2. TESTING CHALLENGES

A test suite is a collection of test cases that invoke program operations and inspect the results to determine if the methods of program P operate correctly [11]. Definition 1 defines a test suite T that contains Δ_0, an initial test state that describes the preliminary configuration of P and is input to the first test that is chosen for execution. The test suite also contains $\{T_1, \ldots, T_n\}$, the set of tests that are used during testing. Before the execution of each test case T_k the test setup operation S_k performs any necessary initializations (e.g., establishing a database connection or writing to a file or a network socket). A test dependency $D_i \in D$ of the form $D_i = (T_j \rightarrow T_k)$ indicates that test case T_j depends upon T_k and must be executed after T_k.

Definition 1. A *test suite* T is a quadruple $(\Delta_0, \{T_1, \ldots, T_n\}, \{S_1, \ldots, S_n\}, D)$, consisting of an initial test state Δ_0, a test case set $\{T_1, \ldots, T_n\}$, a set of test setup operations $\{S_1, \ldots, S_n\}$, and a set of test dependencies D.

Since Definition 1 does not specify a pre-defined order of execution for the tests within T, Definition 2 defines a valid test execution sequence E that contains a possible ordering of the tests within T that satisfies the dependencies inside of D. The test state Δ_k produced by a T_k could contain the output of the test oracle and/or a portion of the application's state that will be inspected by the tester. The sequence E can contain zero or more executions of the tests within T's test set (e.g., the execution of a test set $\{T_1, \ldots T_{10}\}$ could be configured so that if T_2 fails, tests T_3 through T_5 are skipped, tests T_6 through T_{10} are executed, and then test T_2 is re-executed). The test suite T could be executed at regular intervals or when either the program under test or the execution environment changes.

Definition 2. A *valid test execution sequence* E for test suite T is a pair $(\langle T_1, \ldots, T_r\rangle, \langle \Delta_1, \ldots, \Delta_r\rangle)$ such that $\forall T_k \in \langle T_1, \ldots, T_r\rangle \; \exists T_k \in \{T_1, \ldots, T_n\}$, $\Delta_k = T_k(\Delta_{k-1})$ and $\forall D_i \in D$, $satisfy(\langle T_1, \ldots, T_r\rangle, D_i) = \textbf{true}$.

This paper assumes that all of the $\Delta_k \in \langle \Delta_1, \ldots, \Delta_r\rangle$ and the native code bodies for each $T_k \in \langle T_1, \ldots, T_n\rangle$ will be stored in the GC-managed JVM heap. It is challenging to perform testing in an environment where (i) memory is constrained, (ii) the test suites adhere to Definition 1 and Definition 2, (iii) re-testing frequently occurs because of changes in P or the execution environment, and (iv) the tests and the test states are stored in the heap. For example, it is often not possible to avoid an increase in testing time by executing each test individually because of the test dependencies contained within D. Even if $D = \emptyset$, the time overheads associated with the execution of each test in isolation could be high because of the need to load the virtual machine and the test executor for each test. It is also not possible to simply monitor T's testing of P and use this offline behavior profile to determine when native code bodies can be unloaded. This is due to the fact that both T and P can change frequently during testing and the old behavior profile will not be applicable to the new program and tests.

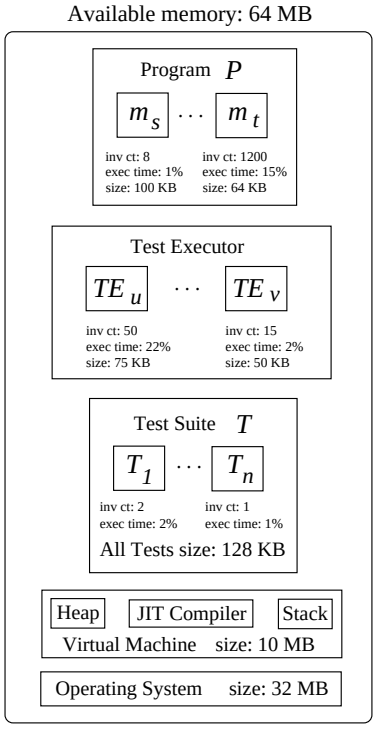

Figure 1: Memory Constrained Testing.

Since the JVM's garbage collector already unloads the data that is stored within the heap, this paper focuses on unloading the native code bodies kept in the heap. It is challenging to use online profiles of the test suite's behavior to adaptively unload code during test suite execution. For example, the recurrent use of the setup operations $S_1, \ldots, S_n$ and the repeated execution of the tests within T is very different than the execution of programs that only run their startup routines once and exhibit phased behavior. While Zhang and Krintz observe that for approximately 70% of their case study applications, 72% or more of the native code bodies are dead after the program's startup phase [18], this trend does not always appear during testing. It is also not possible to simply unload the native code bodies of the tests that have already executed because E can contain the repeated use of a test within T.

