Adaptive Code Unloading for Efficient Dynamic Compilation in Resource-Constrained Environments UCSB Technical Report 2003-14, June 27, 2003

Lingli Zhang Chandra Krintz
Department of Computer Science
University of California, Santa Barbara
{lingli_z,ckrintz}@cs.ucsb.edu

Abstract

Java virtual machines (JVMs) have become increasingly popular for the execution of a wide range of applications on mobile and embedded devices. Most JVMs for such devices use interpretation for bytecode execution. However, JVMs that use dynamic compilation have been shown to enable significant performance improvements. One disadvantage of using a compile-only approach in a resource-constrained environment is that it uses more memory than interpretation to store compiled code for reuse.

With this paper, we address this limitation with a novel framework for adaptive compiled code unloading that can be integrated into any compilation-based JVM. In our framework, a central unloader monitors system resources to adaptively determine the aggressiveness with which to apply unloading. The unloader utilizes both offline and online profile information using various strategies to unload dead or infrequently used code. We evaluate our framework using a non-optimizing compiler JVM configuration as well as a state-of-the-art adaptive optimization configuration (in which hot methods are identified and incrementally optimized). We find that by using adaptive code unloading, we reduce heap residency to significantly improve garbage collection performance. Our results indicate that, for the benchmarks studied, our system reduces program execution time by 22.32% for the dynamic compiler JVM and 11.66% for the adaptively optimizing JVM, when memory is constrained.

1 Introduction

Java Virtual Machines (JVMs) [25] have become increasingly popular for the execution of a wide range of applications on mobile and embedded devices. Researchers estimate that there will be over 720 million Java-enabled mobile devices by the year 2005 [32]. This wide-spread use of Java for embedded systems is the result of the increased capability of modern and next-generation mobile devices, the ease of program development using the Java language [6], and the security and portability enabled by JVM execution.

A Java virtual machine (JVM) translates mobile Java programs from an architecture-independent format, bytecode, into native code for execution. Many JVMs [31, 12, 21] perform translation using interpretation since interpreters are simple to implement and impose no perceivable interruption during execution. However, interpreted program execution time can be orders of magnitude slower than compiled code due to poor code quality, lack of optimization, and re-interpretation of previously executed code. As such, interpretation wastes significant resources of embedded devices, e.g., CPU, memory, battery, etc. [34, 14]

To overcome the limitations that JVM interpretation imposes, next-generation JVMs [1, 30, 2] employ just-in-time (JIT), i.e., dynamic, compilation. These JVMs dynamically compile the bytecode stream of each method as it is invoked into machine code. The resulting execution performance is higher than for interpreted bytecodes since native method code is stored and reused each time a method is invoked repeatedly. In addition, compilation of an entire method at once exposes opportunity for optimization and hence, code quality is significantly improved. As such, dynamic compilation uses underlying resources significantly more efficiently; such use is critical for resource-constrained devices.

Compile-only JVMs require that native code blocks be stored in memory so that they can be reused during execution. In addition, native code is much larger than their bytecode equivalents. This requires additional heap memory and hence, introduces memory management overhead, i.e., garbage collection (GC). Commonly in mobile devices, memory is severely constrained and consumes significant battery power. Hence, techniques are needed that enable compile-only JVMs to make more efficient use of memory.

To this end, we have developed a code unloading framework that attempts to adaptively balance not caching any code (as is done in an interpreter-based JVM) and caching all generated code (as in a compiled-based JVM), according to dynamic memory availability. As such, our system trades off the overhead of memory management for that of dynamic compilation: unloading improves GC performance and unloaded methods that are later invoked are automatically re-compiled prior to execution. Since compilation time is significantly less than GC time when memory is limited, we are able to significantly improve program performance for resource-constrained devices – by up to 22.32% on average for the programs studied.

1.1 Code Unloading Opportunities

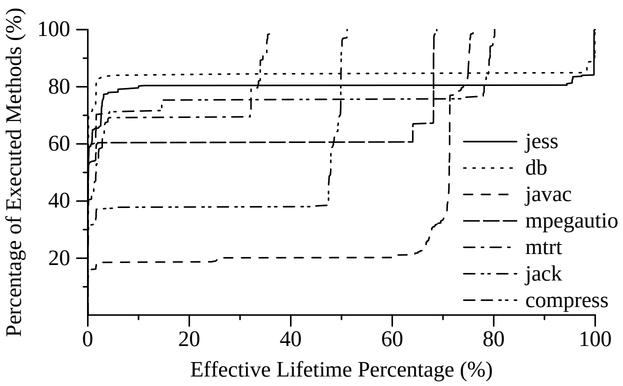
The size of compiled native code is much larger than that of its corresponding bytecode. The table in Figure 1(a) shows the sizes of both bytecode and compiled native code (columns 2 and 3) for the SPECJVM98 benchmarks [27]. These values include only those methods invoked by the program, including Java libraries (but not JVM methods). We collected the data using the dynamic, non-optimizing (fast) compiler in the IBM Jikes Research Virtual Machine (JikesRVM) [2]. On average, native code is 6-8 times larger than bytecode. Since a dynamic (JIT) compiler only compiles methods that are used (lazy compilation [24]), it avoids using memory for unexecuted code. However, the code size required for execution is still significant.

Through other empirical experiments, we found that a large portion of executed code is used only during program startup (the initial 10% of execution). After startup, this code is dead, i.e., never invoked again, but remains in the system and is managed by the garbage collector. As such, dead code will be repeatedly accessed during GC: marked as live if a mark/sweep collector is used or possibly repeatedly moved around in memory if a copying or compacting collector is used. As such, unused methods can consume considerable resources unnecessarily. The final column of the table shown in Figure 1(a) shows the dead code size (in KB) in the system after the first 10% of execution time. In parenthesis is the percentage of the total executed code size that is dead. If we are able to remove this dead code from the system, a large amount of memory can be made available to the system for the majority of program execution.

In addition to dead code, many methods that are live following startup or are compiled after startup are only active for a very short time. These results are presented in Figure 1(b). The graph shows the cumulative distribution functions of compiled code's *effective lifetime percentage*, i.e., the percentage of the effective method lifetime (the time between its first and last invocation) to the total method lifetime (the time from its first invocation to the end of the program). A point, (x, y), on a curve indicates that $y\%$ of methods have an effective

Benchmarks	Byte code (KB)	Native code (KB)	Dead after startup (KB) (Pct. Dead)
compress	12.4	98.4	70.8 (72%)
db	14.5	105.3	89.2 (85%)
jack	42.4	284.9	72.5 (26%)
javac	78.3	468.5	75.9 (16%)
jess	32.9	223.1	167.9 (75%)
mpegaudio	56.6	455.4	357.4 (79%)
mtrt	21.1	161.3	117.6 (73%)

(a)



(b)

Figure 1: In Table (a), we show the code size (in KB) for bytecode and native code (columns 2 and 3). Native code size is significantly larger. Column 4 shows the amount of code (in KB) in the system after the first 10% of execution time that is never used again (the percentage of code that is dead is shown in parenthesis). The graph in (b) is the CDF for method effective lifetime percentage: A point, (x, y), on a curve indicates that $y\%$ of that benchmark's executed methods have an effective lifetime (time between its first and last invocation) of less than $x\%$ of total method lifetime (time from its first invocation to the end of the program).

lifetime of less than $x\%$ of total method lifetime. For most of the benchmarks, over 60% of methods are effectively live for less than 5% of the total time in which they are in the system.

