The Design, Implementation, and Evaluation of Adaptive Code Unloading for Resource-Constrained Devices

LINGLI ZHANG, CHANDRA KRINTZ
University of California, Santa Barbara

Java Virtual Machines (JVMs) for resource-constrained devices, e.g., hand-helds and cell phones, commonly employ interpretation for program translation. However, compilers are able to produce significantly better code quality, and hence, use device resources more efficiently than interpreters, since compilers can consider large sections of code concurrently and exploit optimization opportunities. Moreover, compilation-based systems store code for reuse by future invocations obviating the redundant computation required for re-interpretation of repeatedly executed code.

However, code storage required for compilation can increase the memory footprint of the VM significantly. As a result, for devices with limited memory resources, this additional code storage may preclude some programs from executing, significantly increase memory management overhead, and substantially reduce the amount of memory available for use by the application.

To address the limitations of native code storage, we present the design, implementation, and empirical evaluation of a compiled-code management system that can be integrated into any compilation-based JVM. The system unloads compiled code to reduce the memory footprint of the VM. It does so by dynamically identifying and unloading dead or infrequently used code; if the code is later reused, it is recompiled by the system. As such, our system adaptively trades off memory footprint and its associated memory management costs, with recompilation overhead. Our empirical evaluation shows that our code management system significantly reduces the memory requirements of a compile-only JVM, while maintaining the performance benefits enabled by compilation.

We investigate a number of implementation alternatives that use dynamic program behavior and system resource availability to determine **when** to unload as well as **what** code to unload. From our empirical evaluation of these alternatives, we identify a set of strategies that enable significant reductions in the memory overhead required for application code. Our system reduces code size by 36%-62% on average which translates into significant execution time benefits for the benchmarks and JVM configurations that we studied.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: —*code generation, compilers, memory management, run-time environments*

General Terms: Languages, Performance

Additional Key Words and Phrases: code unloading, code-size reduction, JIT, JVM, resource-constrained devices

Authors' Address: Lingli Zhang and Chandra Krintz, Computer Science Department, University of California, Santa Barbara, CA, 93106; email: {lingli_z,ckrintz}@cs.ucsb.edu.

1. INTRODUCTION

Java virtual machines (JVMs) [Lindholm and Yellin 1999] have become exceedingly popular for execution of mobile and embedded device applications. Researchers estimate that there will be over 720 million Java-enabled mobile devices by the year 2005 [Takahashi 2001]. This wide-spread use of Java for embedded systems has resulted from significant advances in device capability as well as from the ease of program development, security, and portability enabled by the Java programming language [Bracha et al. 2000] and its execution environment (JVMs).

To execute a Java program, the JVM translates the code from an architecture-independent format (bytecode) into the native format of the underlying machine. Many JVMs for embedded and mobile devices translate bytecode using interpretation, i.e., instruction-by-instruction conversion of the bytecode [KVM 2000; ChaiVM ; Kaffe 1998]. Such JVMs employ interpretation since it is simple to implement and imposes no perceivable interruption in program execution. In addition, the native code that is executed is not stored; if code is re-executed, it is re-interpreted. The primary disadvantage of using interpretation is that an 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 a result, interpretation wastes significant resources of embedded devices, e.g., CPU, memory, battery, etc.

To overcome the disadvantages imposed by interpretation, some JVMs [Cierniak et al. 2000; Suganuma et al. 2000; Alpern et al. 2000; HotSpot 2001] employ dynamic (Just-In-Time (JIT)) compilation. Programs that are compiled use device resources much more efficiently than if interpreted due to significantly higher code quality. Compilers translate multiple instructions concurrently which exposes optimization opportunities that can be exploited and enables more intelligent selection of efficient native code sequences. Moreover, compilation-based systems store code for future reuse obviating the redundant computation required for re-interpretation of the same code. According to studies in energy behavior of JVMs and Java applications in [Vijaykrishnan et al. 2001; Farkas et al. 2000], JVMs in the interpreter mode consume significantly more energy than in the JIT compiler mode. Therefore, the JIT approach is a better alternative for embedded JVMs from both performance and energy perspectives.

Despite the execution speedup of compiled code over interpreted code and its power efficiency, dynamic compilation is still not widely used in JVMs for resource-constrained environments due to the perceived memory requirements. Dynamic compilation enlarges the JVM memory footprint in three primary ways: The extra code base introduced by the JIT engine, the intermediate data structures generated by the compiler during compilation, and the compiled code stored for reuse. Significant engineering effort and research [Adl-Tabatabai et al. 1998; Krall 1998; Bruening and Duesterwald 2000] have been performed to address the first two problems by making the JIT compiler more lightweight while still enabling generation of high quality code. In this paper, we present a technique that attacks the third issue: reducing the memory requirements introduced by compiled code.

Compiled native code is significantly larger than its bytecode equivalent (we present data that corroborates this statement in Section 2). In resource-constrained

environments, since the total available memory is limited, the memory consumed by these code blocks reduces the amount of memory available to the executing application. This increase in memory pressure can preclude some programs from executing on the device. Moreover, for systems that use garbage collection to manage compiled code, compiled code increases memory management costs, which can be significant when memory is severely constrained. If applications' code and data share the same heap and are managed by the same garbage collector, the impact of compiled code on memory management costs can be even more severe. As a result, dynamic compilation introduces memory overhead for compiled code not imposed by interpreter-only systems which can in turn negate any benefit enabled by code reuse and improved code quality.

To address this drawback introduced by dynamic compilation, we propose code unloading as an alternative to not compiling code. We designed a framework for dynamic and adaptive unloading of compiled code so that more aggressive dynamic compilation can be performed. Our goal with this system is not only to reduce the size of compiled code. If it were, never storing any compiled code would be the best choice. Instead, our goal is to enable performance improvement via dynamic compilation *while* reducing the dynamic memory requirements of the JVM. That is, we seek to adaptively balance not storing any code (as in an interpreter-based JVM) and caching all generated code (as in a compile-only JVM), according to *dynamic memory availability*, i.e., the amount of memory available to the executing application for allocation of data (as opposed to code) over time. We designed the system as a framework that is highly extensible and that can be easily incorporated into any JVMs with JIT compilers (compile-only systems, or those that combine interpretation and compilation [Delsart et al. 2002; CLDC 2003], i.e., selective compilation JVMs) that are intended for resource-constrained devices.

Our code unloading system decides *what* code to unload and *when* unloading should commence. Each of these decisions can be made using a wide range of unloading strategies, each resulting in different tradeoffs between several sources of overhead and benefit. To study these tradeoffs, we used the framework to investigate a number of unloading strategies which employ dynamic feedback from the program and execution environment to identify unloading candidates and to trigger unloading efficiently and transparently.

We implemented and empirically evaluated our code unloading framework and unloading strategies using a high-performance, open source Java Virtual Machine from IBM T. J. Watson Research Center, the Jikes Research Virtual Machine [Alpern et al. 2000] (JikesRVM). Our system re-compiles methods that have been unloaded but are later invoked by the program – as is done for the method's initial invocation using lazy (Just-In-Time) compilation. Our system, therefore, trades off memory management overhead with recompilation overhead.

Our results indicate that by adaptively unloading compiled code, we are able to reduce code size by 36%-62% on average over the lifetime of the programs. Since the system is able to adapt to memory availability, it introduces no overhead when resources are unconstrained. When memory is highly constrained, reductions in code size translate into execution time improvements of 23% on average for the programs and JVM configurations that we studied.

Table I.   Size and behavior of the compiled native code in JVMs.

Bench-marks	Byte code (KB)	ARM Native Kaffe		IA32 Native Kaffe		Jikes		Dead after startup
		KB	(/BC)	KB	(/BC)	KB	(/BC)	KB (Pct.)
compres	12.4	210.8	(17.0x)	96.7	(7.8x)	98.0	(7.9x)	70.8 (72%)
db	14.5	242.2	(16.7x)	114.6	(7.9x)	105.9	(7.3x)	89.2 (85%)
jack	42.4	788.6	(18.6x)	318.0	(7.5x)	284.1	(6.7x)	72.5 (26%)
javac	78.3	1252.8	(16.0x)	555.9	(7.1x)	469.8	(6.0x)	75.9 (16%)
jess	32.9	559.3	(17.0x)	250.0	(7.6x)	223.7	(6.8x)	167.9 (75%)
mpeg	56.6	1386.7	(24.5x)	464.1	(8.2x)	452.8	(8.0x)	357.4 (79%)
mtrt	21.1	N/A		173.0	(8.2x)	160.4	(7.6x)	117. 6 (73%)

With this paper, we first present an empirical analysis on the potential of code unloading for Java programs in a memory-constrained environment (Section 2). We then detail the code unloading framework that we developed to exploit the code unloading opportunities exposed by our compiled-code analysis (Section 3). We use the framework to investigate a number of unloading strategies and use our empirical evaluation of each in combination to identify a set of strategies that enable the most significant reductions in memory overhead (Section 4). We empirically evaluate the efficacy of our techniques using the SpecJVM benchmarks [SpecJVM98] and a range of JVM compilation configurations (Section 6) including compile-only and selective compilation. Finally, we present related work in Section 7 and conclude in Section 8.

2.   CODE UNLOADING OPPORTUNITIES

To investigate the feasibility of native code unloading for resource-constrained JVMs, we initially performed an empirical analysis of its potential. We performed a series of experiments that measured various static and dynamic characteristics of native code, e.g., size, usage statistics, etc.

Table I shows the size in kilobytes (KB) of the bytecode and native code for the SpecJVM benchmark suite [SpecJVM98]. Column 2 is bytecode size and columns 3–5 show the size of native code and the ratio (/BC) of native code size to bytecode size. We gathered this data using two platforms, ARM and IA32, and two JVMs, the JikesRVM [Alpern et al. 2000], and the Kaffe embedded JVM [Kaffe 1998] with jit3. Since JikesRVM does not have back-end for ARM, we show only data for IA32. This data shows that IA32 native code is 6-8 times that of bytecode for both JikesRVM and Kaffe for these programs. ARM code is even larger (16-25 times that of bytecode) since its RISC-based instructions are simpler than the CISC IA32 instructions (which do more work per instruction). Even if the systems uses the compact instruction form, e.g., ARM/THUMB (potentially reducing native code size by half), the size of compiled native code is likely to be much larger than that of the corresponding bytecode.

The final column shows the amount of code that goes unused after program startup (we define startup as the initial 10% of the execution time). Interestingly, a large amount of executed code becomes dead after program startup; this portion of code remains in the systems and consumes precious system memory needlessly.

Code that is not used after startup can be unloaded. In addition, we found that
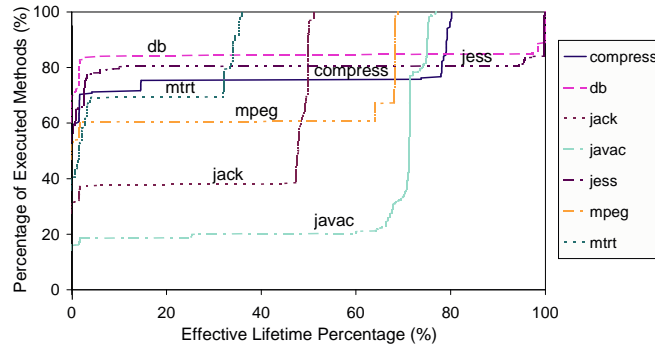
Fig. 1. CDF of effective method lifetime as a percentage of total lifetime. The effective lifetime of a method is the time between its first and last invocations; the total lifetime of a method is the time from its first invocation to the end of the program. A point, (x, y), on a curve indicates that y% of that benchmark's executed methods have an effective lifetime of less than or equal to x% of its total lifetime.

a majority of the code that remains after startup in many benchmarks has short life spans, and thus, can also be considered candidates for unloading. Figure 1 graphs the cumulative distribution functions of *effective lifetime percentage* of methods, i.e., the percentage of the effective lifetime (time between the first and last invocations of a method) over the total method lifetime (time from the method's first invocation to the end of the whole program). This metric is similar to the *trace lifetime* used in [Hazelwood and Smith 2003]. This figure shows that for most of the benchmarks (all but *javac* and *jack*), more than 60% of methods are effectively live for less than 5% of the total time they remain in the system.