The testing technique described by this paper identifies code unloading candidates by using a sample-based or exhaustive online behavior profile to determine which native code bodies are the least frequently used. When native code is removed from the heap, the extra heap space can be used to ensure that the garbage collector is invoked less frequently or the heap can be compressed in order to further reduce the memory footprint of the JVM [8]. Since the tests must execute quickly in a memory constrained environment, the proposed testing technique runs a test suite in a code unloading

JVM that allocates the smallest heap size that ensures the completion of testing and uses any free heap space to guard against GC invocation.

Figure 1 provides an example of a program P that will be tested by a test suite T on a memory constrained device that contains a total of 64 MB of physical memory and devotes 32 MB of the total memory to the operating system. In Figure 1, each of the boxes within the program P, the test executor, and the test suite T represent a native code body that was produced by the JVM's JIT compiler and then stored in the heap. Suppose that the JVM is allocated a total of 10 MB of main memory, the JVM's heap can claim no more than 6 of the 10 MB, and the code bodies from the program (i.e., $m_s, \ldots, m_t$) and the test executor (i.e., $TE_u, \ldots, TE_v$) currently consume 2 MB of heap space. If the garbage collector is triggered when the heap is 75% full, then only 24 tests can be executed before memory management occurs. When heap resources are severely constrained, the additional time overhead incurred by frequent garbage collection causes an over 600% average increase in testing time for all of the applications described in Section 4.

Figure 1 shows that the virtual machine creates a behavior profile that is stored within the native code body and tracks the invocation count ("inv ct"), the percentage of execution time over the life of the program ("exec time"), and the size of individual method bodies ("size"). For example, since method m_s has only been invoked eight times and its execution represents 1% of the entire execution of P, it could be unloaded so that its 100 KB of memory can be used to efficiently execute the remaining tests. Even though it is clearly challenging to perform testing in a resource constrained execution environment, the empirical results in Section 5 indicate that the combination of code unloading and simple program behavior profiles based upon execution frequency can reduce testing time by 34% and space overhead by 78% when JVM heap resources are very limited.

3. CODE UNLOADING

Any memory constrained testing technique that uses code unloading must address the questions *what code must be unloaded?* and *when should code be unloaded?* [18]. This paper uses a JVM that creates either sample-based (denoted *S*) or exhaustive (denoted *X*) profiles of application behavior during program testing so that the least frequently used native code bodies that can be unloaded. This paper also uses JVMs that determine when to invoke an unloading technique by using timers (denoted *TM*), garbage collection triggers (denoted *GC*), or code cache size triggers (denoted *CS*) [18], as described in Figure 2. This paper evaluates the potential of the following code unloading techniques to make testing more efficient: *S-GC*, *X-GC*, *S-CS*, *X-CS*, *S-TM*, and *X-TM*.

For example, $\langle 4, 1, 2, 0.0 \rangle$ would configure *S-GC* or *X-GC* so that the first four GC cycles are assumed to occur during program startup and code unloading occurs every cycle regardless of the residency of the JVM heap. After four cycles of garbage collection, code unloading will initially occur every two cycles. The *S-TM* and *X-TM* code unloading techniques could be described in an analogous fashion. Thus, the tuple $\langle 2, 1, 5, 0.2 \rangle$ indicates that the first two seconds of program testing are considered 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 of execution, the JVM will perform code unloading every five seconds. Finally, 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.

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

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

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

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

Figure 2: Code Unloading Configurations.

4. EXPERIMENT GOALS AND DESIGN

It is important to discern if and how code unloading reduces the time and space overheads that are associated with the execution of the tests that verify the software executing on an embedded device. Equation (1) defines the reduction in space required by native code bodies, denoted $\mathcal{S}_R(P,T)$. The space reduction is the difference between the size of the code bodies before adaptive code unloading is used, $\mathcal{S}_B(P,T)$, and the size of the code bodies after unloading is employed, $\mathcal{S}_A(P,T)$. The percent reduction of space overhead, denoted $\mathcal{S}_R^{\%}(P,T)$ and defined in Equation (2), is the ratio between the space reduction and the space overhead that was incurred before the use of code unloading. The time overhead reduction, $\mathcal{T}_R(P,T)$, and the percent reduction of time overhead, $\mathcal{T}_R^{\%}(P,T)$ could be similarly defined.