Finally, we also found that there are some methods with long effective lifetimes that are rarely invoked or are invoked with long intervening time intervals. Such methods are also candidates for a more aggressive form of code unloading. During the time that a method is inactive, we can unload the code to ease the pressure on the GC system. When the method is later invoked, it will be recompiled dynamically. When the cost of recompilation is less than the improvement we gain in GC performance, unloading will improve performance. Overall, there are many different opportunities that we can exploit using code unloading to improve program execution time.

1.2 Contributions

This paper makes the following contributions.

- **Opportunity Analysis**: It provides an empirical analysis of the code unloading opportunities (Section 1.1).
- **Analysis Framework**: It presents a novel code unloading framework that automatically unloads native code to reduce the overhead of performing garbage collection. This framework facilitates the implementation and empirical evaluation of unloading strategies.
- **Adaptive Algorithms**: Since heap memory use is dynamically changing, we must account for such changes to improve performance. This paper describes a number of techniques that use dynamically changing program and system memory behavior to guide code unloading.
- **Experimental Results**: It presents an empirical evaluation of different variations of our adaptive unloading techniques. The results indicate that our system enables average performance improvements of 22.32% for a non-optimizing compiler system and 11.66% for a state-of-the-art adaptive optimization system for the programs studied.

2 The Code Unloading Framework

To empirically evaluate the benefit enabled by unloading compiled code, we designed the unloading framework shown in Figure 2; this framework can be incorporated into the architecture of any JVM (that implements dynamic compilation). The darkened components in the figure identify our JVM extensions. The *Code Unloader* is the control center of our system. It decides when methods should be unloaded as well as which methods to unload. When a method is selected for unloading, the unloader replaces its address (stored in a table) with that of a recompilation stub. Once the address is replaced, the native code block for the method is no longer reachable by the program and the storage will be reclaimed during the next garbage collection cycle. The recompilation stub is similar to the compilation stub used for lazy, dynamic, compilation [24, 2] but contains additional information that guides recompilation if reloading should occur. If the method is ever invoked again, the recompilation stub causes it to be compiled again prior to execution.

The decisions of the code unloader are guided by information collected from a number of different sources. First, we collect online profile information about the invocation activity of each method. To enable this, we extended the dynamic compiler to instrument methods

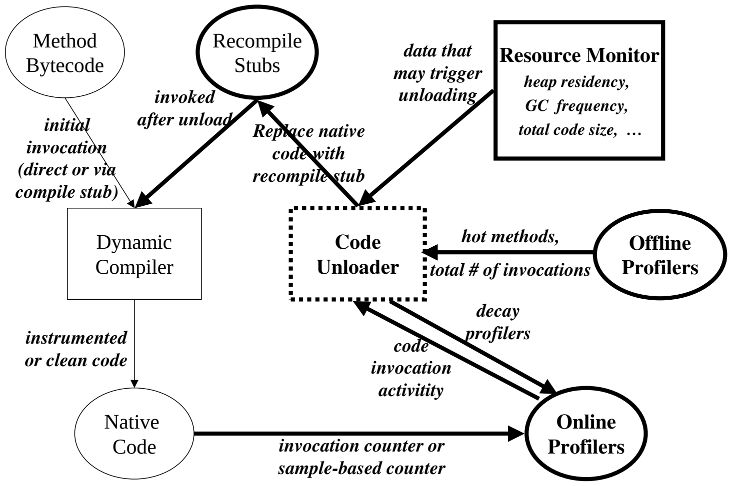


Figure 2: Overview of our code unloading framework

as they are translated with instructions that count method invocations. We consider both exhaustive and sample-based invocation counts; in addition, we decay the counter values [4] so that more recent activity is given higher priority. In addition, the unloader considers profile statistics that are collected off-line. The framework is flexible and hence, can be configured to disregard either type of profile information or to consider additional statistics. Finally, the unloader considers dynamic resource characteristics collected by the *Resource Monitor* component.

The resource monitor forwards information about resource behavior to the unloader. This component can be easily extended to monitor various types of resources. Currently, we use it for memory; the monitor collects heap residency data, garbage collection invocation frequency, and native code size. Using this information, the unloader can decide *adaptively* when unloading should commence. That is, if plenty of resources are available, e.g., no GCs are occurring, residency is low, etc., the unloader suspends all unloading activity. When resources become heavily loaded, the unloader will initiate unloading. In addition, it uses more aggressive unloading strategies (described below) as resource availability becomes critical.

The unloading strategies are algorithms that use resource consumption, application behavior, and offline statistics to answer two questions: "When should unloading take place?" and "What methods should be unloaded?". To answer these questions, the unloader must predict the cost of unloading as well as its benefit. Moreover, since the unloader (and other framework components) operate *while* the program is executing, it must be very efficient so as not to impose significant overhead, i.e., CPU time, memory use, battery consumption. We describe the unloader decision strategies in the following subsections.

2.1 When to unload?

To trigger code unloading, the resource monitor measures memory usage periodically. One way in which we can implement this is to use a timer. However, a regular time interval will not necessarily reflect memory activity. We may miss important events if the period is too long or introduce overhead if the period is too short. In addition, we only want to know when the available memory is getting low. To catch such program behavior, we instead perform measurement during garbage collection since GC is only invoked when the available heap memory becomes limited. At the end of each garbage collection cycle, the resource monitor measures various GC statistics and forwards them to the unloader.

One possible measurement that indicates memory usage is *heap residency*. If the system is short of memory, the heap residency will be high following garbage collection. Currently, users of our system can specify a threshold of heap residency as a command line parameter and use it to trigger the unloading process. One disadvantage to using heap residency is that doing so may raise a false alarm. Some programs may allocate only at the beginning of their execution. In this case, the heap residency may remain high and exceed the threshold; however, it is not necessary to perform unloading continuously because no additional allocations will be performed. As such, heap residency alone is not accurate enough to enable the unloader to make the correct unloading decision.

To avoid false alarms, we use heap residency information indirectly by considering GC frequency. If memory availability becomes severely limited and heap residency remains high, the garbage collector will run frequently. To capture this behavior, the resource monitor forwards the percentage of execution time (so far) that is spent in garbage collection to the unloader to adjust the frequency of the unloading sessions.

As we articulated in Section 1.1, for most benchmarks over 70% of code is dead by completion of the first 10% of program execution time (startup period). To exploit this behavior, the unloader applies unloading more aggressively during this time. That is, we use different unloading strategies for different phases in program lifetime. Since we cannot know the program lifetime (and hence, the number of seconds during the initial 10%), we estimate it using GC cycles. We consider the first 4 GC cycles (empirically determined and can be specified at the command line) as program startup time. During this period, we base the unloading decision on the heap residency alone. This facilitates more aggressive unloading by the unloader. Following this period, we use the percentage of time spent in GC to avoid false alarms in the steady state of program execution.