We also found that methods with long effective lifetimes commonly are invoked infrequently. For example, method spec.benchmarks._213_javac.ClassPath.<init> has an effective lifetime of 75%, but is only invoked 4 times and executed for only 0.1% of its total effective lifetime. We can consider such methods as unloading candidates also when memory is highly constrained.

In summary, the native code size in JVMs is much larger than that of bytecode. Since native code is stored by compilation-based JVMs for reuse, it consumes precious memory space on resource constrained devices. Moreover, the invocation characteristics of the compiled code presents many opportunities for removing code blocks from the system temporarily or permanently. We developed a JVM code unloading framework to investigate ways to exploit such opportunities.

3. CODE UNLOADING FRAMEWORK

Figure 2 depicts the extensible framework for adaptive code unloading that we developed to relieve memory pressure imposed by complied code in resource-constrained JVMs. The outer box is the boundary of a JVM. Inside this box, the left part is the control-flow of a JVM that employs dynamic and adaptive compilation, which we believe is crucial to achieve high performance with small memory footprint in resource-constrained environments. Adaptive compilation is the process of selectively compiling or recompiling code (guided by online performance measurements) that has been interpreted or compiled previously in an attempt to improve perfor-
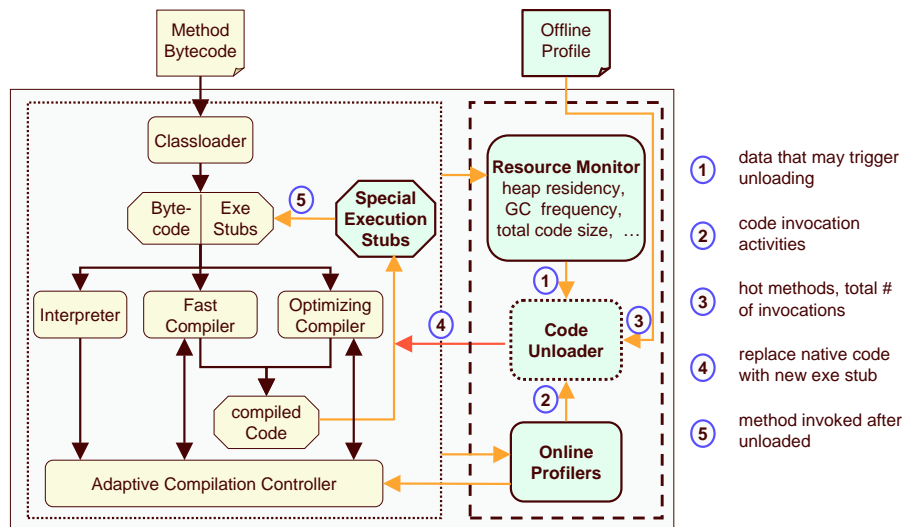
Fig. 2.   Overview of the Adaptive Code Unloading Framework

mance [Arnold et al. 2000a; Cierniak et al. 2000; HotSpot 2001].   The right part of the figure shows our JVM extensions (darkened components) that enable adaptive code unloading.

While programs are executing, the *Resource Monitor* collects information about resource behavior, e.g. heap residency data, garbage collection (GC) invocation frequency, and native code size, etc. The online and offline *profilers* collect information about application behavior, such as hot methods and invocation activity of each method. The code unloading system can share the profilers with the adaptive compilation system if possible to reduce overhead. The *Code Unloader* takes information from these components, analyzes the cost and benefit of unloading, and decides when code unloading should commence and which methods to unload. As such, the framework *dynamically and automatically* identifies dead or infrequently used native code bodies and unloads them from the system to relieve memory pressure *whenever necessary.*

Once the unloader selects a method, it replaces its code entry in the dispatching table with a special stub, which we refer to as the *execution stub*. This stub is similar to the mechanism used by the JVMs to enable lazy, Just-In-Time compilation [Alpern et al. 2000; Krintz et al. 2001]. However, we add additional information to specify how to execute the method, e.g., interpreting, fast compiling without optimization, or compiling at a particular optimization level, if it returns to the system after being unloaded.

The system reclaims the native code block of an unloaded method during the next garbage collection cycle since it is no longer reachable by the program. If the executing program invokes a method that has been unloaded, the execution stub will invoke the interpreter or an appropriate compiler to re-translate the method. If the method is compiled, its address (that of the stub) is replaced with that of the newly compiled method in the dispatching table. Future invocations of the method

by the program execute the compiled method directly through the table entry.

Since the unloader and other framework components must operate *while* the program is executing, we designed the system to be very light-weight. Moreover, the framework implements a flexible and extensible foundation via a well-defined interface that we (and others) can use to investigate, implement, and empirically evaluate various code unloading strategies.

We implemented the framework as an extension to the open-source Jikes Research Virtual Machine (JikesRVM) [Alpern et al. 2000]. JikesRVM is a compiler-only JVM, and thus, the special execution stub either fast compiles the method, or optimizes the method at certain level directly. No interpretation is performed. This implementation can be easily extended to handle interpretation in the execution stubs if an interpreter is included in the JVM. We then used the resulting system to investigate four strategies that identify unloading candidates and four strategies that trigger unloading. We detail each of these strategies in the following sections.

4.   UNLOADING STRATEGIES

There are four primary decisions that any dynamic code unloading system must make: (1) When unloading should be triggered; (2) How unloading candidates should be identified and selected; (3) Whether optimized code should be handled differently from unoptimized code or not; and (4) How the system should record the profile information used in the decision process. There are a number of possible answers to these questions; each of which leads to a different tradeoff between the JVM memory footprint and execution performance. We investigated a number of strategies that attempt to answer these questions in an effort to identify the best-performing combination of design decisions.

4.1   Triggering an Unloading Event

We first investigated various ways in which we can trigger unloading. That is, we implemented strategies that decide *when* to unload. We considered four different triggers: Maximum invocation count, a timer, GC frequency, and code cache size.

The first strategy is called *Maximum Call Times (MCT)* triggered. We use offline profiling to collect the total invocation times for each method. Then the code unloader uses this information to trigger unloading of methods following completion of the last invocation of each. The system records the invocation times upon each method return. This strategy is not adaptive to execution behavior but guarantees all methods are unloaded when the program is finished with them. The strategy introduces no additional compilation overhead since unloaded methods will not be reused again and therefore reloaded. This strategy is not realistic in the sense that it is unlikely that we will be able to have such accurate information (last call time) for all methods given non-deterministic execution and cross-input program behavior. This strategy does however, capture the potential of unloading dead methods and we use it as a limits study.

The second strategy is *TiMer (TM)* triggered. Our system unloads code at fixed, periodic intervals. To be simple and efficient, we use a thread-switch count to approximate the timer since thread switching occurs at approximately every 10 ms in our JVM. This JVM employs stop-the-world style of the garbage collection (GC) which halts all thread switching during GC. To compensate for these time

periods, we extended the GC system to update the thread-switch count at the end of each GC by the amount corresponding to the time spent performing the GC.

Since *TM* is timer based, it is not adaptive, i.e., it does not exploit information about the execution characteristics of the VM, such as memory availability. Moreover, the length of the period is a difficult parameter to set. We found that different period lengths perform better for different programs and even for the same program across inputs. We detail the parameters we use in Section 5.

To address the limitations of TM, we investigated an adaptive *Garbage Collection (GC)* triggered strategy. The intuition behind this strategy is that code unloading frequency should adapt to the dynamically changing resource availability. Unloading frequency refers to how often unloading occurs, e.g., every 60 seconds, every 10 GC cycles, etc. When memory is highly constrained, code unloading should be triggered more frequently to relieve memory pressure. When unconstrained, the system should perform unloading less frequently to reduce the overhead of the unloading process itself and to reuse compiled code as much as possible. To this end, we initially investigated the use of *heap residency*, i.e., the ratio between the amount of memory occupied and the total heap size, to measure memory usage. If the system is short of memory, we will see high heap residency following garbage collection.

However, using heap residency alone to measure memory usage may raise many false alarms. For example, some programs may allocate most of its memory at the beginning of execution. The heap residency will remain high and cause repeated unloading even when no further allocations are made by the program. To avoid false alarms, we use heap residency indirectly by considering GC frequency. When the amount of available memory space is small and programs repeatedly allocate memory, GC will occur frequently. To capture this behavior, at the end of each GC cycle, the resource monitor forwards the percentage of execution time spent in garbage collection so far to the unloader so that the unloader can adjust the unloading frequency.

We specify unloading frequency using a dynamic "unloading window". Our system initiates unloading once per window. The size of an unloading window is defined by a specific number of garbage collection cycles. Users or system administrator can specify a minimal window size using command line options. The unloading system divides this minimal window size by the percentage of time spent in GC to determine the dynamic window size adaptively. The value is decremented upon each GC. When it reaches zero, the system performs unloading. Following each unloading session, the system resets the window size.

As we showed in Section 2, more than 70% of code is dead following the initial 10% of program execution time, i.e., the startup phase, for most benchmarks. To exploit this *phased* behavior, we investigated different unloading strategies that operate at different stages of program lifetime. We define the first 4 GC cycles (which we empirically determined and which can be changed via a command-line parameter) to be startup period. During the startup phase, the program uses heap residency alone to facilitate more aggressive unloading; following this period, the system uses the percentage of time spent in GC to determine when to unload.

The last strategy that we investigated is a *code Cache Size (CS)* trigger. This

strategy is similar to "code pitching" in the Common Language Runtime (CLR) [Box 2002]. In this strategy, we store the compiled native code in a fix-size code cache. When the cache is exhausted, our system performs unloading. The advantage of this strategy is that the size of compiled code body is guaranteed to be below a specified maximum. However, we found that it is very difficult to find a general optimum cache size for all applications. An alternative is to use a small size cache initially and allow the cache to grow as necessary. However, to determine how often and at what increments to grow is equally difficult and application-specific. Regardless of the limitations, we are interested in understanding how this strategy impacts performance. We therefore parameterized this strategy for initial cache size, the growth increment, and the number of unloading sessions that triggers cache growth. We discuss how we selected the parameters for our empirical evaluation in Section 5.

4.2 Identifying Unloading Candidates

To identify which code should be unloaded, we developed strategies that identify methods that are *unlikely* to be invoked in the future. We hypothesize that the methods that have not been invoked recently are not likely to be invoked in the near future and can be unloaded. We present four techniques that use program profiling as well as snapshots of the runtime stack to identify unloading candidates.

The first strategy, called *Online eXhaustive profiling (OnX)*, uses exhaustive online profiling information to identify unloading candidates. To obtain method invocation counts, we modified the compiler to instrument methods. The instrumented code marks a bit each time a method is invoked. When unloading is triggered, the system unloads unmarked methods and resets the mark bits. In this way, both dead methods and infrequently invoked methods can be unloaded. This strategy guarantees that every method that has been invoked since the last unloading session will have its mark bit set. Therefore, only recently unused methods will be unloaded. However, since profiling is performed for every single method invocation, this strategy has the potential for introducing significant execution time overhead.

