Adaptive Code Unloading for Resource-Constrained JVMs

Lingli Zhang Chandra Krintz
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA, 93106, USA
{lingli_z,ckrintz}@cs.ucsb.edu

ABSTRACT

Compile-only JVMs for resource-constrained embedded systems have the potential for using device resources more efficiently than interpreter-only systems since compilers can produce significantly higher quality code and code can be stored and reused for future invocations. However, this additional storage requirement for reuse of native code bodies, introduces memory overhead not imposed in interpreter-based systems.

In this paper, we present a Java Virtual Machine (JVM) extension for adaptive code unloading that significantly reduces the memory requirements imposed by a compile-only JVM. The extension features an unloader that uses execution behavior to adaptively determine **when** to unload as well as **what** code to unload. We implement and empirically identify a set of unloading strategies that enable significant code size reduction (43%-61%). This reduction translates into significant execution time benefits for the benchmarks and JVM configurations that we studied. As such, by using adaptive code unloading, we make compile-only JVMs for embedded devices more feasible.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: *code generation, compilers, memory management, run-time environments*

General Terms

Languages, Performance

Keywords

resource-constrained devices, JVM, JIT, code unloading, code-size reduction

1. INTRODUCTION

Java virtual machines (JVMs) [17] have become increasingly popular for the execution of a wide range of applications on mobile and embedded devices. Researchers estimate that there will be over 720 million Java-enabled mobile devices by the year 2005 [21]. This wide-spread use of Java for embedded systems is the result of the increased capability of mobile devices, the ease of program development using the Java language [4], and the security and portability enabled by JVM execution.

A Java virtual machine (JVM) translates mobile Java programs from an architecture-independent format, i.e., bytecode, into native code for execution. Many JVMs [14, 8, 15] perform translation using interpretation since interpreters are simple to implement, impose no perceivable interruption, and do not require that native code be stored during execution. However, 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 such, interpretation wastes significant resources of embedded devices, e.g., CPU, memory, battery, etc. [22, 10]

To overcome the limitations of JVM interpretation, many JVMs [7, 20, 1, 13] employ just-in-time (JIT), i.e., dynamic, compilation. The resulting execution performance is higher than if interpreted due to improved code quality (that results from translation of multiple instructions at once, exposing optimization opportunities), and to the reuse enabled by storing native code. The latter, however, can be a drawback in a resource-constrained environment since native code is much larger than its bytecode equivalent. Memory consumed by compiled code reduces that is available to the executing application, thereby increasing the cost of memory management, e.g., garbage collection. This cost can be significant when memory is severely constrained. As such, these JVMs introduce memory overhead not imposed by interpreter-only systems.

To reduce this overhead, we developed a framework and set of adaptive strategies, for native code unloading in compile-only JVMs for resource-constrained devices. The strategies that we implemented using the framework seek to adaptively balance not storing any code (as in an interpreter-based JVM) and caching all generated code (as in a compiler-based 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.

If an unloaded method is later re-invoked by the executing application, our system re-compiles it prior to reuse. As such, our system trades off memory management with recompilation overhead, dynamically. We designed the framework so that it is highly extensible and can be easily incorporated into any compile-only JVM intended for resource-constrained devices. Our empirical evaluation of the system shows that we are able to reduce code size by 61% when resources are highly constrained and by 43% when unconstrained. This reduction translates into significant execution time benefits for the benchmarks and JVM configurations that we studied.

In the following section, we motivate our work with an empirical analysis of the memory requirements of compile-only JVMs and discuss the potential opportunities for native code unloading. In Section 3, we describe our adaptive code unloading framework as well as the various strategies that we investigated. We then empirically compare the various strategies and evaluate the overall efficacy of the system in Sections 4 and 5. The remainder of the paper includes related work (Section 6) and our conclusions (Section 7).