To reduce the overhead of unloading, the unloader also considers the frequency with which it triggers unloading. If the unloader has performed unloading within the past several GCs, it foregoes unloading until a longer period of time has past. This "unloading window" contains a fixed number of garbage collection cycles and unloading is performed by the unloader only once for each window. The resource monitor adjusts the size of this window dynamically according to the runtime memory status. A smaller window indicates that more aggressive code unloading is needed. Currently, we use a simple algorithm to determine the unloading window size which divides a minimal window size specified by a user (default is 10) by the percentage of time spent in GC; this value is decremented upon each GC and when it reaches 0, unloading is performed.

At the end of each unloading session a new unloading window size will be determined using this algorithm. In addition, at the end of each GC cycle, the unloader estimates the window size for the next unloading session. If the unloader finds that the estimated window size is smaller than the currently used window size (indicating that memory has become highly constrained), the unloader adjusts the window size to current memory status to trigger more aggressive (more frequent) unloading.

2.2 What to unload?

To determine which methods to unload, we use both online and offline profiling. The profiles we gather identify the effective lifetimes of methods (the time between first and last invocation). To collect profiles online, we modified the compiler to instrument methods with instructions that are executed upon method return. We keep a mark bit for each method compiled; each time a method returns, it first sets its mark bit to 1. When the unloader triggers unloading, it unloads all methods with unmarked bits and resets all marked bits. This mechanism unifies unloading of dead and infrequently invoked methods.

This simple online profiling approach may be too aggressive if the unloading interval is very short. When the unloader is overly aggressive, many methods that are unloaded are later re-used requiring recompilation (which introduces overhead). However, the adaptive unloading window implemented by the resource monitor guarantees that the unloading interval will be long enough to avoid such overhead.

In addition, this profiling approach is exhaustive (we refer to it as *On-X* in our results – "On" for online, "X" for exhaustive). That is, it guarantees that every method that is invoked since the last unloading session will have its mark bit set. As such, the unloader will only unload methods that have not been used (since the last unloading session). Such exhaustive profiles introduce overhead for instrumentation in every method invoked.

To reduce the overhead of instrumentation, we also considered a sample-based approach to online profiling, called *On-S*. For this strategy, the compiler inserts no instrumentation. Instead, the resource monitor periodically (default is 10ms) sets the mark bits of the two top methods on the invocation stacks of application threads. We determined this value empirically using the trade off between the significant overhead required for stack scans and incorrectly unloading a used method. The latter causes used methods to be unloaded and recompiled if reused, introducing overhead that may negate any benefit achieved from unloading. By marking only the top two methods on the stack, we are attempting to balance the two overheads.

This sample-based approach is also adaptive in that it can be turned off when there is sufficient memory available. As such, we can turn off *all* overhead that our unloading system may introduce when memory is not constrained (or possibly when the mobile device is plugged in). Using exhaustive profiling, instrumented execution is always used (turning it off would require recompilation). Both online approaches trigger code unloading only when memory becomes constrained.

We also investigated the efficacy of using a combination of both offline and online profiling information. Offline, we collected the total invocation count, which we refer to as *maxCallCount* hereafter, for each method. We annotated this value in the bytecode of the class file as a method attribute. We modified the JikesRVM to recognize the *maxCallCount* attribute. To enable this, we extended an annotation system that we developed as part of prior work [23]. We used the same input offline as we did online; that is, we had perfect information about the *maxCallCount*.

We investigated variants that make use of *maxCallCount*. For the first, called *OnStack*, we instrument each method prior to its return. The inserted instructions increment a counter and check whether the counter equals *maxCallCount*. If the counter and *maxCallCount* are equal, indicating that this is the last invocation of the method, the method is unloaded immediately prior to the transfer of control back to the method caller. The second variant, *GCBased*, triggers unloading during garbage collection when memory availability becomes severely limited (as indicated by the resource monitor). We instrument methods with instructions that increment a counter for each invocation. During GC, the system checks the the invocation counts of all methods and unloads any method with a count equal to the *maxCallCount* of each. *OnStack* performs incremental unloading (and requires an extra compare instruction) but does not adapt to memory availability. *GCBased* unloads all methods at once that have reached their maximum invocation count. Neither offline variant considers infrequently invoked methods for unloading.

We also use offline profile information to indicate to the unloader which methods are used most frequently. As described in Section 3.3, we use this information to treat such *hot* methods separately.

2.3 JVM Configuration: Baseline and Adaptive Compilation

In the evaluation of our unloading system, we consider two different types of JVM configuration. The first is *fast compilation*, in which all methods invoked by an executing program are compiled with a very efficient compiler that performs no optimization. This configuration is similar to that of the Microsoft Common Language Runtime (CLR) [5] of .NET Compact Framework [35] and other JVMs that use a simple JIT for bytecode method translation. A JVM that uses fast compilation avoids the compilation overhead and reduces the complexity of a JVM that implements a highly optimizing compiler.

The second configuration that we consider is *adaptive compilation*. The adaptive configuration is a technique used in state-of-the-art JVMs [2, 9, 30, 15] that enables optimized performance with very little compilation and optimization overhead. Two compilers are used by such systems, one fast, non-optimizing, and one slow, highly- optimizing. [1] All methods are initially fast-compiled without optimization. In addition the fast-compiler instruments each method so that online profiling can be performed by the system. As methods execute, various statistics are gathered, e.g., method invocation counts, hot call paths, etc., to help the system identify hot methods. Hot methods are recompiled using the optimizing compiler. Instrumentation is inserted for optimized methods also, so that additional levels of optimization can be applied as needed. As such, methods that impact overall execution time execute very efficiently without the overhead required to optimize all methods.

In our unloading system, using the adaptive configuration, methods may be compiled at different optimization levels. As mentioned above, a method slowly progresses through optimization levels according its hotness. All levels of optimization require significantly more time than fast compilation. Therefore, it may be more efficient to recompile unloaded code using fast compilation. However, an optimized method is hot; if it remains hot, it will again have to progress through the optimization levels requiring a long period of unoptimized execution and significant compilation overhead. Thus, the recompilation overhead due to the unloading of methods that are later reused imposes a much larger cost.

We investigated three strategies to improve the efficiency of code unloading when using an adaptive optimization configuration. For the first, we add an optimization level hint to the recompilation stub. If the unloader unloads a hot method that is later invoked, the system recompiles the method at the optimization level it was at when it was unloaded. This strategy eliminates unoptimized execution of hot methods and is called *RO* (Reload Optimized methods using the optimization hint) in our results. With the second strategy, we avoid unloading hot methods altogether; the unloader checks whether a method has been optimized and only unloads it if it was fast-compiled. We call this strategy *EO* (Exclude unloading of Optimized methods). However, some programs have a relatively large percentage of hot methods. For example, *javac* has 78 out of 876 hot methods while *db* only has

[1]HotSpot [15] uses a fast interpreter and a slow, highly-optimizing compiler instead of two compilers, to achieve the same effect.

3 out of 151 hot methods. Our third strategy accounts for such cases. Optimized methods will be unloaded but we delay unloading of them until they are unused for two consecutive unloading sessions. We call this strategy, DO(Delay unloading of Optimized methods).

3 Results

To evaluate the efficacy of code unloading we performed a series of experiments. In this section, we describe our experimental methodology and then present our results.

