Profile-driven Code Unloading for Resource-Constrained JVMs

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

Abstract

Java virtual machines (JVMs) have become increasingly popular for execution of a wide range of applications on mobile and embedded devices. Most JVMs for such devices execute programs using interpretation. However, JVMs that use dynamic compilation have been shown to enable significant performance improvements. A disadvantage of a compile-only approach in resource-constrained environments is that it uses more memory than interpretation to store compiled code for reuse. In this paper, we address this limitation with techniques that attempt to automatically unload dead or infrequently used native code to reduce the memory footprint of a compile-only JVM. We describe a number of profile-based strategies that identify unloading candidates. Our empirical evaluation, using a number of common benchmarks, indicates that dynamic code unloading can reduce code size significantly with negligible overhead. When memory is highly constrained, this reduction translates into significant execution time benefits for the benchmarks, JVM, and JVM configuration that we investigated.

1 Introduction

Java virtual machines (JVMs) [13] 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 [18]. This wide-spread use of Java for embedded systems is the result of the increased capability of mobile devices, the ease of program development using the Java language [3], 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 [17, 6, 10] perform translation using interpretation since interpreters are simple to implement, impose no perceivable interruption, and do not require that native code be stored during execution. However, interpreted program 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, e.g., CPU, memory, battery, etc [19, 8].

To overcome the limitations of JVM interpretation, next-generation JVMs [5, 16, 1] employ just-in-time (JIT), i.e., dynamic, compilation. The resulting execution performance is higher than if interpreted due to improved code quality (that results from translation of multiple instructions at once, exposing optimization opportunities), and to the reuse enabled by storing native code.

The latter (native code caching) can be a drawback in a resource-constrained environment since native code is much larger than its bytecode equivalent. For example, we found, through experimentation using the SpecJVM benchmark suite [14] and two different JVMs (JikesRVM [1] and the Kaffe embedded JVM [10]), that IA32 native code is 6-8 times larger than bytecode. In addition, ARM code is at least two times larger than IA32 code. As such, even for 16 bit ISAs, e.g., ARM/THUMB, which can reduce ARM code size by almost half, the size of compiled native code is significantly larger than its corresponding bytecode.

Our initial experiments also show that, on average, 61% of code bodies in the benchmarks studied, become dead after startup period. In addition, methods that are live after startup commonly have very short lifetimes. Thus, compile-only JVMs require a larger memory footprint to cache native code even though, in many cases, a majority of code is used infrequently ("cold") or goes unused after some initial period.

To reduce the memory requirements of compile-only JVMs, we developed a JVM extension that dynamically unloads native code bodies. When an unloaded method is later reused, our system re-compiles it from bytecode as is done currently for its initial method. As such, our dynamic code unloading trades off reduced memory pressure for recompilation overhead.

Key to the efficacy of our dynamic code unloading system is the selection of unloading candidates. Since re-compilation overhead is introduced when unloaded code is later executed, we must accurately identify code that is dead or cold to avoid introducing unnecessary overhead. Our techniques enable this by using past program execution behavior, i.e., profile information, to identify methods that can be unloaded. As a result, our system is able to reduce the memory footprint of the JVM, free memory for other uses, and avoid unnecessary recompilation overhead. Our empirical evaluation indicates that our techniques for *profile-driven code unloading* significantly reduce code size and introduce negligible overhead.

In the following sections, we describe the code-unloading strategies as well as their implementation issues in an open source JVM. We then evaluate the impact of code unloading on both code size and performance in Section 3. Finally, we discuss related work (Section 4) and conclude in Section 5.

2 Code Unloading Strategies and Implementation

Compile-only JVMs are able to produce code that uses the resources of resource-constricted devices more efficiently than can interpretation-based JVMs since compiled code bodies can have significantly higher code quality and can be reused. Code reuse, however, requires that large native code bodies be cached in memory; in interpretation-based systems, no such caching is performed. As such, compile-only JVMs can impose a large memory footprint, which on memory-constrained devices, can negate any benefit achieved by the use of compilation (over interpretation). The goal of our work is to develop techniques that reduce the memory requirements of compile-only JVMs for resource-restricted devices.