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

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

Any testing technique that uses code unloading will incur an additional space overhead to store the program behavior profiles. Furthermore, code unloading could introduce additional time overheads that are associated with (i) creation and maintenance of the profile, (ii) consultation of the behavior profile to determine what, if any, code bodies should be unloaded, (iii) unloading of the code bodies, and (iv) potential reloading of the code bodies that were previously unloaded. To this end, the experiments described in this paper focus on identifying the tradeoffs that different unloading techniques make between the time and space overhead associated with testing. In general, we are interested in answering the question: *can adaptive code unloading reduce the time and space overheads required to perform program testing in a memory constrained execution environment?*

In our experiments, we use a Jikes Research Virtual Machine (Jikes RVM) x86 version 2.2.1 [5] that includes the adaptive code unloading extensions developed by Zhang and Krintz [18]. The RVM was configured to unload all native code bodies that the behavior profile indicated were not in use since the previous invocation of the code unloader and the RVM always re-initialized the profile at the end of an unloading session. We configured the Jikes RVM to operate in two separate memory configurations called *Min* and

Name	*Min* Size (MB)	# Tests	NCSS
UniqueBoundedStack (UBS) [10]	8	24	362
Library (L) [17]	8	53	551
ShoppingCart (SC) [17]	8	20	229
Stack (S) [17]	8	58	624
JDepend (JD)	10	53	2124
IDTable (ID) [14]	11	24	315

Figure 3: Case Study Applications.

Name	*GC*	*CS*	*TM*
UBS	$\langle 4, 1, 1, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
L	$\langle 5, 1, 3, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
SC	$\langle 3, 1, 1, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 2, .5, 1, 0.0\rangle$
S	$\langle 4, 1, 1, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
JD	$\langle 8, 1, 4, 0.0\rangle$	$\langle 49370, 512, 5\rangle$	$\langle 3, .5, 1, 0.0\rangle$
ID	$\langle 1, 1, 3, 0.0\rangle$	$\langle 65536, 8192, 5\rangle$	$\langle 2, .5, 1, 0.0\rangle$

Figure 4: Unloading Configurations for the Jikes RVM.

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 resource constrained environment that is common when testing occurs on an embedded device.

All of the experiments were conducted on a workstation with dual Intel Xeon Pentium III processors and 512 MB of main memory. The workstation was running GNU/Linux with a 2.4.18-14smp kernel. In an attempt to increase the realism of the experiment, the Jikes RVM was configured to only use one of the two available CPUs. Figure 3 reviews the most important characteristics of the case study applications and their test suites. The *Min* size attribute is the empirically identified minimum heap size and NCSS is the number of non-commented source statements, as calculated by JavaNCSS 21.41. For our experiments, we selected Java programs that contained test suites that could be executed in the JUnit 3.8.1 test execution framework by a Jikes RVM.

Figure 4 describes the different Jikes RVM configurations that were selected to execute the test suites because they demonstrated the best performance in terms of time and space overhead reductions. Each of these virtual machine configurations was produced by ten minutes (or less) of exploratory experiments that systematically modified the C, UC, and U parameters for the *GC* and *TM* techniques and the Z_{init}, Z_{incr}, and U_{CS} parameters for the *CS* approach. To ensure that code unloading is performed regardless of heap residency, we set $H = 0.0$ for the *GC* and *TM* techniques and we did not modify this parameter. Since our preliminary exploration of the space of all possible configuration tuples was not exhaustive, it is possible that the RVM configurations listed in Figure 4 are not optimal for the application. In order to ensure a fair comparison, the sample-based and exhaustive profile RVMs used the same configuration tuple. We executed each test suite on every application five times in order to compute arithmetic means and standard deviations. The experiment results show uniformly small standard deviations. Due to space constraints this measure of dispersion is omitted in Section 5.

5. EXPERIMENT RESULTS

5.1 Time and Space Reductions. The average execution time of a test suite, across all case study applications, was 1.049 sec in the *Full* JVM and 8.047 sec in the *Min* JVM. This 667% increase in testing time overhead indicates that there is a clear need for code unloading to make testing more efficient. Figure 5 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). The check marks (i.e., "✓") indicate that the code unloading technique that produces the most noticeable space reduction does not always produce 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* technique often unloads code too aggressively and reduces space overhead at the cost of incurring additional time overhead to reload previously unloaded code bodies. 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 (this problem could potentially be resolved by increasing H, the heap residency parameter). Since *S-GC* triggers code unloads more than *S-CS* on average (total number of unloads: 11.4 - *S-GC* vs. 2.0 - *S-CS*), it does not reduce test execution time any more than *S-CS* (24.3% - *S-GC* vs. 24.7% - *S-CS*) even though it creates a larger space reduction (79% - *S-GC* vs. 61% - *S-CS*).