3.1 Experimental Methodology

We implemented our code unloading framework in an open source dual-compiler JVM from IBM T.J. Watson Research Center, called JikesRVM (v2.2.1) [2, 18]. JikesRVM is written in Java but all VM code is built into the boot images; as such, we do not consider this code as part of our code unloading evaluation. We evaluate code unloading using two configurations: FastBaseSemispace and FastAdaptiveSemispace. "Fast" indicates that the VM code is built into the boot image. "Semispace" indicates that the garbage collector used in the system is a copying collector. "Base" indicates that it is the fast, non-optimizing configuration. "Adaptive" indicates that adaptive optimization is used (methods are initially fast compiled and instrumented; hot methods are optimized at increasingly higher levels).

To gather our results, we repeatedly executed a set of benchmarks on a dedicated Toshiba Protege 2000 laptop (750Mhz PIII Mobile) running Debian Linux v2.4.20. The benchmarks include the SpecJvm98 suite [27] and Java Cup v2.0 [17] (jcup). We execute the benchmarks using the large input size (100) for all SpecJvm benchmarks and "parser.cup" for jcup (provided as part of the Java Cup distribution). For each benchmark, we identified the minimal heap size that each application can run and used that size in each of our experiments.

We show the general statistics of the benchmarks in Table 1. We gathered this data using the reference system that does not include any extensions for our code unloading framework. We refer to this system as *clean*. This is the system to which we compare our unloading strategies. The left half of the table is for the fast configuration and the right half is for the adaptive configuration. For each half, the first column is the native code size in kilobytes (KB). Only those methods that are invoked are compiled (application and library). The

Benchmarks	Fast Configuration Code Size (KB)	Fast Configuration Init Heap Size (MB)	Fast Configuration Exec Time (s)	Adaptive Configuration Code Size (KB)	Adaptive Configuration Init Heap Size (MB)	Adaptive Configuration Exec Time (s)
compress	98.4	20	89.2	143.8	22	38.6
db	105.3	22	109.4	157.8	24	107.4
jack	284.9	6	874.5	372.4	9	181.2
javac	468.5	24	190.9	582.8	26	201.4
jcup	167.2	6	4.0	237.6	7	8.6
jess	223.1	8	467.8	311.8	11	206.1
mpegaudio	455.4	9	74.5	541.4	12	42.9
mtrt	161.3	18	502.2	237.8	22	93.8

Table 1: Benchmark Characteristics

second column shows minimum size of the heap that is required for each benchmark to run to completion (identified empirically). The last column is the application execution time using the minimal heap size. We must assign larger minimal heap sizes for execution using the adaptive configuration since the execution environment includes the measurement and optimization system that enables adaptive optimization.

3.2 Fast JVM Configuration

We first present results for the fast configuration. For this configuration, all methods are fast compiled without optimization when initially invoked.

We investigate the four different variants described in Section 2.2: *Off-OnStack*, *Off-GCBased*, *On-X* and *On-S*. The first two (with the "Off-" prefix) use a *maxCallCount* for each method, which we collected using offline profiling. This value is the maximum number of invocations for each method. We use this perfect information to guide both incremental unloading (*Off-OnStack*) and adaptive unloading (*Off-GCBased*). The other variants, *On-X* and *On-S*, use online profiling information only. "X" indicates exhaustive profiling, and "S" indicates sample-based profiling. All variants except for *Off-OnStack* trigger unloading only when memory becomes constrained.

Figure 3 shows the impact that code unloading has on performance. The y-axis is the percent reduction in execution time over the clean, reference system. We show data for each variant and each benchmark. The last group of bars shows the average improvement across all benchmarks.

By reducing the amount of live data in the system, code unloading, in general, significantly improves performance since less time is spent in GC. Most benchmarks have positive

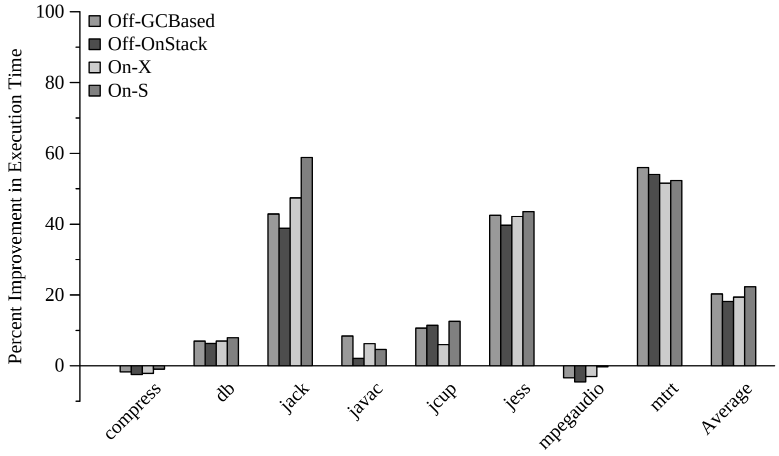


Figure 3: Percent improvement in execution time due to code unloading for the fast configuration. These results show improvements over the clean, reference system (no unloading). Bars with the "Off-" prefix use an offline profile and those with an "On-" prefix use online profiling. For online measurements, "X" indicates exhaustive profiling and "S" indicates sample-based profiling. For offline measurements, "OnStack" indicates incremental unloading as soon as method returns from its last invocation and "GCBased" indicates that unloading is performed only when resources become constrained.

performance improvement with our unloading mechanism. Improvements for *jack*, *jess* and *mtrt* are significant (over 40%). This is due to the continuous memory allocation requirements of these three applications. When memory is critical, continuous memory allocations trigger frequent garbage collections, and thus frequent and effective unloading. Alternately, compress and mpegaudio have no performance improvement. This is because both benchmarks have relatively small memory requirements (18 and 3 GC cycles, respectively). In this case, the unloading benefit is smaller than the overhead introduced by profiling and unloading mechanisms. However, the overhead for these two benchmarks is less than 5% for all strategies. For *On-S*, the overhead is less than 1%.

For the two methods that use annotation of offline profile information (*maxCallCount*), *Off-GCBased* performs better than *Off-OnStack* for most benchmarks since *Off-OnStack* must compare the invocation count against *maxCallCount* before every method return. These results also indicate that releasing dead code as soon as possible, i.e., incrementally, is not necessary.

For the two methods that use online profiling, *On-S* performs better than *On-X* due to its minimal profiling overhead. For example, *On-S* reduces the overhead introduced by exhaustive profiling for mpegaudio from -3.02% to -0.30%. On average, *On-S* is the best among all four variants (up to 58.82% and with average of 22.32%). The average improvements of *Off-OnStack*, *Off-GCBased* and *On-X* are 18.19%, 20.29% and 19.41%, respectively. Online profiling does not require ahead-of-time efforts of offline profiling and annotation, and yet enables similar performance improvements; in some cases even better performance. Online techniques can enable improvements in performance to a larger degree since they consider infrequently invoked methods (not just dead methods). As such, these results indicate that both exhaustive online profiling and offline profiling is unnecessary.