To this end, we implemented *dynamic code unloading* within a compile-only JVM to periodically unload method code bodies. If the unloaded code is later reused, the system automatically and transparently re-compiles the bytecode of methods to produce new native code bodies. To limit the recompilation overhead introduced by our approach, our techniques attempt to identify methods that are *unlikely* to be invoked in the future. To enable this, we use past method invocation behavior as an indication of future behavior, i.e., a method can be unloaded if it has not been invoked recently. We investigated four strategies for collecting past behavior for use in the identification of unloading candidates:

- Online eXhaustive profiling (OnX)
- Online Sample-based profiling (OnS)
- Offline profiling (Off)
- No Profiling (NP)

In the first strategy (OnX), we modified the compiler to instrument methods. We keep a mark bit for each method compiled; the mark bit indicates that the method has been used recently. Each time a method returns, the instrumented code sets its mark bit to 1. When unloading occurs, unmarked methods will be unloaded and all marked bits will be reset. This mechanism unifies the unloading of dead and infrequently invoked methods. OnX guarantees that every method that has been invoked since the last unloading session will have its mark bit set. As such, only recently unused methods will be unloaded. However, exhaustive profiles introduce profiling overhead for every single method invocation.

To address this limitation, we considered a sample-based profile (OnS) approach as our second strategy. For this strategy, the compiler inserts no instrumentation. Instead, the JVM sets the mark bits of the **two** methods on the top of invocation stacks of application threads for every thread switch (which occurs approximately every 10 ms in our prototype JVM). We determined this value (two) empirically in attempt to balance the tradeoff between the significant overhead required for complete stack scans and incorrectly unloading used methods.

In our next strategy, we investigated the efficacy of having perfect knowledge about method lifetimes. We gather the total invocation count offline for each method. We then annotated this value in the class file as a method attribute for use by the JVM during program execution. At runtime, we use online profiling to identify when the last invocation of a method occurs; at which time, we mark the method for unloading. We refer to this strategy as Off. The advantage of this strategy is that it will not introduce any recompilation overhead since we unload a method only when it becomes dead. However, this strategy is based on offline profiling, which imposes extra burden on users and suffers the cross-input problem.

In the final strategy, called NP for "no profiling", we simply unload all methods that are not currently on the runtime stack when unloading occurs. This strategy is the most aggressive form of code unloading. The advantages of this strategy are that it is easy to implement and has no profiling overhead. However, it might unload too many methods that would be invoked in the future, which may introduce significant recompilation overhead.

Besides selecting unloading candidates, the JVM must decide when to trigger the unloading session. In our currently implementation, we use a *Timer triggered (TM)* strategy. That is, we unload code at fixed, periodic intervals. To implement this strategy efficiently, we approximate time using a thread-switch count. In our prototype JVM, thread switching occurs at approximately every 10 ms. However, garbage collection (GC) in this JVM is stop-the-world which means that our thread switch counters will be highly inaccurate when heap memory is critical and GC occurs frequently. To solve this problem, we modified the GC system to update our thread switch count at the end of every GC cycle according to the time spent in each cycle.

When a method code body is selected for unloading, its address is replaced with that of a recompilation stub. In our prototype JVM, compiled code is stored in the heap that is managed by a garbage collector. Thus, 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. For JVMs that do not store compiled code in the application heap, we can use our unloading strategies to explicitly free the memory. We are studying the efficacy of this latter approach as part of future work.