2. CODE UNLOADING OPPORTUNITIES

To investigate the feasibility of adaptive code unloading in compile-only JVMs, we empirically evaluated the size and behavior of the native code used by Java programs when compiled. Table 1 shows the size of bytecode and native code for the SpecJVM benchmark suite [18]. Column 2 is bytecode size (in KB) and columns 3–5 show the ratio of native code size to bytecode size. We generated this data using two different JVMs: JikesRVM [1] and the Kaffe embedded JVM [15] (using jit3). Since Kaffe has JIT back-ends for both IA32 and ARM, we present the ratios for both instruction sets. This table shows that IA32 native code is 6-8 times that of bytecode for both JikesRVM (IA32) and Kaffe (IA32). ARM code is at least two times larger than IA32 code. As such, even for 16 bit ISAs, e.g., ARM/THUMB, which can reduce ARM code size by almost half, the size of compiled native code is much larger than its corresponding bytecode.

	Byte	Native code / Bytecode			Dead after
	code	Jikes	Kaffe		startup
Benchs	(KB)	IA32	IA32	ARM	KB (Pct.)
compres	12.4	7.9	7.8	17.0	70.8 (72%)
db	14.5	7.3	7.9	16.7	89.2 (85%)
jack	42.4	6.7	7.5	18.6	72.5 (26%)
javac	78.3	6.0	7.1	16.0	75.9 (16%)
jess	32.9	6.8	7.6	17.0	167.9 (75%)
mpeg	56.6	8.0	8.2	24.5	357.4 (79%)
mtrt	21.1	7.6	8.2	N/A	117.6 (73%)

Table 1: Opportunities for code unloading

Interestingly, a large amount of executed code is used only during program startup (initial 10% of execution time) for the benchmarks we studied. After startup, this code is never invoked again, but remains in the system and consumes precious system memory. The final column in Table 1 shows the percent of native code that is dead after program startup (collected from JikesRVM).

Moreover, code that remains in the system after startup can have short lifetimes, and thus, should also be considered for unloading. Figure 1 shows that a majority of the code in many applications has a short life span: the figure graphs the cumulative distribution functions of *effective method lifetime*

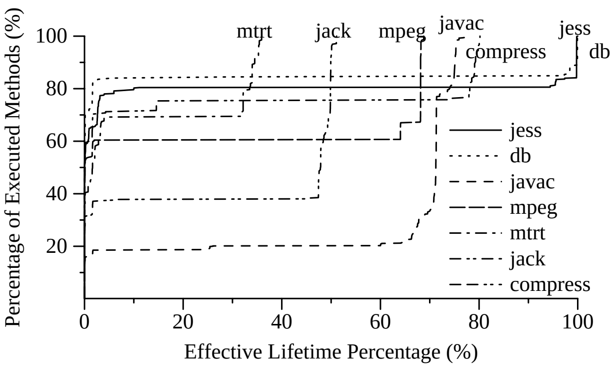


Figure 1: Opportunities for code unloading: short-lived methods. The figure graphs the cumulative distributions of effective method lifetimes (time between its first and last invocation) as a percentage of their total lifetime (time from its first invocation to the end of the program): A point, (x, y), on a curve indicates that y% of that benchmark's executed methods have an effective lifetime of less than x% of its total lifetime.

percentage, i.e., the percentage of the effective lifetime (the time between the first and last invocation of a method) over the total method lifetime (the time from its first invocation to the end of the program). For most of the benchmarks, over 60% of methods are effectively live for less than 5% of the total time they are in the system.

In addition, even long-lived methods may be executed infrequently during their lifetime. For example, the effective lifetime of method spec.benchmarks._213_javac.ClassPath.<init> is 75%. However, this method is only invoked 4 times during its lifetime and its execution time is only 0.1% of total effective lifetime. When memory is constrained, such infrequently used methods can also be considered as unloading candidates to relieve memory pressure.

3. CODE UNLOADING

To exploit the available code unloading opportunities and to relieve the memory pressure imposed by compile-only JVMs for memory-constrained devices, we developed an extensible framework for the implementation of strategies that decide ***What*** code to unload and ***When*** unloading should occur. Since unloading a method that will be invoked later introduces recompilation overhead, we must find a good balance between memory pressure and recompilation overhead.

Our framework is depicted in Figure 2 and can be incorporated into JVM that implements dynamic compilation. The darkened components in the figure identify our JVM extensions. The *Code Unloader* is the control center of our system. The *Resource Monitor* forwards information about resource behavior to the unloader. This component can be easily extended to monitor various types of resources; we currently use it for memory. The monitor collects heap residency data, garbage collection (GC) invocation frequency, and native code size. The on-line and off-line profilers provide the unloader with the information about application behavior. Currently, we collect information about the invocation activity of each method.