	Compilation Time (s)					Garbage Collection Time (s)				
Benchmarks	clean	OffGCBased	OffOnStack	OnX	OnS	clean	OffGCBased	OffOnStack	OnX	OnS
compress	0.03	0.02	0.03	0.03	0.04	8.80	8.86	8.82	8.86	8.77
db	0.03	0.03	0.03	0.03	0.04	43.39	36.51	37.35	37.22	36.02
jack	0.07	0.10	0.08	0.71	1.44	857.47	483.19	518.18	442.78	342.62
javac	0.15	0.18	0.16	0.23	1.09	161.16	144.55	155.04	148.98	151.71
jcup	0.05	0.05	0.05	0.05	0.06	3.17	2.74	2.72	3.13	2.67
jess	0.13	0.16	0.10	0.15	0.39	438.15	238.77	253.72	240.54	234.36
mpegaudio	0.08	0.07	0.07	0.07	0.07	1.57	1.40	1.41	1.41	1.40
mtrt	0.04	0.04	0.03	0.05	0.12	473.28	191.92	198.09	212.77	210.98

Table 2: Comparison of compilation time and garbage collection time for the fast configuration using code unloading.

To better understand the differences across variants, we show, in Table 2, the execution time (in seconds) spent in dynamic compilation (left half) and in garbage collection (right half). For most strategies and benchmarks, the compilation time of the unloading system is longer than that of the clean system. This illustrates the recompilation overhead introduced by our unloading mechanism. Variant *On-S* has the largest overhead in most cases because it unloads code more aggressively than others and thus makes more incorrect unloading decisions. Each wrong unloading will incur the overhead of recompilation. However, in many cases, *On-S* also achieves the best garbage collection time savings. Thus, incorrect unloading decisions made by *On-S* are sometimes "correct" from the perspective of memory. The results indicate that though we introduce overhead for recompilation, the garbage collection savings are much greater. As such, we are trading off GC time for less-expensive compilation overhead to improve program performance.

3.3 Adaptive JVM Configuration

In this section, we present our experimental results for the adaptive configuration. We collected data for the five different online unloading strategies described in Section 2.3. For the three sample-based online variants, we handle *hot* methods specially. We annotate

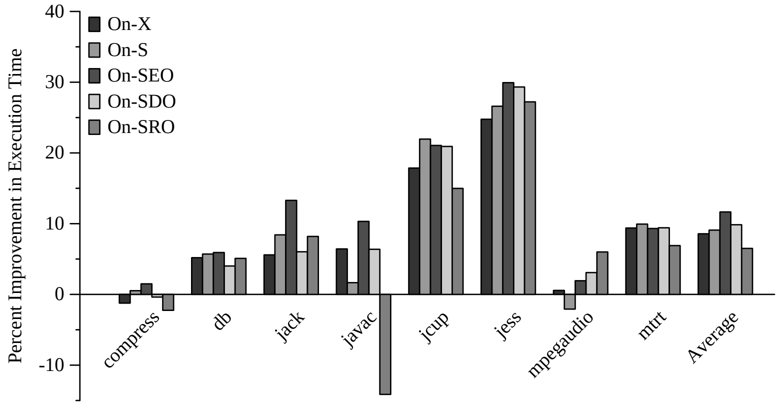


Figure 4: Percent improvement in execution time due to code unloading using the adaptive configuration over the clean system. In each case, we use online profile information ("On-" prefix) to guide unloading decisions. We present five unloading variants: X uses exhaustive profiling information (all others use sampled-profiled information), SEO avoids unloading optimized (hot) methods, SDO delays unloading of hot methods for a single unloading session, SRO recompiles the reloaded hot methods at the optimization level they were previously optimized at (the default is to fast-compile such methods and to re-discover their "hotness").

hot methods given the optimization decisions made by the adaptive system during offline execution of each benchmark. We do not annotate the actual level, we only indicate that a method is hot if it is selected for optimization at any level offline.

Since the fast configuration indicates that online profiling achieves better performance than offline, we report results for our online profiling strategies only, for the adaptive configuration. The five unloading strategies that we evaluate in this section are *On-X*, *On-S*, *On-SEO*, *On-SDO*, and *On-SRO*. Each has a prefix of "On-" to indicate the use of online profile information. *On-X* uses the exhaustive profile strategy and all others use the sample-based (indicated by the use of *S*) approach. *On-X* and *On-S* use the default unloading strategy: if a method is unmarked when an unloading session begins, then it is unloaded. All other strategies (*On-SEO*, *On-SDO*, and *On-SRO*) are variants on the sample-based profile strategy. *On-SEO* regards optimized methods as hot and *excludes* them from the unloading candidates list. *On-SDO delays* unloading of *optimized* (hot) methods for two unloading sessions (the default is that they are considered for unloading during the next unloading session).*On-SRO* attempts to avoid the overhead of adaptive learning (the time for the adaptive system to discover that the method is hot for a second time when it is reloaded) by optimizing the method at the level it was at when it was unloaded (if any).

Figure 4 shows the percent improvement enabled by our code unloading framework over the clean, reference, system using the adaptive optimization configuration. Similar to the results using the fast configuration, the results for the adaptive configuration indicate that sample-based profiling (*On-S*) is able to reduce execution time to a greater degree than the exhaustive profiling strategy (*On-X*). On average, *On-X* and *On-S* have 8.57% and 9.09% performance improvements, respectively, over the clean system for the programs studied. javac and mpegaudio are two cases in which *On-X* performs better than *On-S*. This is because the *On-S* strategy is too aggressive and unloads optimized code that is later used again. Upon reloading, these methods are fast-compiled and instrumented. The adaptive system later finds that the methods are (still) hot, and optimizes them. These two sources of overhead (extra compilation and unoptimized execution during learning period) significantly degrade the performance of the *On-S* strategy.

The sample-based variants improve the situation, however. *On-SEO* (exclude hot methods from unloading candidate list) enables the largest improvement in most cases (11.66% on average). This indicates that when we mistakenly unload optimized methods, the penalty is high. In addition, it means that hot methods remain hot throughout the execution of the program. *On-SDO* (delay unloading of hot methods until the next unloading session) does not perform as well as *On-SEO* because the overhead of inaccurate unloading is still too high. *On-SRO* (re-optimized hot methods at their previous level upon reload) performs worse than *On-SDO* in most cases and can significantly degrade performance (see javac and compress). The degradation is due to the re-optimization overhead incurred: Either optimization is unnecessary since the method is no longer hot, or the cost of optimization is not amortized by the method's remaining execution time. On average, *On-SDO* has 9.85% improvements and *On-SRO* has 6.50% improvements over the clean system.

Tables 3 and 4 the breakdown of compilation and garbage collection time, respectively, with and without unloading. *clean* is the reference system; the other columns show the effect of each of the adaptive unloading varients. The data indicates that compilation time is more expensive than fast compilation. However, GC time still outweighs this overhead since only hot methods are optimized. As such, each variant significantly reduces GC overhead which enables the overall performance improvements shown in Figure 4. *On-SEO* introduces the least amount of compilation overhead for reloading, in most cases.

Benchmark	Compilation time (s)					
	clean	On-X	On-S	On-SEO	On-SDO	On-SRO
compress	0.37	0.34	0.46	0.39	0.42	0.57
db	0.34	0.38	0.43	0.42	0.52	0.38
jack	0.34	0.91	1.60	1.31	1.61	1.63
javac	0.58	0.80	2.32	1.90	1.97	2.74
jcup	0.07	0.08	0.10	0.10	0.09	0.09
jess	1.09	1.37	1.36	1.09	0.97	1.60
mpegaudio	0.86	0.92	0.98	1.09	1.11	1.16
mtrt	1.26	1.27	1.24	1.00	1.34	1.31

Table 3: Comparison of compilation time for the adaptive configuration using code unloading.