To overcome this limitation, we also investigated *Online Sample-based profiling (OnS)*. In this strategy, the JVM sets the mark bits of the top **two** methods on invocation stacks of application threads for every thread-switch (approximately every 10 ms). We determined this value (two) empirically in an attempt to balance the tradeoff between introducing significant overhead of complete stack scan and incorrectly unloading used methods. Moreover, this sample-based approach can be turned off when sufficient memory is available to avoid *all* overhead. For exhaustive profiling (OnX), we do not turn off profiling since doing so requires recompilation and possibly on-stack replacement [Fink and Qian 2003].

We also investigated the efficacy of using perfect knowledge of method lifetime – to facilitate our MCT (maximum call times) trigger. For this strategy, we gather the total invocation count for each method *offline*. Then we annotate this value in the class file as a method attribute for use by the JVM during program execution using an annotation system that we developed in prior work [Krintz and Calder 2001; Krintz 2003]. At runtime, we use online profiling to identify a method's last invocation; at which time, we mark the method to be unloaded. Instead of the 1-bit counter used in *OnX*, this strategy requires that we increment an integer counter for each invocation. We assume that the same input is used for both offline profiling

and online execution, that is, we use perfect information of method invocation counts. We refer to this strategy as *Offline exhaustive profiling (Off)*.

Our final strategy, called *NP* for "*No Profiling*", simply unloads all methods that are not currently on the runtime stack when unloading is triggered. This strategy is the most aggressive one. The advantages include simplicity of implementation and avoidance of all profiling overhead. However, this strategy is not adaptive and may unload methods that will be invoked in near future, introducing significant recompilation overhead.

4.3   Unloading Optimized Code

Heretofore, we have not considered whether the method we are unloading is optimized or not. Unloading optimized code has the potential of increasing the performance penalty of unloading when a method is later reused. This is because optimizing code is much more expensive than compiling code without optimization. We refer to the latter as *fast compilation*. For JVM configurations in which only fast compilation is used, there is no optimized code to unload.

However, in addition to a JVM configured with only a fast compiler, we consider an *adaptive* configuration. In an adaptive optimization, a method is initially fast compiled. The JVM samples methods to identify those where the most time is spent, i.e., that are "hot". The system then uses a counter-based model (as HotSpot [HotSpot 2001] employs) or a cost/benefit analytic model (as JikesRVM [Arnold et al. 2000b] employs) to decide when and at which level to compile or recompile a method depending on its hotness. Higher level of optimization enables larger performance improvement, but also introduces additional compilation overhead.

For an adaptively optimizing JVM configuration equipped with code unloading, we must determine the level at which unloaded optimized code should be compiled if it is later reused. If we use fast compilation, the method will have to progress through the optimization levels again if it remains hot after unloading. Alternately, if we optimize the method at the level at which it was when it left the system, it may no longer be hot when it returns; this imposes unnecessary compilation overhead on the program.

We implemented three additional strategies to study the performance impact of unloading optimized code. First, we insert an optimization level hint to the recompilation stub. If an unloaded hot method is later invoked, our system re-compiles it at the optimization level that it was before unloaded. We call this strategy *RO*: *R*e-Optimize hot methods using an optimization level hint. Second, we avoid unloading all hot methods. We call this strategy *EO*: *E*xclude *O*ptimized methods from unloading. This strategy avoids the compilation overhead of optimization. However, some programs may have a significant percentage of hot methods (that should be unloaded). For example, in *javac* from the SpecJVM98 benchmark suite, there are 78 out of 876 methods that are hot. In comparison, *db* only has 3 out of 151 methods that are hot. Our third strategy accounts for cases like *javac*: The optimized (thus hot) methods will be unloaded. However, we delay unloading until the method is unused for two consecutive unloading sessions. If an optimized method is unloaded after giving second chance, it is fast compiled at next time it is invoked. We call this strategy *DO*: *D*elay unloading of *O*ptimized methods.

4.4 Recording Profile Information

Another implementation issue that we must address is how the system should record profile information. As we described above, for the implementation of the "what" strategies, we use a bit array to record information gathered either by instrumented code or by sampling. Every time a method is invoked or is on the top of stack when thread switching occurs, the system sets a bit in the array that corresponds to the method. When unloading occurs, the system unloads all unmarked methods and resets the array.

The benefits of using a bit array to record profile information, are that the implementation is simple and access to the array is very efficient. However, a bit array does not capture the temporal relationship between method invocations. That is, as long as two methods are invoked since last unloading session, they will be treated equally by the next unloading session no matter which one is the more recently invoked.

To evaluate whether such temporal information is important or not, we implemented an additional mechanism for recording profile information in which all methods are linked via a doubly linked list. A method is inserted at the end of the list when it is compiled. Whenever the method is invoked again (in the exhaustive profiling case) or sampled (in the sample-based case), the system moves it to the end of the list. As a result, all methods are always ordered by their last invocation (or sample) time. Such an implementation enables our use of the framework to investigate the efficacy of using the popular *Least Recently Used (LRU)* cache replacement policy for code unloading.

5. EXPERIMENTAL METHODOLOGY

We implemented and empirically evaluated the efficacy of our code unloading framework and various unloading strategies in the Jikes Research Virtual Machine (JikesRVM) [Alpern et al. 2000] (x86 version 2.2.1) from IBM Research. Although JikesRVM was not originally designed for embedded systems, it has two different compilation configurations that we believe are very likely to be implemented in the next-generation JVMs (embedded or not): *Fast*, non-optimizing compilation, and *Adaptive* optimization (in which only methods that have the most performance impact are optimized). We investigate both compiler configurations since it is unclear as to how much optimization should be used by JVMs for embedded devices.

One limitation of JikesRVM is that it does not implement an interpreter and thus, we are unable to use it directly to evaluate the impact of code unloading on selective compilation, i.e., a system that employs interpretation for cold methods and compilation (and increasing levels of optimization) for hot methods, e.g., as in HotSpot JVM from Sun Microsystems [HotSpot 2001]. To our knowledge, no open source JVM implements an interpreter, a highly optimizing compiler, and adaptive optimization. As such, we use simulation to evaluate the impact of code unloading for selective compilation JVMs in Section 6.3. We describe our simulated setup and assumptions in that section.

We investigate maximum heap sizes of MIN and 32MB to represent memory resource constraints. MIN has the minimum heap size that is necessary for each benchmark to run completion (identified empirically) without an out of memory

Table II.   Benchmark Characteristics for Fast Configuration

Benchs	Code (KB)	MIN (MB)	Memory Used(MB)	Exec Time (s)		GC Ratio(%)		CMP Ratio(%)	
				min	32MB	min	32MB	min	32MB
compress	98.4	20	122.2	66.8	61.2	9.99	3.10	0.04	0.04
db	105.3	22	83.7	78.8	50.6	38.96	7.41	0.03	0.05
jack	284.9	6	238.1	624.9	17.9	97.42	28.35	0.01	0.31
javac	468.5	24	232.3	128.9	46.8	77.61	41.89	0.10	0.26
jess	223.1	8	274.3	303.3	27.6	91.99	25.86	0.02	0.20
mpeg	455.4	9	14.3	56.1	54.4	1.69	0.00	0.11	0.11
mtrt	161.3	18	149.3	321.5	29.6	92.66	21.29	0.01	0.12

Table III.   Benchmark Characteristics for Adaptive Configuration

Benchs	Code (KB)	MIN (MB)	Memory Used(MB)	Exec Time (s)		GC Ratio(%)		CMP Ratio(%)	
				min	32MB	min	32MB	min	32MB
compress	143.8	22	130.3	26.3	21.5	23.59	8.07	0.80	1.04
db	157.8	23	95.3	115.6	45.1	64.78	12.42	0.21	0.51
jack	372.4	9	248.4	130.6	18.2	88.72	32.70	0.33	2.04
javac	582.8	26	247.2	152.3	55.5	80.94	49.42	0.44	1.08
jess	311.8	11	288.7	136.4	23.2	88.23	37.34	0.50	2.73
mpeg	541.4	12	45.9	29.7	20.3	30.27	3.27	2.56	4.45
mtrt	237.8	23	152.8	50.4	22.6	71.05	38.70	1.36	2.93

exception. We use MIN as an example of a scenario in which memory is highly constrained and 32MB as an example of unconstrained memory. Given this experimental methodology, we believe that our results lend insight into the potential benefits of adaptive code unloading on future, compilation-only and selective compilation JVMs for embedded systems.

In our experiments, we repeatedly ran the SpecJVM benchmarks (input 100), on a dedicated Toshiba Protege 2000 laptop (750 MHZ PIII Mobile) with Debian Linux (kernel v2.4.20) using both the Fast and Adaptive JikesRVM compilation configurations. In both configurations, the commonly used VM code is compiled into the boot image. In addition, we employ the default JikesRVM garbage collector, a semispace copying collector. In all of our results, we refer to the reference (unmodified) system as *clean*.

The general benchmark statistics are shown in Table II (Fast configuration) and III (Adaptive configuration) for the clean system. In each table, the first column is native code size (including all compiled methods, applications and libraries) in kilobytes (KB). The second column is the empirically identified MIN value. The third column is the total size of memory allocated during the execution. The last six columns show the total execution time (in seconds), the percentage of time spent in GC and the percentage of time spent in compilation. Note that JikesRVM is equipped with a facility to track the time spent by each thread. For IA32, it uses system call "gettimeofday" to get the current time. CPU time accumulation of one thread is stopped when the thread is switched out and resumed once its execution resumes. Based on this timing facility, JikesRVM is able to measure time spent in GC and time spent in compilation accurately.

To compare the different strategies, a set of parameters is required. We empirically evaluated a wide range of parameters for each strategy and only report results using best-performing values (on average) across all benchmarks. We set 10 GC

Table IV. Average code size reduction (%) of different "what" and "when" strategies

What	MIN		32MB	
Strategies	Fast	Adaptive	Fast	Adaptive
Off-TM	38.3	N/A	31.9	N/A
NP-TM	55.1	43.8	51.0	40.3
OnS-TM	53.1	42.8	50.5	38.7
OnX-TM	45.6	34.4	43.7	27.0
When	MIN		32MB	
Strategies	Fast	Adaptive	Fast	Adaptive
Off-MCT	42.7	N/A	50.9	N/A
OnS-CS	52.0	40.3	56.7	41.6
OnS-GC	61.8	46.9	46.3	36.0
OnS-TM	46.7	42.8	30.6	38.7

cycles as the unloading window size for GC (garbage collection triggered), 10 seconds as the interval for TM (timer triggered). For CS (code size triggered), initial cache size is 64KB and grows by 32KB for every 10 unloading sessions (triggered by a full cache).

6. RESULTS

In the subsections that follow, we evaluate the efficacy of our code unloading strategies for memory footprint reduction. We then present the impact of this reduction in memory pressure on performance. We evaluated all possible permutations of "what" and "when" strategies. However, we only present a subset of results in this section for conciseness and clear illustration. Finally, we explore the potential impact of coupling code unloading and selective compilation, i.e., a JVM configuration that employs interpretation of cold methods and compilation of hot methods using increasing levels of optimization.

6.1 Memory Footprint Reduction

We first compare the average code size reduction over a clean version of the system, in Table IV. The clean system is a compile-only JikesRVM system with no code unloading extensions. The left half of the table is for MIN memory configuration and the right half is for 32MB (again, MIN is the minimum heap size in which the program will run and 32MB represents a system with less memory pressure). For each heap size, we also present the results for different compilation configurations (Fast or Adaptive). We show two sets of data in this table: one for "what" strategies and the other for "when" strategies.