Based on information collected from both the resource

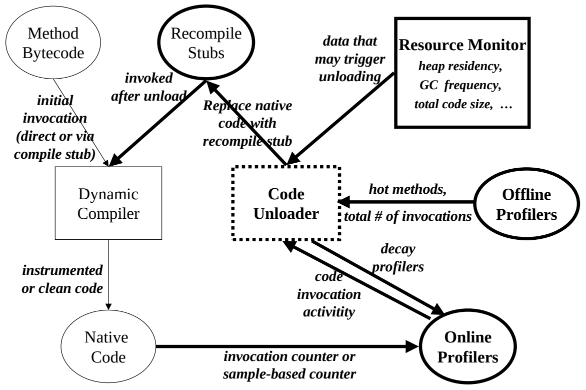

Method Bytecode
Recompile Stubs
data that may trigger unloading
Resource Monitor
heap residency, GC frequency, total code size, …
invoked after unload
initial invocation (direct or via compile stub)
Replace native code with recompile stub
Dynamic Compiler
Code Unloader
hot methods, total # of invocations
Offline Profilers
decay profilers
instrumented or clean code
code invocation activitity
Native Code
invocation counter or sample-based counter
Online Profilers

Figure 2: Overview of code unloading framework

monitor and profilers, the unloader predicts the cost and benefit of unloading, when code unloading should commence, and which methods to unload. When a native code body is selected for unloading, the unloader replaces its address with that of a recompilation stub, in a way similar to that for the compilation stub in lazy, dynamic compilation systems [16, 1]; however our stub contains additional information that guides recompilation if reloading should occur. Once the address is replaced, the native code block for the method is no longer reachable by the program and the storage will be reclaimed during the next garbage collection cycle. If the method is ever invoked again, the recompilation stub causes it to be compiled again prior to execution. Moreover, since the unloader (and other framework components) operate *while* the program is executing, we designed it to be very efficient. In the next two sections, we describe the unloading strategies that we developed using this framework.

3.1 What to unload?

To identify code that should be unloaded, we must predict which methods are *unlikely* to be invoked in the future. To enable this, we monitor execution and identify methods that have not been invoked recently. We hypothesize that such methods are not likely to be invoked in the near future and can be unloaded. In prior work [24], we investigated four different "what" strategies:

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

The strategies, if ranked in order from most aggressive to least aggressive in terms of unloading, are *NP*, *OnS*, *OnX* and *Off*. In the OnX strategy, the compiler instruments methods so that a mark bit will be set to 1 every time a method is invoked. In the OnS, however, instead of instrumentation, the JVM sets the mark bits of the **two** methods on the top of invocation stacks of application threads for every thread switch (which occurs approximately every 10 ms in our prototype JVM). In both strategies, unmarked methods are unloaded and all marked bits are reset to 0 whenever an unloading session occurs.

We used the Off strategy to investigate the efficacy of having perfect knowledge about method lifetimes. In this strategy, we collect total invocation count for each method offline; we then annotate the class file with this value which is extracted by the JVM upon class loading. We unload methods following their final invocation. In the NP strategy, we unload all methods that are not currently on the runtime stack when unloading occurs.

We found that if we performed unloading at regular timer-triggered intervals, OnS performed best across the benchmarks and JVM configurations that we studied. This is because it imposes less profiling overhead than Off and OnX, and is more aware of application behavior than NP. The complete results can be found in [24].

3.2 When to unload?

In the initial investigation of this work described above, we used a naive approach to trigger unloading, i.e., a *Timer Triggered (TM)* approach. To implement this strategy efficiently, we approximated time using a thread-switch count since in our testbed JVM, thread switching occurs at approximately every 10 ms. This strategy also increments the count according to the time spent in garbage collection. Using this approach, our system unloads code at regular intervals. However, such an approach does not account for the dynamically changing underlying memory availability and the resource requirements of the executing program. In addition, such a strategy is not general, i.e., the period that enables the best performance varies across applications, JVMs, and available resource levels. To address these limitations, we implemented an adaptive *Garbage collection triggered (GC)* strategy.