Benchmark	Garbage collection time (s)					
	clean	On-X	On-S	On-SEO	On-SDO	On-SRO
compress	10.72	10.60	11.07	10.62	11.04	11.81
db	52.02	45.96	45.78	45.61	47.76	46.14
jack	166.24	154.18	149.03	140.71	153.41	149.85
javac	168.49	154.38	161.70	148.39	154.56	193.35
jcup	7.66	6.09	5.73	5.78	5.82	6.32
jess	190.24	136.19	132.36	126.44	127.28	131.53
mpegaudio	12.39	12.28	12.17	14.61	14.33	14.53
mtrt	74.19	63.75	63.33	66.03	64.94	66.53

Table 4: Comparison of garbage collection time for the adaptive configuration using code unloading.

3.4 Energy Savings Due To Code Unloading

Memory access is one of the largest consumers of battery power. In addition, execution of Java programs consumes significant amounts of memory [34, 14]. Since the performance improvements due to code unloading result from reductions in garbage collection overhead, and since the GC accesses significant amounts of memory, code unloading should also reduced power consumption. To evaluate this hypothesis, we performed a series experiments that measure the impact of code unloading on battery life.

Using the same infrastructure (a Toshiba Protege 2000 laptop with a 750Mhz PIII Mobile processor running Debian Linux v2.4.20), we collected Advanced Power Management(APM) data [3] exported via the /proc operating system interface. We charged the laptop battery completely then executed a script that measured APM data periodically. When the battery level reached 90%, the script invoked a JikesRVM configuration using a benchmark as input. The script then waited for the program to terminate then measured the APM data again. We repeatedly performed the experiment for each benchmark and JikesRVM configuration. We measured the percent of the battery consumed by both the clean JikesRVM system and the sample-based online strategy using the fast configuration.

The percent battery consumption due to program execution with and without code un-

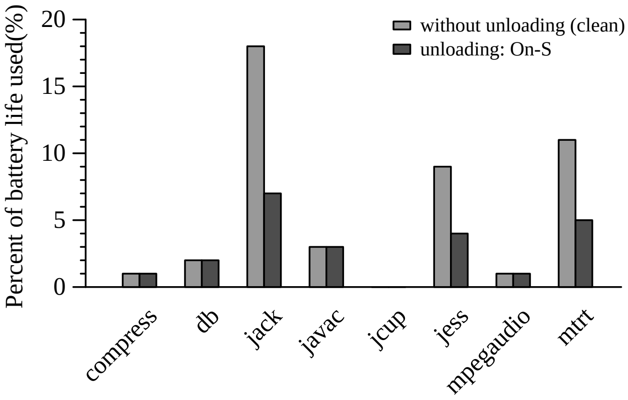


Figure 5: Percent of battery life used for a single run of each benchmark program using the fast configuration. Data for the clean system as well as the sample-based, online, unloading system is shown.

loading is shown in Figure 5. The improvement in performance does translate to a reduction in battery consumption. As we saw in the prior figures, improvements for jack, jess, and mtrt are significant. These benchmarks reduce consumption of battery life by more than half. All other benchmarks have the same consumption behavior with and without unloading. We believe that there are minor savings for these benchmarks. However we are unable to measure the difference since APM data reports measurements at a very coarse grain.

4 Related Work

There are many research efforts that have focused on code size reduction for restricted resource environments. The technique that is most related to our work is *code pitching* used in Microsoft .NET Compact Framework [29]. The virtual machine for this framework uses a JIT compiler to translate intermediate code (CIL) into native code without optimization. When the total size of the code heap exceeds a maximum, the entire contents of the buffer are "pitched"(discarded) [28, 29]. This form of unloading does not consider any dynamic information (except total code size) to determine what and when to unload. As such, it will perform unloading unnecessarily when resources are not constrained. In addition, it discards all code (even hot methods) requiring recompilation of all methods that are invoked in the future.

The HotSpot technology of Sun's Java virtual machine [15, 11] limits the size of compiled code by only compiling the hottest methods and interpreting all other methods. In contrast to their “never cache cold methods” strategy (which imposes large re-interpretation overheads), our framework enables a more flexible code caching strategy which can adapt to system resource status: whether and how long a method's code is cached is dynamically decided by the code unloader according to the runtime information and system memory status.

Other mechanisms for code size reduction that have been pursued by other researchers include compression. Compression is a compact encoding of data to reduce storage and transfer requirements. There are a number of general compression techniques for Java bytecode [26, 7, 10]. Some of which consider program behavior to guide compression. In [13], the authors apply data compression techniques to compress infrequently native code to reduce the code size. They use offline profiling information to select the methods to be compressed. Another type of code compression technique is code abstraction. In [22], the authors use procedural abstraction to save code size. This technique identifies repeated sets of instructions and replaces them with a procedural call. The authors of [10], implement a similar technique in which they factorize the common bytecode instruction sequence into macros. All such compression techniques require some form of online decompression or decoding and must be very efficient since they consumes valuable resources. Java bytecode optimization [33, 16, 19, 20] can be also thought of as a form of compression since it reduces redundancy in the program data and instructions. All these techniques for code size reduction can be considered bytecode preprocessors; while our work tries to remove dead or infrequently used code at runtime. All such bytecode compression techniques are complementary to our unloading system since we only consider native code as unloading candidates.

The energy behavior of JVMs and Java applications is characterized in [34, 14]. Both works confirm that an interpreter consumes significantly more energy than in the JIT compiler mode and verified that the JIT approach is a better alternative for embedded JVMs from both performance and energy perspectives. In [8], G. Chen et al. focused on the energy impact of various parameters of a mark/sweep garbage collector in a multi-bank memory architecture and proposed a GC-controlled leakage energy optimization technique which shuts off memory banks that do not hold live data. Our work is complementary to each of this and similar power-optimizing techniques.

5 Conclusions

In this paper, we first investigate the opportunity of unloading compiled code in JIT-based JVMs for mobile and embedded devices . We find that, for most benchmarks, over 70% (in size) of code is dead after the initial 10% of execution time, and over 60% of methods are active for less than 5% of their own lifetime. These results indicate that there are many opportunities that we can exploit using code unloading to improve overall performance. We next propose a code unloading framework (that can be integrated into any JVM that implements dynamic compilation) that exploits this performance potential. In our framework, a code unloader uses information about the system memory status to determine when to initiate an unloading and utilizes both online and offline profile to select methods for unloading.

We implement our code unloading framework in IBM JikesRVM and investigate a number of different unloading strategies. Through experimentation using both unoptimized JIT configuration and an adaptive optimization configuration, we find that by adaptively unloading the dead and infrequently used code, we can not only reduce the memory requirements of the JVM runtime but also achieve significant performance improvements. Overall, our best strategies enable an average performance improvement by 22.32% for the unoptimized configuration and 11.66% for the adaptive configuration.

References

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast,Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 280–290, May 1998.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[3] Advanced Power Management BIOS Interface Specification Revision 1.2. `http://www.microsoft.com/ whdc/hwdev/archive/BUSBIOS/amp_12.mspx`.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[5] D. Box. *Essential .NET, Volume I: The Common Language Runtime.* Addison Wesley Professional, Nov. 2002.

[6] G. Bracha, J. Gosling, B. Joy, and G. Steel. *The Java Language Specification*. Addison Wesley, second edition, June 2000.