The upper part of Table IV shows the four "what" strategies as described in Section 4.2: Off (Offline profiling), NP (No Profiling), OnS (Online Sample-based profiling), and OnX (Online eXhaustive profiling). We choose *Timer-triggered (TM)* to be the common "when" strategy since its periodicity enables us to focus only on the impact of different "what" strategies. We omit the results of Off-TM strategy for Adaptive configuration since the adaptive optimization system in JikesRVM is non-deterministic: It uses timing information to decide when and how to optimize and hence, we are unable to obtain deterministic offline profile of method invocation

counts for the fast compiled version and the optimized version separately, which are required by Off-TM strategy to trigger unloading correctly.

For both memory configurations, *NP-TM* performs the best, followed by *OnS-TM*, *OnX-TM* and *Off-TM*. Strategy *Off-TM* does not unload a method until it is dead. Thus, it is the least aggressive. *NP-TM* always discards all compiled methods except those on the runtime stack during an unloading session resulting in the largest reduction in average code size. Online exhaustive profiling is more accurate than sample-based profiling in capturing recently invoked methods. Thus, *OnX-TM* unloads fewer methods than *OnS-TM*.

The code size reduction in 32MB setting is less than that in MIN setting in most cases. The reason for this is that programs execute and complete faster when more memory is available. Hence, fewer unloading sessions were triggered. Similarly, the reduction in Adaptive configuration is less than that in Fast configuration.

The bottom part of Table IV compares four "when" strategies as described in Section 4.1: MCT (Maximum Call Time triggered), CS (Code Size triggered), GC (Garbage Collection triggered), and TM (TiMer triggered). We used the best-performing "what" strategy – OnS (online, sample-based profile) – for these experiments. The MCT "when" strategy requires an accurate, offline exhaustive profile of methods' maximum invocation numbers (thus, we prefix the name with "Off-"). Again, we omit Off-MCT for Adaptive configuration due to the non-determinism.

Similarly to the results of "what" strategies, all "when" strategies have a significant code size reduction for all configurations. *OnS-GC* adapts the best to the memory availability: the more the memory is limited, the more native code is unloaded. *OnS-TM* is also sensitive to memory pressure since our implementation of this strategy updates the timer period for the time spent in garbage collection. OnS-TM is less adaptive than OnS-GC, however, since it does not account for changes in phases of program execution. In contrast, *Off-MCT* and *OnS-CS* are not sensitive to memory availability at all.

Next, we show how code size changes over time with and without unloading. We only focus on the best-performing combinations of *What* and *When* strategies: online, sample-based profiling triggered by GC invocation count (OnS-GC). Figure 3 shows code size over the lifetime of each benchmark using MIN and the Fast compilation configuration. The x-axis is the elapsed execution time in seconds and the y-axis is the native code size in kilobytes. We record code size following each GC and at the end of execution. If unloading occurs during a GC, we record the code size both before and after unloading. We show the results for the clean, OnS-GC, and OnX-GC systems. By comparing OnS-GC with OnX-GC, we can better understand the impact of the more aggressive unloading performed by the OnS strategy. The vertical line for each graph indicates the time at which the program ends.

The graphs in this figure illustrate the impact of code unloading on heap residency. Code size in the clean system becomes stable after a very short startup period and remains at a high level until the application ends. In contrast, both OnX-GC and OnS-GC quickly reduce the code size significantly. OnS-GC is more aggressive than OnX-GC since it unloads any methods that it believes (inaccurately) are not used recently. In addition, both strategies exploit phase behavior by unloading code more aggressively in the early stages of execution, thus, they reduce
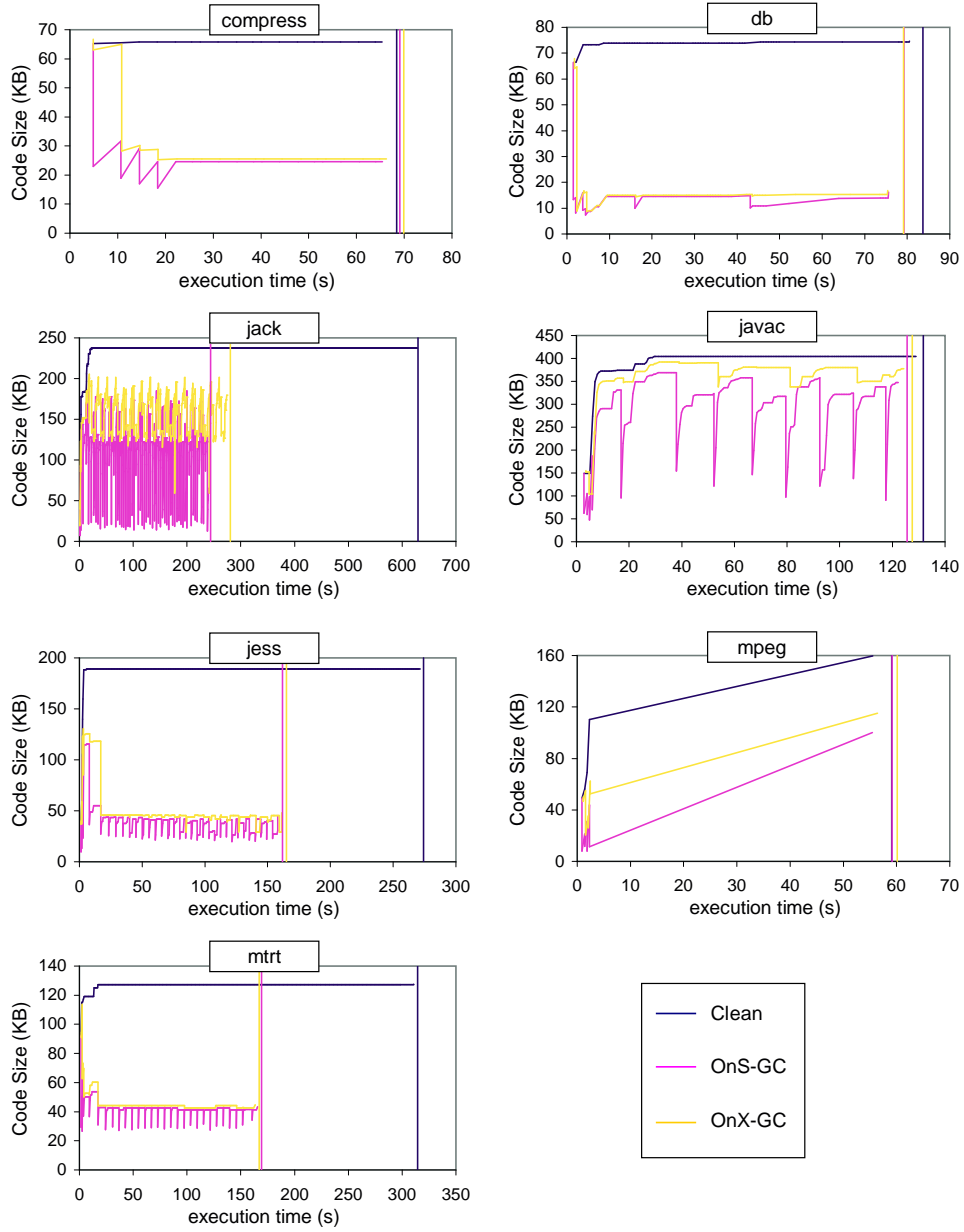
Fig. 3. Size of code residing in the system during program execution of Clean, OnX (Online eXhaustive profiling) and OnS (Online Sample-based profiling) strategies of Fast Configuration

code size significantly for many applications (such as compress, db, etc.) that have a large amount of dead code following program startup.

The above results show code size reduction enabled by code unloading. Our system requires very little memory for the implementation of code unloading in the form of internal data structures and code. As we discussed in Section 4.4, we use a bit array to record profiling information, one bit per compiled method. This implementation requires less than 50 bytes of memory on average. In addition, we add approximately 100 lines of Java code to the code base to perform profiling and code unloading.

6.2   Impact on Execution Performance

Code size reduction is not our only concern. If it were, never caching any code would be the best choice. Our ultimate goal is to achieve the best execution performance while maintaining a reasonably small memory footprint.

The performance of our JVM enhanced by adaptive code unloading is influenced by the overhead of recompilation, profiling, and memory management. Memory management overhead refers to the processing cost of stored native code. No matter how native code is stored in a JVM, when memory size is limited, the more of the heap that is allocated for native code, the less that is available for the application, and the more management overhead that is imposed by the stored code. Thus, unloading code when memory is highly constrained can reduce management overhead. The significance of such reduction, however, depends upon how native code is managed in a JVM.

In general, there are three ways to manage native code in a JVM: (1) using a dedicated memory area not managed by the garbage collector (GC); (2) using a GC-managed heap area separated from application heap; and, (3) using a GC-managed heap area shared by application. Storing native code in a GC-collectable memory area eases the memory management because garbage collector can manage the code memory and application memory uniformly. However, this introduces the extra GC overhead. Storing code in a dedicated memory area removes the interference between the native code and the applications. However, it also introduces extra overhead for maintaining multiple heaps and preventing code memory from being used by applications. It is an open question as to which of the three approaches is the best.

In this work, we evaluated the third option, i.e., storing code in the same GC-managed heap shared by the application. In subsections that follow, we compare the performance impact of different "what" and "when" strategies. We then evaluate the different ways in which we can handle optimized code and gather profiling information. Finally, we summarize two best strategies and show how our strategies adapt to available heap size.

6.2.1   *Comparison of What Strategies.* Figure 4 shows the performance impact of the various strategies that decide *What* methods to unload: offline profiling (*Off*), no profiling (*NP*), online sample-base profiling (*OnS*), and online exhaustive profiling (*OnX*). For each strategy, we use the timer-triggered (TM) *When* strategy (with a 10 s period).

The y-axis in all graphs shows the percent improvement (or degradation) over
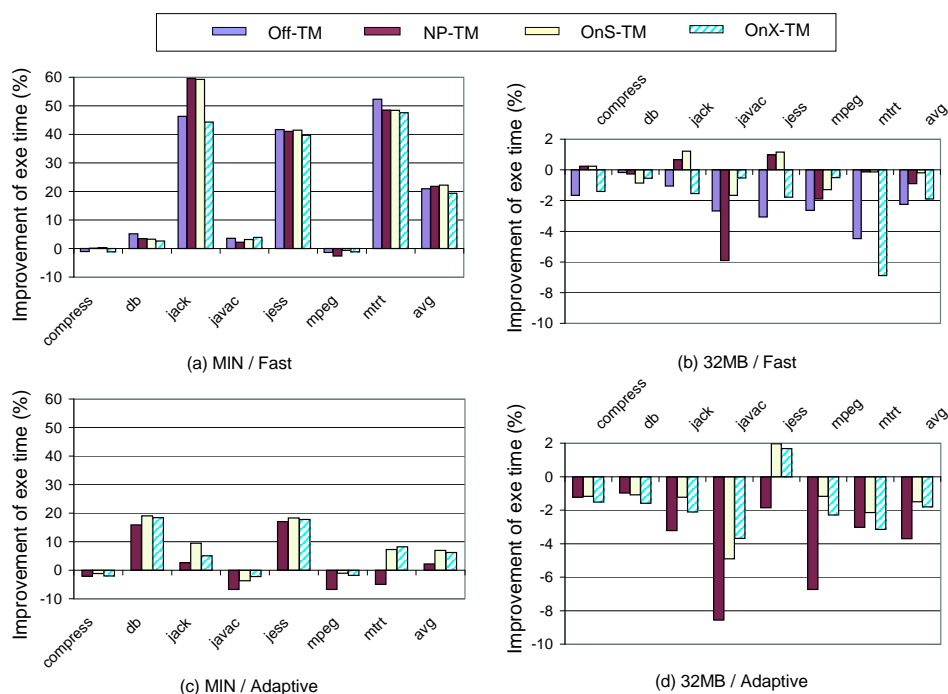
Fig. 4. Comparison of performance impact of the four "what" code unloading strategies. Each graph shows results for one of the four memory and compilation combinations. MIN/32MB indicates the memory configuration used, and Fast/Adaptive indicates the compilation configuration. Strategies investigated are Off (Offline profiling), NP (No Profiling), OnS (Online Sample-based profiling), and OnX (Online eXhausize profiling), with same "when" strategy, i.e., TM (TiMer triggered).

the clean system. Graphs (a),(b),(c), and (d) show four different combinations of memory and compilation configurations: (a) and (b) show performance results when no optimizations are performed (Fast); (c) and (d) are results with adaptive compilation; (a) and (c) show the results when memory highly constrained (MIN); and (b) and (d) show the results when memory is unconstrained (32MB).