The intuition behind this GC strategy is that the frequency of code unloading should adapt to the dynamically changing resource behavior. Code unloading should be triggered more frequently when memory is highly constrained to relief memory pressure, but less frequently otherwise to reduce overhead. One measurement that represents memory usage behavior is *heap residency*. If the system is short of memory, the heap residency will be high following garbage collection. One disadvantage to using heap residency is that doing so may raise a false alarm. For example, some programs may allocate only at the beginning of their execution. In this case, the heap residency may remain high and exceed the threshold; however, it is not necessary to perform unloading continuously because no additional allocations will be performed. As such, heap residency alone is not accurate enough to enable the unloader to make the accurate unloading decisions. To avoid false alarms, we use heap residency information indirectly by considering GC frequency. If memory availability becomes severely limited and heap residency remains high, the garbage collector will be invoked frequently. To capture this behavior, at the end of each GC cycle, the resource monitor forwards the percentage of execution time (so far) that is spent in garbage collection to the unloader so that it can adjust the frequency of the unloading sessions.

We use "unloading window", i.e., the number of garbage collection cycles, to define unloading frequency. We perform unloading once for each window. Currently, we use a simple algorithm to determine the unloading frequency. We divide a minimal window size specified by a user by the percentage of time spent in GC; this value is decremented upon each GC and when it reaches 0, unloading is performed. At the end of each unloading session, a new unloading window size is determined using this algorithm.

As we articulated in Section 2, for most benchmarks over

70% of code is dead by completion of the first 10% of program execution time (startup period). To exploit this *phased* behavior, we trigger unloading for different phases in program lifetime. Since we cannot know the program lifetime (and hence, the number of seconds during the initial 10%), we estimate it using GC cycles. We specify the first 4 GC cycles (empirically determined and specified via a command-line parameter) as program startup. During this period, we base unloading decisions on heap residency. This facilitates more aggressive unloading by the unloader. After the startup period, the unloader uses the percentage of time spent in GC to avoid false alarms during the steady state of program execution.

We also investigated two additional code unloading strategies: *Maximum call times triggered (MCT)* and *Code Cache Size Triggered (CS)*. For MCT, our goal is to only unload a method if it will not be invoked in the future so as to avoid recompilation overhead: When a method completes its final execution, we unload it. We use offline profiling to estimate the maximum number of times a method is invoked; the framework uses this value to unload methods after their last use. Notice that, if the count is inexact (due to differences in cross-input behavior, inlining decisions, non-determinism), some recompilation overhead will be introduced. In addition, MCT considers only *dead* code, does not unload *infrequently* executed methods, and does not consider (or adapt to) underlying resource availability.

The final strategy that we implemented is Code Cache Size Triggered (CS) unloading. Such a strategy is similar to that implemented as code pitching in the Common Language Runtime (CLR) [3]. For this strategy, we store native code bodies in a fixed-size code cache. When the cache becomes full, we perform unloading. An advantage of this strategy is that the code size is guaranteed to be less than a specified maximum. One limitation of this approach is the determination of an appropriate cache size for all applications. If the size is too small, execution will thrash between recompilation and unloading. If the size is too large, the system acts as if there are no unloading capabilities. An alternative is to assign a small size initially and allow the cache to grow as necessary. However, determining how often and at what increments to grow is similarly difficult and application-specific.

Regardless of these limitations, we were interested in understanding the performance impact of this type of strategy. To this end, we implemented this strategy so that is parameterizable. The parameters include initial cache size, the growth increment, and the number of unloading sessions that triggers cache size growth.

3.3 Unloading Optimized Code

We are interested in the performance impact of our code unloading strategies on JVMs with either fast, non-optimizing JITs or adaptive optimizing compilers. In our prototype JVM configured to adaptively optimize, a method slowly progresses through optimization levels according its hotness. All levels of optimization require significantly more time than fast compilation. Therefore, it may be more efficient to recompile unloaded code using fast compilation. However, if the unloaded method remains hot, it will again have to progress through the optimization levels, which requires a long period of unoptimized execution and significant compilation overhead. Thus, how to recompile an unloaded, previously optimized method will affect the balance of recompilation overhead and performance benefits.