[7] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: An efficient compressed format for java archive files. In *Proceedings of CASCON'98, Toronto, Ontario*, pages 294–302, Nov. 1998.

[8] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning Garbage Collection in an Embedded Java Environment. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, Feb. 2002.

[9] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[10] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[11] The cldc hotspot(tm) implementation virtual machine. `http://web2.java.sun.com/products/cldc/wp/CLDC_HotSpot_WhitePaper.pdf`.

[12] H.-P. Company. ChaiVM. `http://www.chai.hp.com`.

[13] S. Debray and W. Evans. Profile-guided code compression. In *Proceeding of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 95–105, 2002.

[14] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[15] The Java HotSpot Virtual Machine, Technical White Paper. `http://java.sun.com/` products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.ps.

[16] Jarg bytecode optimizer and classfile shrinker. `http://jarg.sourceforge.net/`.

[17] CUP Parser Generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[18] IBM Jikes Research Virtual Machine (RVM). `http://www-124.ibm.com/developerworks/oss/jikesrvm`.

[19] Jode bytecode optimizer. `http://jode.sourceforge.net/`.

[20] JoGa : Java Optimizer with Global Analysis. `http://www.nq4.de`.

[21] Kaffe – An opensource Java virtual machine. `http://www.transvirtual.com/kaffe.htm`.

[22] D. Kim and H. J. Lee. Iterative procedural abstraction for code size reduction. In *International Conference on Compiler, Architecture and Synthesis for Embedded Systems (CASES)*, Grenoble, France, Oct. 2002.

[23] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, June 2001.

[24] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. *Software-Practice and Experience*, 31(8):717–738, 2001.

[25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, second edition, Apr. 1999.

[26] W. Pugh. Compressing Java Class Files. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 247–258, May 1999.

[27] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[28] Rotor - the shared source cli. `http://research.microsoft.com/programs/europe/rotor/default.aspx`.

[29] D. Stutz, T. Neward, and G. Dhilling. *Shared Source CLI Essentials*, page 251. O'Reilly Associates, Inc., Mar. 2003.

[30] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[31] I. Sun Microsystems. White paper: Java(TM) 2 Platform Micro Edition(J2ME(TM)) Technology for Creating Mobile Devices, May 2000. `http://java.sun.com/products/cldc/wp/KVMwp.pdf`.

[32] D. Takahashi. Java chips make a comeback. *Red Herring*, July 2001.

[33] F. Tip, C. Laffra, and P. F. Sweeney. Partial Experience with an Application Extractor for Java. In *Proceeding of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, pages 292–305. ACM Press, Nov. 1999.

[34] N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam, and M. J. Irwin. Energy Characterization of Java Applications from a Memory Perspective. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, Apr. 2001.

[35] A. Wigley, S. Wheelwright, R. Burbidge, R. MacLeod, and M. Sutton. *Microsoft .NET Compact Framework (Core Reference).* Microsoft Press, Jan. 2003.

A Other experimental results

This appendix shows some results collected under the same experimental environment that have not been put into the paper, including the impact of code unloading on memory footprint and the adaptation of the code unloading framework, etc.

A.1 Memory footprint reduction of Code Unloading

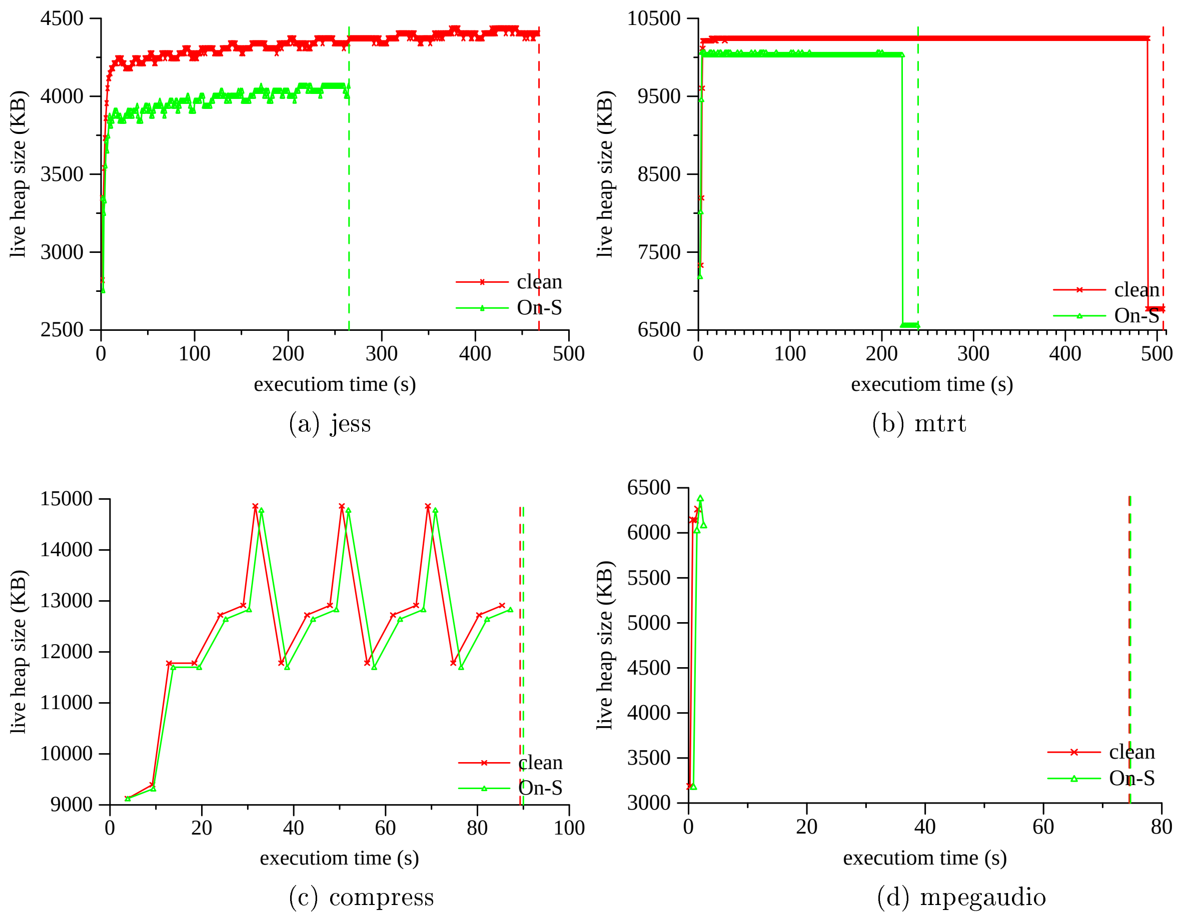
The most direct impacts of code unloading should be the reduction of the memory footprint of the Java virtual machine. Table 5 gives the average *live heap size* (number of bytes (in KB) contained in the heap following each garbage collection) for each variants of the fast configuration. The four variants have very similar results: for most benchmarks, the average *live heap size* has a reduction of over 100KB. Note that this reduction is not the size of the code that is unloaded. It only indicates how the memory usage reacts to the unloading mechanism. Our further analysis into these benchmarks revealed that the reduction of average heap residency is highly correlated with the program's memory usage pattern.