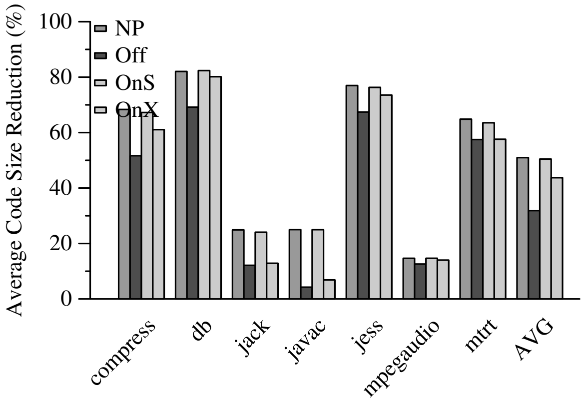
The recompilation stub is similar to the compilation stub used for lazy, dynamic, compilation [11, 1] but contains additional information that guides recompilation if reloading should occur. If the method is ever invoked again, the recompilation stub causes it to be compiled again prior to execution.

3 Experimental Evaluation

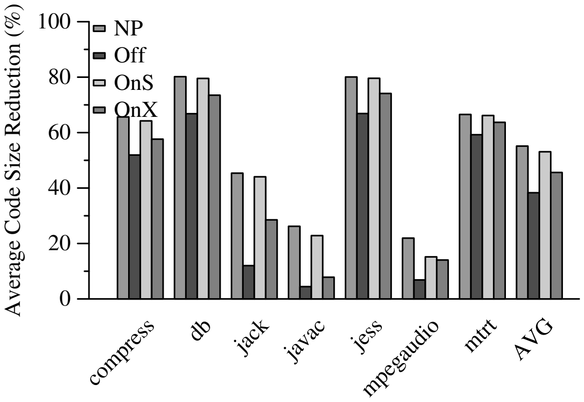
We implemented our code unloading strategies in JikesRVM [1], an open-source JVM from IBM Research (x86 version 2.2.1). JikesRVM has a *baseline* configuration, in which all methods invoked by an executing program are compiled with a very efficient, non-optimizing compiler. This configuration is similar to that of the Microsoft Common Language Runtime (CLR) of .NET Compact Framework [21] and other embedded JVMs that use a simple JIT for bytecode method translation [12, 10]. Even though this JVM is not intended for embedded systems, by using its *baseline* configuration and limiting its working memory to be less than 32 MB, we believe that our results using JikesRVM as a prototype lend insight into the potential benefits of code unloading in compile-only JVMs for resource-restricted environments.

We implemented our techniques in JikesRVM version 2.2.1 for x86 with the default garbage collector (a semi-space copying collector). We plan to investigate the efficacy of our techniques using other garbage collectors, e.g., mark/sweep, generational, and hybrid collectors, as part of future work.

We gathered our performance data by repeatedly executing single runs of each of the SpecJVM benchmarks [14] using input size 100 on a dedicated Toshiba Protege 2000 laptop (750 MHZ PIII Mobile) running Debian Linux (kernel v2.4.20). The native code sizes of these benchmarks range from 98 KB to 469 KB. We empirically evaluated our techniques using two memory configurations: MIN and 32MB. MIN means



(a) 32MB



(b) MIN

Figure 1: Comparison of code size reduction for the four unloading strategies

the minimum heap size that is required for each benchmark to run to completion (identified empirically). MIN is used to represent the situation that memory is highly constrained, while 32MB is used to represent the situation that memory is not highly constrained. The timer to trigger code unloading is set to be 10 s. In all of our results, we show the improvement percentage over the unmodified system, which is referred as *clean*.