To investigate impact of unloading *optimized* code, we implemented three additional strategies to handle optimized code. For the first, we add an optimization level hint to the recompilation stub. If the unloader unloads a hot method that is later invoked, the system recompiles the method at the optimization level it was at when it was unloaded. This strategy eliminates unoptimized execution of hot methods and is called *RO* (Reload Optimized methods using the optimization hint).

With the second strategy, we avoid unloading hot methods altogether; the unloader checks whether a method has been optimized and only unloads it if it was fast-compiled. We call this strategy *EO* (Exclude unloading of Optimized methods). However, some programs have a relatively large percentage of hot methods. For example, *javac* has 78 out of 876 hot methods while *db* only has 3 out of 151 hot methods. Our third strategy accounts for such cases. Optimized methods will be unloaded but we delay unloading of them until they are unused for two consecutive unloading sessions. We call this strategy, *DO*(Delay unloading of Optimized methods).

4. EXPERIMENTAL METHODOLOGY

To evaluate the efficacy of code unloading, we implemented our framework and various *what* and *when* strategies for adaptive code unloading in the open source Jikes Research Virtual Machine (JikesRVM) [1] (x86 version 2.2.1) from IBM Research. Even though this JVM is not intended for embedded systems, it implements two different compilation configurations that we believe are likely to be implemented in next-generation JVMs (embedded or not): *Fast*, non-optimizing compilation, and *adaptive* optimization (in which only those methods discovered to most impact execution performance are optimized). By limiting its working memory to be less than 32 MB, we believe that our results using JikesRVM as a prototype lend insight into the potential benefits of adaptive code unloading in compile-only JVMs for resource-restricted environments.

For our experiments, we repeatedly executed the SpecJVM benchmarks [18] (input 100), on a dedicated Toshiba Protege 2000 laptop (750 MHZ PIII Mobile) running Debian Linux (kernel v2.4.20). We employed both of the available JVM configurations, fast and adaptive. In addition, we report results for two memory configurations: MIN and 32MB. MIN is the minimum heap size that is required for each benchmark to run to completion (identified empirically). MIN is used to simulate the situation that memory is highly constrained, while 32MB is used to simulate the situation that memory is not highly constrained. In all of our results, we refer to the reference (unmodified) system as *clean*.

We show the general benchmark statistics in Table 2 using the clean system. The left half of Table 2 is for the fast configuration and the right half is for the adaptive configuration. For each half, the first column is the native code size (for all invoked methods, application and library) in kilobytes (KB). The second column is empirically identified MIN value and the last two columns show the application execution time (in seconds) for both MIN and 32MB configurations.

One difficulty in comparing the different *when* strategies is that each requires a set of parameters that impact per-

	Fast configuration				Adaptive configuration			
	Code	Min Heap	Exec Time (s)		Code	Min Heap	Exec Time (s)	
Benchs	Size(KB)	Size(MB)	MIN	32MB	Size(KB)	Size(MB)	MIN	32MB
compress	98.4	20	66.8	61.2	143.8	22	26.3	21.5
db	105.3	22	78.8	50.6	157.8	23	115.6	45.1
jack	284.9	6	624.9	17.9	372.4	9	130.6	18.2
javac	468.5	24	128.9	46.8	582.8	26	152.3	55.5
jess	223.1	8	303.3	27.6	311.8	11	136.4	23.2
mpeg	455.4	9	56.1	54.4	541.4	12	29.7	20.3
mtrt	161.3	18	321.5	29.6	237.8	23	50.4	22.6

Table 2: Benchmark characteristics for both fast and adaptive configuration

formance. Tuning parameters for each application manually might achieve better performance for that application but is not general. Thus, we empirically evaluated a wide range of parameters for each strategy and, for brevity, report results using best-performing parameter values (on average) across the benchmarks studied. We use 10 GC cycles as unload window size for GC (garbage collection triggered), 10 s as the interval for TM (timer triggered). For CS (code size triggered), the initial cache size is 64KB and the growth increment is 32KB, and then we grow the code cache each time 10 unloading sessions occur (triggered by a full cache).

5. RESULTS

In this section, we first evaluate the reduction of memory footprint enabled by different code unloading strategies. Then, we analyze their impact on execution performance in details.