In general, when memory is critical, unloading some code bodies significantly relieves memory pressure. As stated above, compiled code is stored in a heap that is shared by the applications and is managed by the garbage collection system, for these results. The results indicate that, for such systems, reducing the amount of native code in the system significantly improves performance when memory is highly constrained since less time is spent in GC.

With the fast compiler (Figure 4(a)), Jack, jess, and mtrt show execution time reductions of over 40%. This is due to the continuous memory allocation requirements (and hence, GC activity) for these applications. Under high memory pressure, majority of the execution time is spent on thrashing between allocation and garbage collection. Therefore, a small amount of memory freed up by code unloading results in significant performance improvement. Compress and mpegaudio show

performance degradation. This is because both benchmarks have relatively small memory requirements (18 and 3 GC cycles, respectively). As such, the benefit from unloading turns out to be less than the overhead introduced by unloading. The impact of code unloading also depends on unloading opportunities provided by the application, such as the percent of short-lived methods. For example, approximately 80% of javac's methods have long lifetimes (see Figure 1), which substantially limits the potential of unloading.

Using a fast-compilation JVM configuration, the re-compilation caused by the unloading imposes little overhead since compilation is very fast: the average total compilation time (in seconds) when memory is constrained is 0.1 (374 methods) for clean and Off-TM, 0.5 (3822 methods) for NP-TM, 0.4 (3330 methods) for OnS-TM, and 0.2 (1062 methods) for OnX-TM. Thus, aggressive unloading policies commonly perform well. One example of this is NP, the second-best-performing strategy overall. It performs well on average since it imposes no overhead for profiling and the re-compilation cost is small.

Off-TM, the offline profile-base strategy, performs best in one benchmark (mtrt) since it is able to only unload dead methods with no recompilation overhead. However, it is unable to capture infrequently used methods, which turns out to be an important opportunity for unloading when memory is highly constrained. As such, Off-TM does not do as well as the other strategies for most benchmarks. Similarly, OnX-TM does not perform as well as OnS-TM since it is less aggressive than OnS-TM. The average performance improvement, when memory is highly constrained and the fast compiler is used, for each of the strategies, is 20.9% for Off-TM, 21.8% for NP-TM, 22.2% for OnS-TM, and 19.4% for OnX-TM.

With adaptive compilation, the performance improvements gained by code unloading is not as significant as for the Fast configuration. This is because the minimal heap sizes we used for the MIN configuration are the minimal PEAK memory requirements that programs need to run. Due to optimizations, the Adaptive configuration requires larger minimal heap sizes (see Table III). However, optimizations do not happen all the time, and as such, a larger minimal heap sizes actually reduces memory pressure, which reduces GC overheads and improvements enabled by code unloading.

The tradeoff between overheads is slightly different with adaptive compilation (Figure 4(c)). Now recompilation overheads are much larger (since optimization is used). Blindly discarding all compiled code, as is done in NP-TM, does not enable the performance levels of the other strategies that use profile information. However, OnS-TM is still better than OnX-TM, which indicates that the profile overhead saved by sampling and the GC overhead reduction enabled by more aggressive unloading of OnS is still more important than the recompilation overhead introduced by inaccurate sampling information when memory is highly constrained. The average performance improvement in this case is 2.2% for NP-TM, 6.9% for OnS-TM, and 6.2% for OnX-TM.

The overhead of unloading (for profiling and recompilation) is more apparent when memory is unconstrained since unloading is unnecessary and thus, pure overhead. The average performance improvement (negative for degradation), when memory is unconstrained and the fast compiler is used (Figure 4(b)), is -2.3%
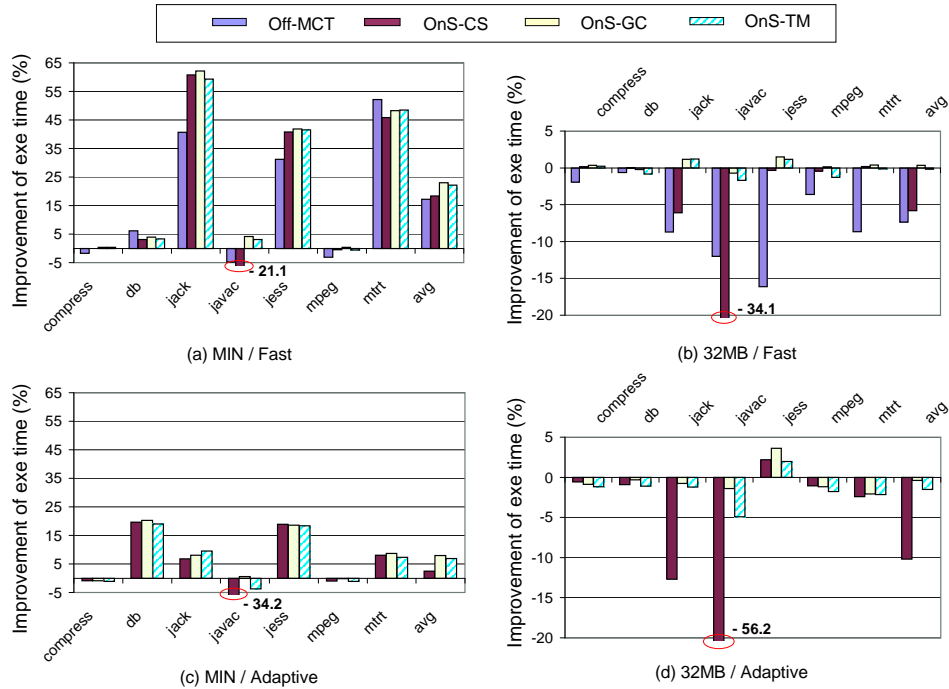
Fig. 5. Comparison of performance impact of the four "when" code unloading strategies. Each graph shows results for one of the four memory and compilation combinations. MIN/32MB indicates the memory configuration used, and Fast/Adaptive indicates the compilation configuration. Strategies investigated are MCT (Maximum Call Times triggered), CS (code Cache Size triggered), GC (Garbage Collection triggered), and TM (TiMer triggered), with the same "what" strategy, i.e., OnS (Online Sample-based profiling), except MCT, with requires offline profiling (Off) to produce an exact count of maximum invocations.

for Off-TM, -0.9% for NP-TM, -0.2% for OnS-TM, and -1.9% for OnX-TM. Off-TM and OnX-TM both perform poorly when memory is unconstrained since both impose overhead for exhaustive method profiling. For the adaptive configuration (Figure 4(d)), the average performance improvement is -3.7% for NP-TM, -1.5% for OnS-TM, and -1.8% for OnX-TM. We can see that the negative impact of NP strategy is more apparent in this configuration due to higher recompilation overhead and less memory pressure.

In summary, the results indicate that a less aggressive and inexact unloading policy with low online measurement overhead (OnS) enables significant performance improvements when memory is critical. In addition, such a strategy imposes little or no overhead when there is ample memory available. That is, OnS provides an adequate estimate of infrequently used methods so that GC overhead can be reduced.

6.2.2 *Comparison of When Strategies.* We next consider the impact of the various strategies that determine *when* unloading should be performed. For all of these results, we use the best-performing "what" strategy, OnS (for all strategies except

Off-MCT – which requires offline profiling to produce an exact count of maximum invocations). The four *when* strategies that we investigated are MCT (trigger: max invocation count using perfect-profile information), CS (trigger: size of cached code), GC (trigger: GC count), and TM (trigger: timer alarm).

Figure 5 shows the performance results due to the various "when" unloading strategies. The format of the figure is the same as those presented previously. The y-axis in both graphs is the percent improvement (or degradation) over the clean system. With the fast compiler, the average performance improvement achieved by our "when" strategies is 17.2% for Off-MCT, 18.4% for OnS-CS, 23.0% for OnS-GC, and 22.2% for OnS-TM when memory is highly constrained. When memory is not critical (32MB), recompilation and profiling overhead introduced by code unload outweighs the GC benefits gained. The average improvement is -7.4% for Off-MCT, -5.8% for OnS-CS, 0.4% for OnS-GC, and -0.2% for OnS-TM.

In general, the best-performing strategy is GC which uses the frequency of garbage collections to trigger unloading. MCT imposes large profiling overhead. It also requires an accurate, input-specific, offline profile, which may not be realistic for mobile programs. CS works well when method working set size is similar to the code cache size. However, it is impossible to accurately predict code cache size. An incorrect prediction may cause significant performance degradation since it results in unnecessary unloading sessions, thus, introducing recompilation overhead. For example, the code cache size of javac in Fast configuration grows to 360 KB at the end of execution; this is much larger than the initial code cache size (64KB).

The performance impact for the Adaptive configuration is similar, except that: first, the GC benefits due to code unloading is smaller because of the larger MIN sizes as discussed previously; and second, the recompilation penalty of aggressive unloading is larger because of expensive optimizations. In summary, the average performance improvement is 2.5% for OnS-CS, 7.9% for OnS-GC, and 6.9% for OnS-TM when memory is highly constrained. When memory is not critical, the improvement (degradation if negative) is -10.2% , -0.4% , and -1.5% respectively.

6.2.3 *Handling Optimized Code.* To improve the performance of code unloading for the adaptive compiler configuration, we investigated three additional variants of OnS for unloading optimized code. They include delaying unloading of optimized code for an additional unloading session (*OnS-DO*), excluding optimized code when unloading (*OnS-EO*) and unloading optimized code and re-optimizing it at the same level if re-invoked (*OnS-RO*). The default OnS uses fast compilation when the unloaded method is re-invoked. All these strategies use the best "when" strategy, GC (garbage collection triggered). Figure 6 shows the results with both MIN (a) and 32MB (b) configurations.