3.1 Impact on Memory Footprint

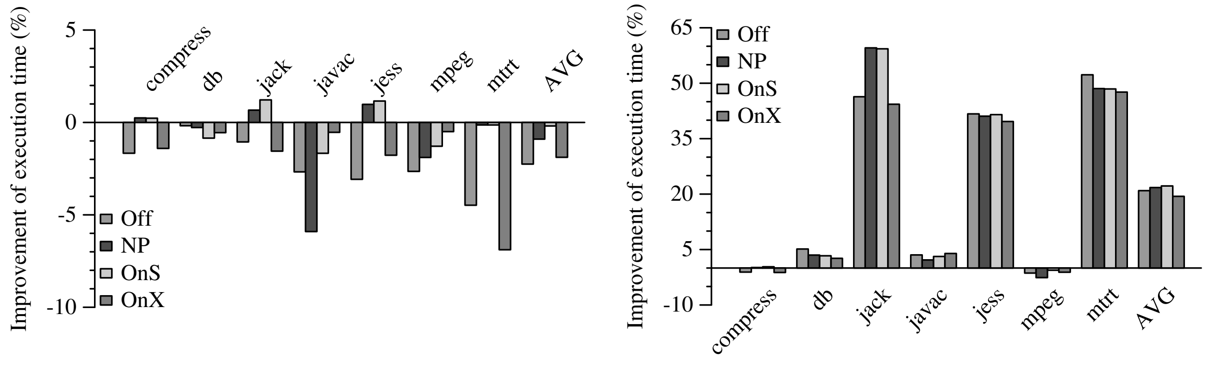
Figure 1 (a) and (b) show the impact of various unloading strategies on average code size. The y-axis in both graphs in the figure is the percent reduction of code size (the amount of code stored in memory) over a clean version of the system. In both memory configurations, MIN and 32MB, the order in which the techniques perform (from worst to best) is: *Off*, *OnX*, *OnS*, *NP*. Strategy *Off* does not unload a method until it becomes dead. Thus, it is the least aggressive strategy. While strategy *NP* always discards all compiled methods except those on the runtime stack during an unloading session. As such, it achieves the largest reduction in average code size. Online exhaustive profiling is more accurate than sample-based profiling in capturing recently invoked methods, and thus, it unloads fewer methods than *OnS*. On average, the average code size reduction, when memory is highly constrained, for each of the strategies, is 38% for Off, 46% for OnX, 53% for OnS, and 55% for NP. When memory is not constrained, the reduction is 32% for Off, 44% for OnX, 50% for OnS, and 51% for NP. In summary, all unloading strategies enable significant code size reduction.

3.2 Impact on Execution Performance

Reducing code size is not our only concern; if it were, never caching any code would be the best choice. Our goal is to achieve the best balance between memory footprint and execution performance. Figure 2 (a) and (b) show the impact of our code unloading strategies on the total execution time of benchmarks. The y-axis in both graphs in the figure is the percent improvement (or degradation) over the clean system. Please note that the y-axis scale in each graph is different, for clarity. Total execution time is a balance of recompilation overhead, profiling overhead, and memory management overhead.

Figure 2 (a) indicates that our code unloading strategies introduce negligible overhead when memory is unconstrained. The average performance overhead for this configuration is 2.3% for Off, 0.9% for NP, 0.2% for OnS, and 1.9% for OnX. The overhead of Off and OnX is greater since profile information is gathered for every method invocation. Strategy OnS achieves the best performance since it imposes less profiling overhead and is able to capture method invocation behavior more effectively than the other strategies.

Figure 2 (b) shows the performance improvements due to code unloading when memory is highly-constrained; on average, the reductions are 20.9% for Off, 21.8% for NP, 22.2% for OnS, and 19.4% for OnX. These improvements are in part due to the fact that in JikesRVM, compiled code is stored in a heap that



(a) 32MB

(b) MIN

Figure 2: Comparison of performance impacts of the four unloading strategies

is managed by the garbage collection system. Thus, garbage collection time is part of memory management overhead in this system. Our results indicate that when memory availability is critical, reducing the amount of native code in the system significantly improves performance since less time is spent in GC.

Since, not all JVMs manage native code bodies in a garbage-collected heap, it is unclear whether these execution time improvements will translate to such systems – even though the reductions in code size will be similar. As part of future work, we plan to investigate the execution performance benefits for such JVMs enabled by dynamic code unloading and our profiling strategies.