5.1 Impact on Memory Footprint

We first present the average code size reduction due to code unloading across benchmarks, in Table 3. The left half of the table is for the fast configuration and the right half is for the adaptive configuration. The initial investigation of different "what" strategies in [24] shows that strategy OnS (online, sample-based profile) performs best across the benchmarks and JVM configurations that we studied. Thus, for all of the "when" strategies except MCT, we use the online, sample-based profile "what" strategy (OnS- prefix); MCT, as described above, requires a precise, offline exhaustive profile of the maximum number of invocations made to a method (hence, we use the Off- prefix). The adaptive optimization system in JikesRVM is non-deterministic in that it uses timing information to decide when to optimize. As such, we are unable to collect deterministic offline profile information about invocation counts of methods compiled by a given compiler. Thus, we omit the results for MCT for the adaptive configuration.

Both configurations show a significant code size reduction. The *OnS-GC* strategy adapts well to the memory availability: the more restricted memory is, the more native code is unloaded. In contrast, *Off-MCT* and *OnS-CS* are not sensitive to memory pressure. *OnS-TM* is also sensitive to

When	Fast		Adaptive	
Strategies	MIN	32MB	MIN	32MB
Off-MCT	42.7	50.9	N/A	N/A
OnS-CS	52.0	56.7	40.3	41.6
OnS-GC	61.8	46.3	46.9	36.0
OnS-TM	46.7	30.6	42.8	38.7

Table 3: Comparison of four "when" strategies for average code size reduction (%)

memory since our implementation of the OnS-TM strategy is aware of time spent on garbage collection. However, OnS-TM is not as adaptive as OnS-GC since it does not account for phase changes in the program execution.

We next provide a more detailed view of how the amount of code in the system changes over time with and without unloading. We focus on the best-performing combination of *What* and *When* strategies: online, sample-based profiling using an unloading trigger of GC invocation count (OnS-GC).

Figure 3 tracks code size over the lifetime of each benchmark program using the minimal heap size in which each application can run for the fast configuration. The x-axis is elapsed execution time in seconds and the y-axis is the native code size in kilobytes. We gathered this data by reporting code size following each GC and again at the end of execution. If unloading occurred during a GC, we report the code size both before and after unloading. We show results for the clean system, OnS-GC, and OnX-GC. By including OnX-GC, which uses exhaustive profiling to identify recently unused methods more accurately, we can better understand the impact of more aggressive unloading like OnS does. The vertical line for each graph indicates the time at which the program ends.

This data shows the impact of code unloading on heap residency. Code size in clean becomes stable after a very short startup period and remains at a high level until the application exits. Both OnX-GC and OnS-GC, however, quickly reduce the code size significantly. OnS-GC is more aggressive than OnX-GC since it is inaccurate in that it will unload methods that it believes (incorrectly) were not used recently. In addition, the unloading strategies exploit startup phase behavior by unloading code more aggressively in the early stages of program execution. As we saw in Section 2, many applications (such as compress, db, etc.) have a large amount of dead code following program startup. For such programs, aggressive unloading during startup dramatically reduces code size.

5.2 Impact on Execution Performance

Reducing code size is not our only concern; if it were, never caching any code would be the best choice. Our goal is to achieve the best balance between memory footprint and execution performance.

The execution performance of a JVM that implements adaptive code unloading is affected by recompilation overhead, profiling overhead, and memory management overhead. Memory management overhead refers to the extra overhead caused by the stored native code. No matter how native code is stored in a JVM, when memory is limited, the more memory allocated for storing native code, the less memory is available to the application, and the more man-