The data in the figure shows that OnS-DO-GC (delay unloading) works best for most of benchmarks. The reason for this is that it gives the optimized code an extra chance to stay in the system. It also exploits the opportunities to unload outdated optimized code, unlike OnS-EO-GC. OnS-RO-GC saves the learning time to progressively re-compile a hot method when it is re-invoked. However, the result shows that in most cases, it does not work well because many of hot methods are no longer hot following unloading/reloading. For example, method *_201_compress.Input_Buffer.getbyte()* is hot (invoked more than 1.5 million times)
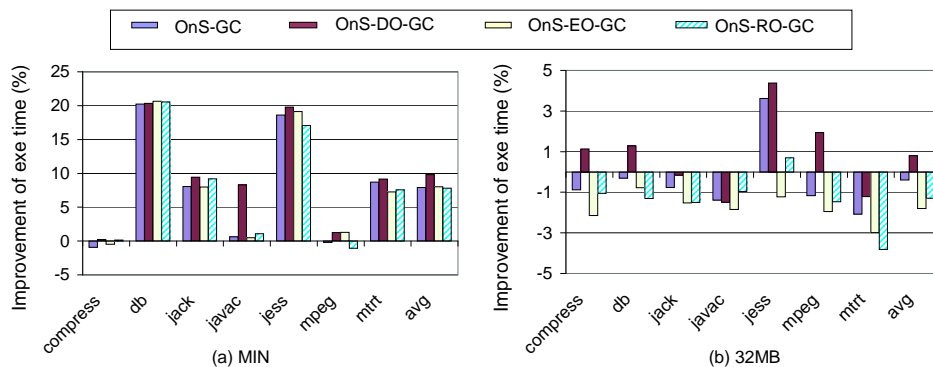
Fig. 6. Comparison of four variants of online sample-based profile for Adaptive configuration. MIN/32MB indicates the memory configuration used. The strategies investigated are OnS-GC (Online Sample-based profiling, Garbage Collection triggered) with different ways to handle optimized code: OnS-GC treats optimized methods as same as other methods and recompiles them with the fast compiler; OnS-DO-GC gives the optimized methods a second chance to stay in the system, but recompiles them with the fast compiler if they do get unloaded and reinvoked later; OnS-EO-GC excludes optimized methods from unloading at all; OnS-RO-GC unloads a optimized method normally, but recompiles it at the optimizing level that it was compiled before unloaded.

before it is unloaded the first time. Due to the phase shift of the program, it is less hot during subsequent execution. However, the method is still invoked periodically (approximately 10 invocations between the two unloading sessions). Since its hotness is not enough to be recognized by the sampling profiler, it is unloaded during every unloading session and optimized upon returning. This introduces unnecessary yet significant overhead.

In summary, the average performance improvement (or degradation if negative), is 7.9% for OnS-GC, 9.8% for OnS-DO-GC, 8.0% for OnS-EO-GC, and 7.8% for OnS-RO-GC when memory is highly constrained (MIN). When resources are unconstrained (32MB), it is -0.4% for OnS-GC, 0.8% for OnS-DO-GC, -1.8% for OnS-EO-GC, and -1.3% for OnS-RO-GC.

6.2.4 *Comparison With LRU.* As we mentioned in Section 4.4, we used a bit array to record profile information with low overhead. One limitation of this implementation is that it does not capture temporal order of method invocations. One possible implementation alternative that can record temporal information is a LRU list: all methods are linked together in the order of their last invocations; whenever a method is invoked or sampled, it is moved to the end of the list. The overhead of maintaining this LRU list is much higher than that of a bit array since it requires several linked-list operations for each update. Moreover, the LRU list requires memory space for two reference fields per method. In this section, we investigate whether more accurate temporal order of method invocations will enable more efficient code unloading in spite of the memory overhead introduced.

We selected our best "what" and "when" strategy combinations so far (OnS-GC for the Fast configuration and OnS-DO-GC for the Adaptive configuration) and reimplemented their mechanism of recording profile information using a LRU linked
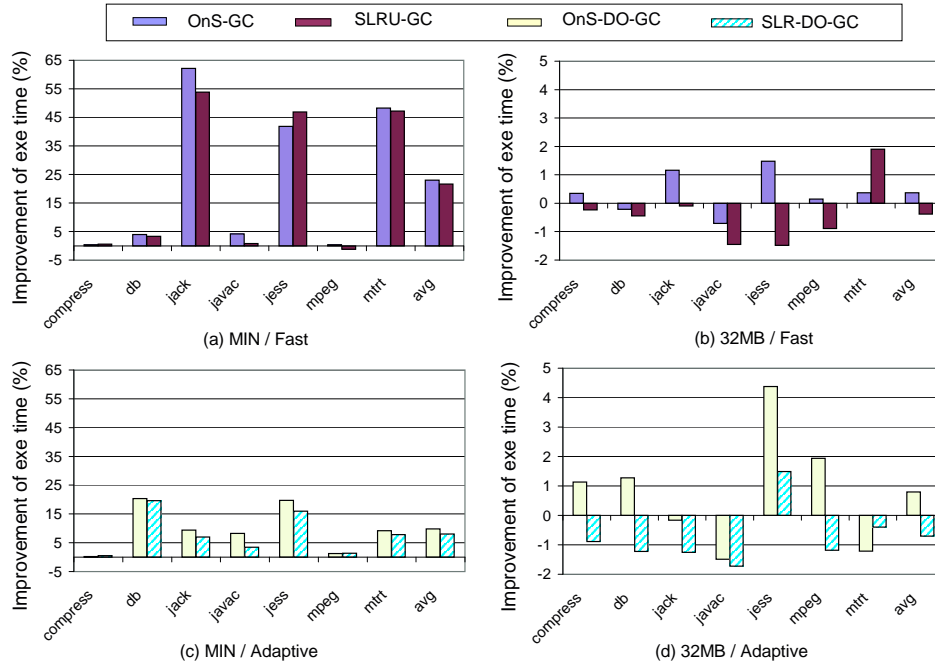
Fig. 7. Comparison of performance impact of the two ways to record profile information: bit array (OnS-GC/OnS-DO-GC) and LRU list (SLRU-GC/SLRU-DO-GC). Each graph shows results for one of the four memory and compilation combinations. MIN/32MB indicates the memory configuration used, and Fast/Adaptive indicates the compilation configuration.

list. Note that for the bit array, we unload all unmarked methods and reset all bit to 0 when unloading is triggered. Thus, the aggressiveness of unloading is controlled by the "when" strategy and no parameter is needed during code unloading. While with a LRU list, the profile information is not discrete (0 or 1), and hence, we need an extra parameter to decide how much we should unload from the LRU list when unloading is triggered. For now, we added a new command-line option, called *unloadFraction*, to control the portion in terms of code size of the LRU list that will be unloaded during each unloading session. We chose a fraction parameter instead of a absolute code size parameter to adapt to different workloads.

Similar to the scenarios of other parameterized strategies, there is no generally best value for this *unloadFraction* parameter across programs. We empirically evaluated a wide range of values for this parameter, and report results using best-performing parameter values (on average) across the benchmarks studied. They are: 40% for the MIN/Fast configuration, and 10% for the other configurations. Figure 7 shows our performance results: OnS-GC (OnS-DO-GC) labels the bit array implementation and SLRU-GC (SLRU-DO-GC) labels the LRU implementation. The memory and compilation configurations are: (a) MIN/Fast; (b) 32MB/Fast; (c) MIN/Adaptive; and (d) 32MB/Adaptive. This figure shows that with a fast compiler, more accurate temporal information provided by the LRU implementation
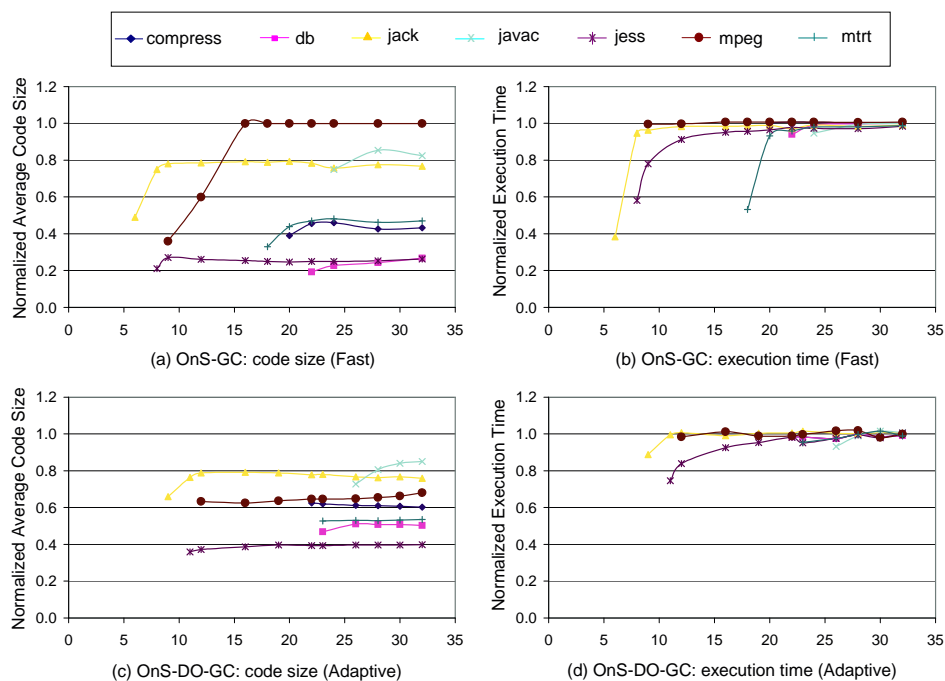
Fig. 8. Summary of code side reduction and performance improvements enabled by our best strategies (OnS-GC for fast compilation and OnS-DO-GC for adaptive compilation) across different heap sizes.

does not enable enough benefits to amortize the additional overhead introduced. With an adaptive compilation configuration, the bit array still performs better than the LRU list in most cases. *mtrt* is an exceptional case, in which accurate temporal order of method invocations does enable more efficient code unloading. One possible reason is that *mtrt* is the only multi-threaded application in the benchmark suite. The interaction between threads makes its performance more sensitive to temporal order of method invocations. In such cases, maintaining a LRU list can improve performance since it enables more accurate code unloading. The average performance improvements across benchmarks are: 23.6% for OnS-GC and 21.6% for SLRU-GC in the MIN/Fast setting; 0.6% for OnS-GC and -0.3% for SLRU-GC in the 32MB/Fast setting; 9.8% for OnS-DO-GC and 8.0% for SLRU-DO-GC in the MIN/Adaptive setting; 0.8% for OnS-DO-GC and -0.7% for SLRU-DO-GC in the 32MB/Adaptive setting.

In summary, we conclude that for the cases we studied, maintaining a LRU list does not enable significantly more efficient code unloading; the bit array implementation is a better choice since it imposes lower overhead and effectively trades off the costs and the benefits of unloading.

6.2.5 *Adaptivity to heap sizes.* Next, we summarize the improvements on code size and overall performance for a range of heap sizes. These results indicate the

adaptivity of our strategies. We present results of the best-performing combination of *What* and *When* strategies: online, sample-based profiling using GC invocation count triggered unloading (OnS-GC) for the Fast configuration, and OnS-DO-GC for the adaptive JVM configuration (Figures 8).

In all of the graphs in Figure 8, the x-axis is heap size. The y-axis in the left graphs is the average code size normalized to the clean version and the y-axis in right graphs is the execution time normalized to the clean version. Graph (a) displays the impact of OnS-GC on code size when the heap size grows from the minimum to 32MB. We can see that when memory is limited, OnS-GC unloads more aggressively, resulting in 61% code size reduction on average. When memory availability grows, the aggressiveness decreases quickly since fewer garbage collections are invoked. On the other hand, the startup strategy guarantees that even when memory is not critical, e.g., 32MB, the dead code in the startup phase will be unloaded. The code size reduction with a 32MB heap is 43% on average.

The reduction in code size enables execution time benefits while imposing very little overhead. Graph (b) shows the normalized execution time of OnS-GC for different heap sizes. We can see that with constrained memory space, unloading can not only reduce the size of cached code, but also improve execution speed by trading off GC time for compilation overhead. When memory size grows, the improvement decreases quickly since there is not much GC overhead to reduce. These results show that our framework and the OnS-GC strategy are able to adapt to dynamic memory availability using the Fast compiler configuration.