In summary, across strategies and heap sizes, sample-based profiling achieves the best balance between small memory footprint and recompilation overhead. Strategy NP works well in most cases due to its zero profiling overhead and the low recompilation overhead in our *baseline* compiler configuration. However, if the cost of compilation increases (say for optimization), NP may not be as efficient as the OnS strategy since it blindly unloads native methods regardless of whether they were executed recently.

4 Related Work

Several code cache management techniques have been proposed in the area of dynamic binary translation [2, 7, 15]. The technique most related to our work is *code pitching* used in Microsoft .NET Compact Framework [15]. In this framework, all code in the code buffer is discarded when the buffer exceeds some threshold. A similar strategy is used in the Dynamo [2] dynamic optimizer from HP. In Dynamo, when the cache fills, all fragments are “flushed” to free up space for new traces. Code pitching and cache flushing can both be configured using our *NP* strategy. This simple strategy is chosen by these systems due to ease of implementation, low profiling overhead, or no linking problems. We found, however, that in our target systems (JVMs), where cached code is commonly method-based, selective unloading of code using lightweight profiling techniques can achieve better balance between execution performance and memory use.

Another related area is code size reduction in JVMs for restricted resource environments. The HotSpot JVM from Sun Microsystems [9] limits the size of compiled code by only compiling the hottest methods and interpreting all others. Other work employs *profile-driven deferred* compilation or optimization [4, 20] to avoid generating code for cold spots in the programs. In contrast to their “never cache cold methods” strategy which may impose large re-interpretation overheads, our techniques enable more methods to be compiled, while still keeping code size small. Even in these “never cache cold methods” systems, code unloading techniques can also be used to manage “hot” methods.

5 Conclusions

In this paper, we present a set of techniques that reduce the memory requirements of compile-only JVMs for resource-constrained devices. Since compile-only JVMs cache native code bodies for reuse, they impose memory overhead that interpreter-based systems do not. However, compile-only JVMs produce significantly more efficient code quality and can exploit optimization opportunities. To achieve the benefits from both compile-only JVMs (more efficient use of device resources) and interpretation-based JVMs (reduced memory requirements), we extended a compile-only JVM to enable *dynamic code unloading*.

In our system, the JVM periodically unloads (frees from memory) a subset of the compiled code bodies that it determines are unlikely to be invoked again. If the code is invoked again, it is transparently re-compiled. We investigated a number of strategies that utilize online profiling, offline profiling, and snapshots of the runtime stack to select unloading candidates. Our results indicate that dynamic code unloading reduces the memory requirements of the compile-only JVM and introduces only negligible overhead. Overall, our best strategy, enables an average reduction in code size of 50% when memory is unconstrained and 53% when memory is highly constrained. Moreover, the overhead introduced by this strategy is less than 1%. In a JVM that stores the code in the garbage-collected heap such as the one we studied, these reductions in code size translate into significant performance improvements when memory is highly constrained: 22% on average across the benchmarks that we studied.

References

[1] 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.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

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

[4] D. Bruening and E. Duesterwald. Exploring Optimal Compilation Unit Shapes for an Embedded Just-In-Time Compiler. In *Proceeding of the 2000 ACM Workshop on Feedback-directed and Dynamic Optimization FDDO-3*, December 2000.

[5] 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.

[6] Hewlett-Packard Company. ChaiVM. http://www.chai.hp.com.

[7] K. Ebcioglu, E. R. Altman, M. Gschwind, and S. W. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[8] 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.

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

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

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

[12] Latte - a jit compiler for java. http://latte.snu.ac.kr/publications/.

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

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

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

[16] 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.

[17] Inc. 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.

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

[19] 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)*, April 2001.

[20] J. Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceeding of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, pages 166–179. ACM Press, October 2001.

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