	Average Live Heap Size (KB)				
Benchmarks	clean	Off-GCBased	Off-OnStack	On-X	On-S
compress	12475	12367	12363	12408	12399
db	11648	11506	11489	11521	11522
jack	3646	3494	3512	3455	3375
javac	12744	12625	12678	12635	12538
jcup	3040	2897	2860	2990	2901
jess	4330	3990	4001	3991	3971
mpegaudio	5433	5357	5337	5411	5439
mtrt	10162	9844	9848	9873	9864

Table 5: Average live heap size for the fast configuration. Clean is the default, reference system. The other columns show the impact of using code unloading using online and offline profile information.

To better understand how the memory usages of programs react to the unloading mechanism, Figure 6 shows the *heap residency* curves for four typical benchmarks. The x-axis is the execution time of each benchmark in seconds. The y-axis shows is heap residency in kilobytes. A point on a curve indicates the amount of live data in the heap at the end of a garbage collection cycle. The vertical dashed lines show the program termination time. Since the curves of all four unloading variants are similar, we show only the curves of the clean version and the *On-S* variant. *jess* and *mtrt* (Figure 6(a) and (b)) are representatives of the benchmarks with a significant reduction of memory footprint. Totally, there are 690 and 656 garbage collection cycles during the execution of the clean version for *jess* and *mtrt* respectively, which indicates very frequent memory allocation requirements of these two benchmarks. At all points, *On-S* is lower than the clean version with an apparent gap. The dashed lines of these two figures show that the programs terminate much earlier when code unloading is used.

Figures 6(c) and (d) show the curves of two benchmarks for which unloading has little benefit. For *compress*, the *On-S* curve is only slightly below the clean version. This is due to its small code size and large total heap size. Furthermore, only 18 garbage collection cycles are triggered during the execution of *compress*, which means *compress* requires much less frequent memory allocations than that of *jess* and *mtrt*. *mpegaudio* is a more extreme example: it uses very little heap data and there are only 4 garbage collection cycles that all occur at program startup time. A small number of cycles provides our system with little opportunity for unloading. However, it also indicates



(a) jess (b) mtrt

(c) compress (d) mpegaudio

Figure 6: Examples of live heap size patterns for the fast configuration

looser memory constraints and smaller necessity of code unloading. Our unloading system is able to identify such situation and as such adapt the amount of overhead it introduces. The dashed lines in these two figures show that the *On-S* version terminates at almost same time as clean version does for both benchmarks.

The code unloading techniques have similar impacts on the memory footprints in the adaptive configuration. In Table 6, we list the average *live heap size* data of all benchmarks using different variants for the adaptive configuration. It is in the same format as Table 5. For almost all benchmarks, the reduction of average live heap size is more than $100KB$. Again, the best case is *jess*, which has more than $400KB$ reduction for all our unloading variants. Note that, *compress* and *mpegaudio* also have big reduction of average live heap size this time because the adaptive system aggravates the memory constraints of these two applications, which triggers more aggressive code unloading. The live heap size curves of benchmarks are not shown here because they are similar to those in the fast configuration except that the gaps between clean version and *On-S* version are larger due to larger code size produced by the adaptive system.

A.2 Adaptation to the Memory Status

As we describe in Section 2.1, our code unloading framework attempts to find the best balance between "always caching all compiled code" and "never caching any code" according to the run-

Benchmarks	Average Live Heap Size (KB)					
	clean	On-X	On-S	On-SEO	On-SDO	On-SRO
compress	14376	14065	14310	14135	14196	14492
db	13316	13273	13280	13269	13261	13146
jack	5334	5239	5120	5118	5127	5116
javac	14521	14368	14274	14254	14262	14435
jcup	4726	4609	4526	4527	4528	4584
jess	6179	5771	5749	5702	5702	5709
mpegaudio	8262	8156	8134	8147	8115	8112
mtrt	11610	11320	11332	11548	11321	11367

Table 6: Average live heap size for the adaptive configuration. Clean is the default, reference system. The other columns show the impact of using code unloading using exhaustive and different sample profiling strategies.

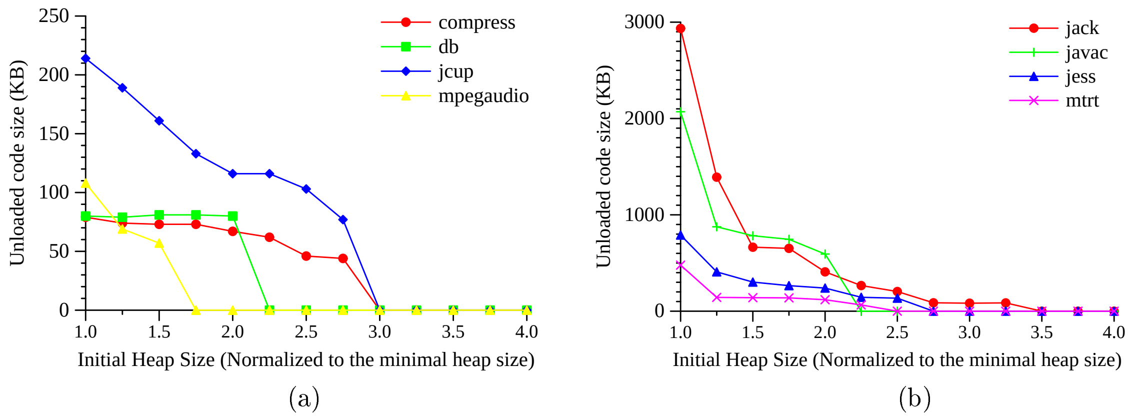


Figure 7: Adaptation of code unloading to memory criticalness

time memory status. When memory constraints are loose, caching more compiled code improves performance by avoiding the recompilation overhead. However, when memory constraints become critical, caching less compiled code will gain better performance by reusing the memory occupied by the compiled code to reduce the garbage collection time. Figure 7 shows how aggressive the code unloader of the *On-S* version unloads compiled code at different initial heap sizes. The x-axis is the initial heap size we give in the command-line to run the application (normalized to the minimal heap size used in Table 1 column 3). The y-axis shows the total code size unloaded by the code unloader during the execution time. To be clearer, we split all benchmarks to two figures with different y-axis scales. Figure 7(a) contains curves for compress, db, jcup and mpegaudio and Figure 7(b) is for jack, javac, jess and mtrt. For all benchmarks, the unloaded code size decreases when the initial heap size increases, which indicates that the code unloader does less aggressive unloading when the memory becomes less critical. Most of curves drop to zero before the initial heap size grows to 3 times of the minimal heap size, where the code unloader decides to cache all compiled code due to the loose memory constraints. Benchmarks in Figure 7(a) represent those applications with few garbage collocation cycles during executions even at the minimal heap sizes, in which case the code unloader only tries to unload dead code at program startup time if the heap residency is higher than a certain threshold. Since these benchmarks do not have critical memory constraints, increasing heap size does not affect the unloading decision dramatically. By contrary, the benchmarks in Figure 7(b) have very severe memory constraints under the minimal heap size. However, such memory constraints become looser quickly when the initial heap size increases. As such, the code unloader performs very aggressive unloading strategies at the most critical memory level and decreases the aggressiveness of unloading dramatically with larger initial heap size. In short, Figure 7 indicates that our code unloading framework is able to adapt the aggressiveness of unloading to the real memory status.