The experiment results in Figure 5 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 time overhead reductions. 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 6 presents the average time and space percent reductions across all of the chosen case study applications. These results demonstrate that *S-GC* most effectively reduces space overhead and *S-CS* is the most effective reducer of time overhead. Finally, these results clearly indicate that, for the selected applications, it is beneficial (on average) to employ code unloading in an attempt to reduce the time and space overhead of executing tests when memory is constrained.

5.2 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 5 shows that the time overhead associated with testing for ID is always increased by a small factor (e.g., $\mathcal{S}_R^{\%}(P,T)$ ranges from $-.29$ to -1.4). While no unloading technique causes ID to perform more than four unloads, *S-CS* and *S-TM* both require the unloading of more than 635 native code bodies. When ID's number of unloaded bodies are compared with the corresponding value for the similarly sized L application (272.4 - *S-CS*, 533 - *S-TM*, 542 - *S-GC*), it is clear that ID must be unloading the native code of an external library. In fact, ID uses the Apache log4j logging utility throughout testing and thus these 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. Since all of the unloading techniques still reduced the space overhead of ID, this indicates that the testing of applications with large working sets can still benefit from code unloading because it enables the use of heap compression if the RVM footprint must be reduced [8].

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 5: Reductions for (a) `UBS` (b) `L`, (c) `SC`, (d) `S`, (e) `JD`, and (f) `ID`.

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 6: Reductions Across All Applications.

5.3 Profile Types. The results provided by Figure 5 also show that the exhaustive program behavior profile does not normally enable the reduction of time overhead noticeably more than the sampled profile. Since the exhaustive profile is embedded within the code bodies [18], it frequently creates an average code size that is greater than the corresponding code size for the sample-based technique. This greater code size does limit the potential of code unloading if the test suite contains either a greater number of tests or tests with larger code bodies. 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. This can be attributed 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 virtual machine heap [18].

6. CONCLUSIONS AND FUTURE WORK

This paper explains 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 proposed testing technique makes it possible to more effectively isolate defects and establish a confidence in the correctness of programs by improving the efficiency of resource constrained testing. The experiments described in this paper use the Jikes RVM to execute the JUnit test suites of small and moderate scale Java programs in order to measure the time overhead and the average size of native code bodies. The results reveal that it is possible to reduce both the time overhead of testing by 34.3% and the space overhead of the native code bodies by 78.8%. Future research will investigate the impact that garbage collection [7] and heap compression [8] algorithms for memory constrained environments could have upon the performance of testing. We will also develop new approaches to test prioritization that reorder the execution of a time suite in an attempt to minimize time and/or space overhead while maximizing metrics such as structural test coverage [14]. Finally, we will conduct further experiments that incorporate additional case study applications, test suites, and real embedded execution environments (e.g., [13]).

7. REFERENCES

[1] `http://www.ajile.com/`.

[2] `http://www.embedded-web.com/`.

[3] `http://tynamo.qindesign.com/`.

[4] `https://micromatica.dev.java.net/`.

[5] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeno in Java. In *Proc. of the 14th OOPSLA*, pages 314–324, 1999.

[6] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93:449 – 466, February 2005.

[7] David F. Bacon, Perry Cheng, and David Grove. Garbage collection for embedded systems. In *Proceedings of the 4th EMSOFT*, pages 125–136, 2004.

[8] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Proc. of the 18th OOPSLA*, pages 282–301, 2003.

[9] Michael Chen and Kunle Olukotun. Targeting dynamic compilation for embedded environments. In *Proc. of the 2nd JVM*, pages 151–164, 2002.

[10] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, September 2004.

[11] Gregory M. Kapfhammer. *The Computer Science Handbook*, chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004.

[12] Rick Lehrbaum. Focus on embedded systems: Embedded Linux and Java—wave of the future? *Linux Journal*, 2002(94):13, 2002.

[13] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gun Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proc. of the 3rd MobiSys*, June 2005.

[14] Matthew Rummel, Gregory M. Kapfhammer, and Andrew Thall. Towards the prioritization of regression test suites with data flow information. In *Proc. of the 20th SAC*, Santa Fe, New Mexico, March 2005.

[15] Tom Sanders. Java sets sail for the final frontier. May 2005. `http://www.vnunet.com/`.

[16] Nik Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Proc. of the 2nd JVM*, pages 119–126, 2002.

[17] P. David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In *Proc. of the 2nd XP/Agile Universe*, pages 131–143, 2002.

[18] Lingli Zhang and Chandra Krintz. Adaptive code unloading for resource-constrained JVMs. In *Proc. of LCTES*, pages 155–164, 2004.