Similarly, Graph (c) summarizes the effect of OnS-DO-GC on code size and (d) shows its impact on execution time, with the Adaptive configuration. Since the adaptive configuration requires a larger minimal heap size than that of Fast configuration, the curves in (c) and (d) start from a larger initial heap size. On average, the code size reduction for OnS-DO-GC is 43% with minimal heap size and 38% with 32MB. The performance improvement is 10.3% and 0.1%, for MIN and 32MB, respectively.

6.3   Code Unloading for Selective Compilation Systems

The results in the previous section show that adaptive code unloading is able to monitor the system with low overhead, to make intelligent decisions about what methods to unload and when to trigger unloading, and to reduce code size dramatically without sacrificing performance. Moreover, if system memory is highly constrained, unloading code can enable significant performance improvement when code is stored with application on the garbage-collected heap since much less time is spent performing memory management.

The experimental methodology that we consider is a compile-only Java Virtual Machine, JikesRVM. An alternative to a compile-only system for embedded devices is one that employs *selective compilation*: methods are initially interpreted then compiled using increasingly higher levels of optimization as they become "hot". Even though interpretation of methods that are invoked multiple times has been shown to waste significant resources on embedded devices [Vijaykrishnan et al. 2001; Farkas et al. 2000], selective compilation JVMs produce less compiled code since they interpret many methods. In addition, for methods that are executed for a very small portion of total program execution time, the overhead required to

compile them may not be amortized; selective compilation systems can interpret these methods.

In this section, we consider the impact of code unloading on selective compilation JVMs for resource-constrained systems. Code unloading has the potential to impact the performance of these systems in two ways. First, unloading will reduce the amount of native code stored in the system and possibly reduce memory management overhead by evicting a subset of compiled methods. Second, using code unloading, selective compilation systems can be more aggressive about compilation and optimization decisions. That is, since code unloading reduces the effective memory requirements for stored native code, more interpreted methods can be compiled – which has the potential for improving performance for methods that are invoked repeatedly.

Since our research platform, JikesRVM, is a compile-only system and there are no open-source, selective compilation, systems available (that implement an interpreter, a highly optimizing compiler, and an adaptive optimization system), we investigated the impact of code unloading for selective compilation JVMs using simulation. As we did previously, we consider the CISC, IA-32 architecture. Code unloading for a RISC system, e.g., one that uses the StrongARM processor, will produce even better results in terms of the amount of code unloaded since, as we articulated earlier, RISC native code is 16-25 times larger than x86 equivalent. As such, by considering x86, our results indicate a lower bound on the potential of code unloading for RISC systems.

Selective compilation systems decide which methods to compile and when to compile them using a number of system metrics in much the same way as the compile-only system. Such systems must consider the cost of applying compilation and optimization and the performance that will result if compilation is applied (or not applied). The latter metric requires an estimate of how long the method under consideration will execute in the future. If a method is hot and compiled too late, the compilation overhead may not be amortized and the resulting performance improvement may not impact overall program performance. A hot method thus, should be compiled (i.e., the selective compilation system must identify the method) as early as possible.

A selective compilation system can use a performance model to decide when to compile a method. For this discussion and our evaluation we assume that the model estimates total application execution time as the sum of execution times of every method invocation. We also assume that the speedup of compiled code over interpretation is a constant factor for all methods given the same compilation level. Given these assumptions, we can model the execution time of an application as:

$$T_{exe} = \sum_{i=1}^{n} \begin{cases} T_{jit_i} + T_{intrp_i} * I_i + \frac{T_{intrp_i}}{speedup} * (N_i - I_i) & \text{if } I_i < N_i \\ T_{intrp_i} * N_i & \text{otherwise} \end{cases} \quad (1)$$

where $i = 1, ..., n$ denotes an invoked method, assuming there are n methods invoked. T_{exe} denotes the estimated overall execution time, T_{jit_i} denotes the time spent compiling the ith method, T_{intrp_i} denotes the execution time of one invocation of a method if it is interpreted, *speedup* is the performance improvement achieved by executing stored compiled code, I_i denotes the invocation count of a method

before it is compiled, and N_i denotes the total invocation count of a method. If method i is compiled at some point, the first case of the formula is used, otherwise, the second case is used.

Ideally, if the performance gain that results from compiling a method exceeds the compilation cost, the system should compile the method upon initial invocation of the method to enable optimal improvement, i.e., I_i should be zero for such a method. However, it takes time for a real system to identify (i.e. *learn about*) profitable methods, and thus, I_i will be greater than zero and is equivalent to N_i for those methods for which compilation overhead cannot be amortized.

The actual value of I_i depends on the mechanism that a JVM uses to define a "hot" method. One way a JVM can identify hot methods is by using counters. When the invocation count of a method exceeds a pre-defined threshold, the method is compiled. With this technique, I_i is same for all methods and can be replaced by the threshold in the model. Note that most systems also count back edges to catch methods with long loops; we ignore backedges in this portion of the study to simplify our analysis. In summary, model (1) indicates that the execution time of an application varies given different levels of JIT efficiency, the quality of the compiled code, and the mechanism the JVM uses to identify hot methods.

To understand the dynamics of selective compilation and code unloading, we conducted several experiments. First, we employed the profiling facilities of the Kaffe virtual machine [Kaffe 1998] to gather execution time and compilation time for each method of the SpecJVM98 benchmark suite. Although Kaffe is not the only JVM with such profiling ability, we chose Kaffe since it is an open source project and we can easily extend the profiler to gather more information, e.g. bytecode size, compiled code size, etc. Moreover, Kaffe implements an interpreter and a JIT (but does not implement selective compilation, i.e., mixed-mode execution).

The average speedup that results from using JIT compilation over interpretation in Kaffe is over 20 times. This is because the interpreter is not well-tuned in any way (and not because the JIT applies aggressive optimization – only very simple optimizations are implemented in Kaffe). In a product JVM with selective compilation and a highly tuned interpreter, e.g., HotSpot, the difference between interpreted and JIT execution is much smaller, e.g., 3∼15 times. Since HotSpot is not an open source system, we estimate the speedup enabled by selective compilation over interpretation using the speedups and compilation rates (bytecode in bytes per millisecond) that the JikesRVM compilers enable.

We obtain the compilation rates and speedups enabled by the JikesRVM compilers (and used by JikesRVM to make adaptive optimization decisions) by computing the geometric mean of each across a large set of applications. We assume that compilation with the minimal amount of optimization enables a speedup of 2 times over interpretation of a method; this value has been shown to be a reasonable and conservative estimate in other studies [Adl-Tabatabai et al. 1998; Krall 1998; Suganuma et al. 2000; Yang et al. 1999]. We then use the JikesRVM compilation rates and speedups for higher levels of optimization (used when methods remain hot for a long period).

We simulate the execution time and the size of compiled code using different "hot" thresholds (method invocation counts). In Figure 9 we show the average
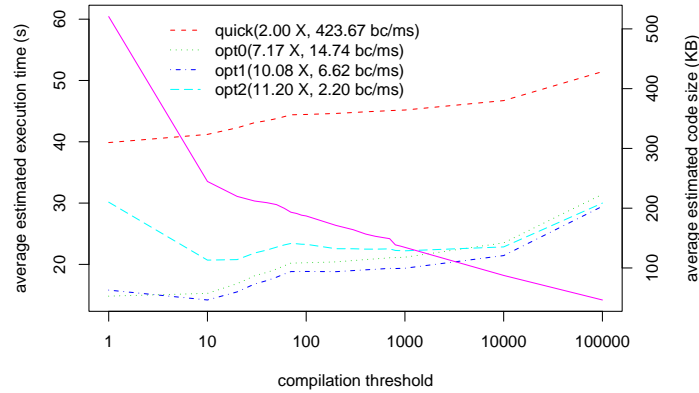
Fig. 9. Average execution time and code size estimation for SpecJVM98 benchmarks using three parameters: compilation threshold, JIT overhead, and speedup of compiled code over interpreted code. The x-axis is the compilation threshold in log scale. The left y-axis denotes the estimated execution time in seconds, and the right y-axis is the estimated code size in kilobytes. The four dashed lines denote the estimated execution time at the four compilation levels, and the solid line represents the changes in code size for different thresholds.

impact of this compilation threshold (x-axis using a log scale) across all of our benchmarks. The left y-axis denotes the estimated execution time in seconds, and the right y-axis is the estimated code size in kilobytes. The four dashed upward lines denote the estimated execution time at the four compilation levels. The solid downward line represents the changes in code size for different thresholds. We provide the estimated speedup and compilation rate (byte code in bytes per millisecond (bcb/ms)) of each compilation level in the legend. Since Kaffe does not have multiple compilation levels, we cannot accurately estimate the change in code size for each compilation level. We measured the average size of native code produced by JikesRVM using different compilation levels. The size ratios of *opt0*, *opt1*, and *opt2* comparing to the quick compiler are around 0.64, 1.00, 1.11. Level *opt1* and *opt2* produce larger code than level *opt0* due to more aggressive inlining. We used these ratios to estimate roughly code size changes for different thresholds. Since all of the code size estimation lines are parallel, we only show the line for the quick compiler for clarity.

The figure indicates that a threshold of 10 achieves the best balance between code size and performance across all compilation overhead/speedup configurations. Code size drops dramatically when threshold moves from 0 to 10, which indicates that many methods are invoked fewer than 10 times. In addition, the performance improvements gains that result from compilation are negligible or negative if we compile these methods since the compilation overhead is not amortized. Once the threshold exceeds 10, the rate of decrease in code size slows while the performance gain becomes more apparent.

A smaller threshold results in more compiled code being stored by the system. At threshold 10, the size of generated code is about half of that of a compiler-only JVM and yet is still substantial (250KB) for embedded devices and can result in significant memory management overhead. If the memory is highly constrained, the JVM
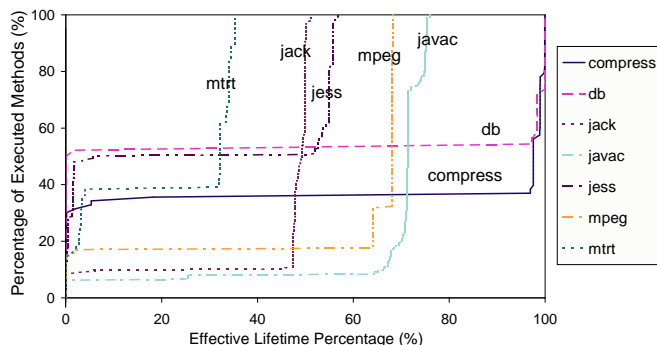
Fig. 10. CDF of effective method lifetime as a percentage of total lifetime as shown in Figure 1. However, we only consider hot methods here. On average, 30% of these methods have effective lifetime percentage of less than 5%. The effective lifetime percentage for most of the other 70% of the "hot" methods is less than 60%.

may not be able to store these code blocks to achieve the optimal performance. Our adaptive code unloading system can help in this situation. By monitoring the execution behavior of the applications and resource availability with low overhead, the adaptive code unloading system enables the JVM to make better use of the precious memory by evicting less useful code blocks so that more aggressive compilation can be performed to carry out performance that is closer to optimal.

Another interesting question that we investigated is: How many methods continue to be hot after they are compiled and how long is their hot period? To investigate this question, we considered the effective lifetimes of methods (as we did previously in Section 2 in Figure 1). In Figure 10, we again plot effective lifetimes but omit those methods identified as cold (invoked fewer than 10 times) in the previous data set. The data shows that for hot methods, on average 30% of them have effective lifetime percentage of less than 5%. That is, the time between the first and last invocations of a method is less than 5% of the total time these methods are in the system. Moreover, the effective lifetime percentage of most of the other 70% of all methods executed is less than 60%.

This data indicates that most methods compiled in a selective compilation system become useless very soon after they are compiled. Our adaptive code unloading system can remove these methods to avoid this memory waste, to reduce memory management overhead, and to enable aggressive compilation decisions in selective compilation JVMs as well as compile-only JVMs.

7. RELATED WORK

This paper is a compilation of and extension to our initial work on profile-guided and adaptive compiled code unloading [Zhang and Krintz 2004b; 2004a]. This paper provides a more complete analysis of the performance of VM code unloading, comparisons with related techniques, e.g., LRU, and an investigation into the efficacy of our system for selective compilation environments. This body of research is related to two primary areas of prior work: code management systems and code size reduction techniques.

Several code cache management techniques have been proposed in prior work. One such technique is *code pitching* which is used in Microsoft .NET Compact Framework [Stutz et al. 2003]. 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 area exceeds a specified maximum, the system "pitches" (discards) the entire contents of the buffer [SSCLI 2002; Stutz et al. 2003]. The VM expands the code cache when a newly compiled method cannot be accommodated even after code pitching or if the overhead of pitching is greater than 5% of the total execution time. Users can specify the initial code cache size and the upper bound of growing. The default value is 64MB for the initial size and the maximum integer value for the upper bound. The minimum initial size allowed specified is 64KB. Code pitching is easy to implement and imposes no profiling overhead. However, it needlessly unloads code 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.

Code cache management has also been used in binary translation systems. The Dynamo project [Bala et al. 2000] and its successor DELI [Desoli et al. 2002] from HP, extract and optimize hot instruction traces from an executing program being translated. These systems store hot traces, called "fragments", in a fragment cache to be reused. When the cache fills, the systems "flush" the cache, discarding all fragments. Dynamo also performs a flush when it detects a dramatic increase in fragments over a short time. These systems employ this simple flushing strategy since many fragments are linked together in the fragment cache and selective unloading can introduce significant unlinking overhead. Our target is the Java virtual machine for which cached code is commonly method-based and unlinked. As such, selectively unloading code using lightweight profiling techniques like sample-based profiling are able to achieve good performance without unlinking overhead.

DynamoRIO [Bruening et al. 2003] is another Dynamo extension that performs dynamic binary optimization. DynamoRIO uses an unbounded code cache by default. However, users can specify a size limit for the code cache. To manage a bounded code cache, DynamoRIO employs a circular buffer similar to that described in [Hazelwood and Smith 2002]. The granularity of such a circular buffer mechanism (FIFO) is investigated in [Hazelwood and Smith 2004]. Their results show that a medium-grained eviction policy results in better performance than both coarse and fine granularities.

The DAISY software emulation system from IBM [Ebcioglu et al. 2001] also employs code cache management. DAISY uses "tree-groups" to represent translated instructions, where control flow joins are disallowed. This causes a code space expansion problem due to tail duplication. The authors overview a simple, low-overhead, generational garbage collection technique to manage a large translation cache (100MB or more). However, we did not find any implementation details on this approach and thus were not able to compare it to our framework. [Hazelwood and Smith 2003] investigates a similar mechanism using DynamoRIO [Bruening et al. 2003] and a generational cache simulator. Our strategies described in Section 4.3 handle the optimized code separately and can be considered as a simplified form of generational cache management.

The purpose of our work is to provide an flexible framework to empirically investigate the efficacy of different unloading strategies and implementation designs, and to help the JVM designers choose the best strategies. Both strategies used in the .NET compact framework and in Dynamo can be configured as *NP-CS* in our framework. NP-CS uses code cache size as the unloading trigger and throws away anything without any profile information when unloading is performed. Our results indicate however, that doing so does not work as well as using a sample-based, GC-triggered configuration.

Code size reduction for restricted resource environments is another research area that is related to our work. Sun's HotSpot technology [HotSpot 2001; 2003] limits the size of compiled code by only compiling the hottest methods and interpreting all other methods. Other work uses *profile-driven deferred* compilation or optimization [Bruening and Duesterwald 2000; Whaley 2001] 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 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 determined by the code unloader according to runtime information and system memory status. Moreover, our code unloading techniques can also be used to manage "hot" methods in these "never cache cold methods" systems,

Another mechanism for code size reduction that have been pursued by other researchers is compression. Compression is a compact encoding of data to reduce storage and transfer requirements. A number of different techniques for compressing compiled code are described in [Ernst et al. 1997; Lucco 2000; Debray and Evans 2002; Drinić et al. 2003]. These techniques, like those for deferred compilation, are complementary to our approach and can be used in combination with our code unloading framework to further reduce the memory overhead of compiled code.

8.   CONCLUSIONS

In this paper, we empirically evaluate the opportunity for dynamically unloading compiled code in compile-only and selective compilation JVMs for mobile and embedded devices. This study shows that, for most of the benchmarks we studied, 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 the time that they are managed by the system. This data indicates that there is much native code in the system that consumes vital memory resources needlessly.

To exploit these opportunities, we developed a dynamic code unloading framework that can be integrated into any compilation-based JVM. Our framework dynamically and adaptively unloads code to relieve memory pressure in resource-constrained systems, while maintaining the performance benefits enabled by compilation. Our system uses dynamic memory availability and program behavior to determine when to perform unloading and to select methods to unload.

We implemented our code unloading framework in the IBM JikesRVM and investigated a number of different unloading strategies and implementation alternatives. Overall, our best strategies enable an average reduction in code size of 47% while introducing zero overhead, when memory is unconstrained. When memory is highly

constrained, our system reduces code size by 62% and execution time by 23% on average across benchmark programs and JVM configurations. Our results also show that for systems that employ adaptive optimization, the best approach to unloading optimized code is to delay unloading initially and to recompile unloaded hot methods without optimization upon reloading. Another result our work exposes is that an LRU approach to unloading does not improve code unloading performance in general and introduces additional overhead by requiring the system to track and estimate usage order.

In summary, our work makes compilation-based JVMs in resource constrained environment more feasible. It does so by enabling more efficient management of compiled code by adapting to dynamic program behavior and resource availability. In the future, we plan to investigate coupling adaptive code unloading with different types of garbage collectors, alternative storage mechanisms, partial method unloading, and novel adaptive strategies.

REFERENCES

ADL-TABATABAI, A., CIERNIAK, M., LUEH, G., PARIKH, V., AND STICHNOTH, J. 1998. Fast,Effective Code Generation in a Just-In-Time Java Compiler. In *ACM Conference on Programming Language Design and Implementation*. 280–290.

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., BURKE, M. G., P.CHENG, CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M. F., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J. C., SMITH, S. E., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño Virtual Machine. *IBM Systems Journal 39,* 1, 211–221.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. 2000a. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000b. Adaptive optimization in the Jalapeño JVM: the controller's analytical mode. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices 35,* 5, 1–12.

BOX, D. 2002. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley.

BRACHA, G., GOSLING, J., JOY, B., AND STEEL, G. 2000. *The Java Language Specification*, Second ed. Addison Wesley.

BRUENING, D. AND DUESTERWALD, E. 2000. 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*.

BRUENING, D., GARNETT, T., AND AMARASINGHE, S. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*. 265–275.

ChaiVM. ChaiVM. http://www.chai.hp.com.

CIERNIAK, M., LUEH, G., AND STICHNOTH, J. 2000. Practicing JUDO: Java Under Dynamic Optimizations. In *ACM Conference on Programming Language Design and Implementation*. 13–26.

CLDC 2003. The cldc hotspot(tm) implementation virtual machine. White Paper. http://web2.java.sun.com/products/ cldc/wp/CLDC_HotSpot_WhitePaper.pdf.

DEBRAY, S. AND EVANS, W. 2002. Profile-guided code compression. In *ACM Conference on Programming language design and implementation*. 95–105.

DELSART, B., JOLOBOFF, V., AND PAIRE, E. 2002. Jcod: A lightweight modular compilation technology for embedded java. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*. Springer-Verlag, 197–212.

Desoli, G., Mateev, N., Duesterwald, E., Faraboschi, P., and Fisher, J. A. 2002. Deli: A new run-time control point. In *35th Annual International SYmposium on Microarchitecture (MICRO'02)*.

Drinić, M., Kirovski, D., and Vo, H. 2003. Code optimization for code compression. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 315–324.

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

Ernst, J., Evans, W., Fraser, C. W., Proebsting, T. A., and Lucco, S. 1997. Code compression. In *ACM conference on Programming language design and implementation*. ACM Press, 358–365.

Farkas, K. I., Flinn, J., Back, G., Grunwald, D., and Anderson, J. M. 2000. 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*.

Fink, S. J. and Qian, F. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization*. 241–252.

Hazelwood, K. and Smith, J. E. 2004. Exploring code cache eviction granularities in dynamic optimization systems. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 89–99.

Hazelwood, K. and Smith, M. D. 2002. Code cache management schemes for dynamic optimizers. In *Workshop on Interaction between Compilers and Computer Architecture (Interact-6)*.

Hazelwood, K. and Smith, M. D. 2003. Generational cache management of code traces in dynamic optimization systems. In *36th Annual International Symposium on Microarchitecture*.

HotSpot 2001. The Java HotSpot Virtual Machine. White Paper. http://java.sun.com/products/hotspot/docs/ whitepaper/Java_HotSpot_WP_Final_4_30_01.ps.

Kaffe 1998. Kaffe – An opensource Java virtual machine. http://www.kaffe.org.

Krall, A. 1998. Efficient JavaVM just-in-time compilation. In *International Conference on Parallel Architectures and Compilation Techniques*, J.-L. Gaudiot, Ed. North-Holland, Paris, 205–212.

Krintz, C. 2003. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization*.

Krintz, C. and Calder, B. 2001. Using Annotation to Reduce Dynamic Optimization Time. In *ACM Conference on Programming Language Design and Implementation*. 156–167.

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

KVM 2000. Java(TM) 2 Platform Micro Edition(J2ME(TM)) Technology for Creating Mobile Devices. White Paper. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

Lindholm, T. and Yellin, F. 1999. *The Java Virtual Machine Specification*, Second ed. Addison Wesley.

Lucco, S. 2000. Split-stream dictionary program compression. In *ACM conference on Programming language design and implementation*. ACM Press, 27–34.

SpecJVM98. SpecJVM'98 Benchmarks. http://www.spec.org/osg/ jvm98.

SSCLI 2002. Rotor - the shared source cli. http://research.microsoft.com/programs/europe/rotor/default.aspx.

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

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

Takahashi, D. 2001. Java chips make a comeback. *Red Herring*.

Vijaykrishnan, N., Kandemir, M., Tomar, S., Kim, S., Sivasubramaniam, A., and Irwin, M. J. 2001. Energy Characterization of Java Applications from a Memory Perspective. In *USENIX Java Virtual Machine Research and Technology Symposium*.

WHALEY, J. 2001. Partial Method Compilation using Dynamic Profile Information. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*. ACM Press, 166–179.

YANG, B., MOON, S., PARK, S., LEE, J., LEE, S., PARK, J., CHUNG, Y. C., KIM, S., EBCIOGLU, K., AND ALTMAN, E. 1999. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

ZHANG, L. AND KRINTZ, C. 2004a. Adaptive Code Unloading for Resource-constrained JVMs. In *ACM Conference on Languages, Compilers, and Tools (LCTES)*. 155–164.

ZHANG, L. AND KRINTZ, C. 2004b. Profile-driven Code Unloading for Resource-constrained JVMs. In *3rd International Conference on the Principles and Practice of Programming in Java*.