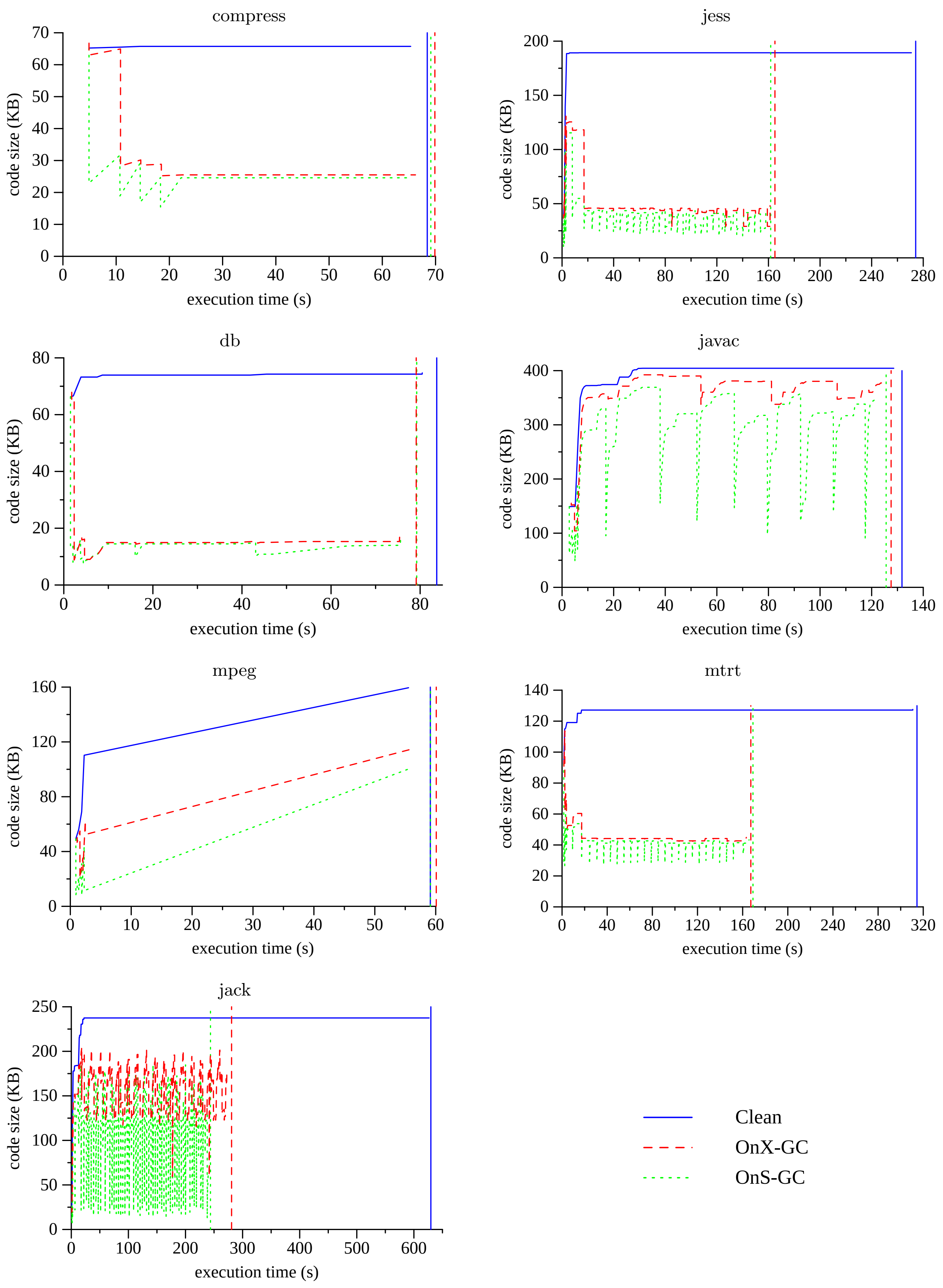
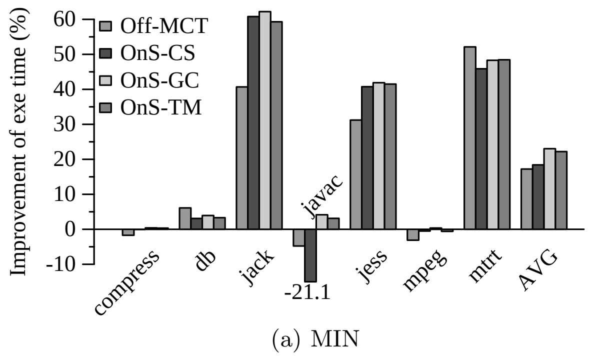


Figure 3: Code Size Comparison of Clean, OnX and OnS strategies of Fast Configuration



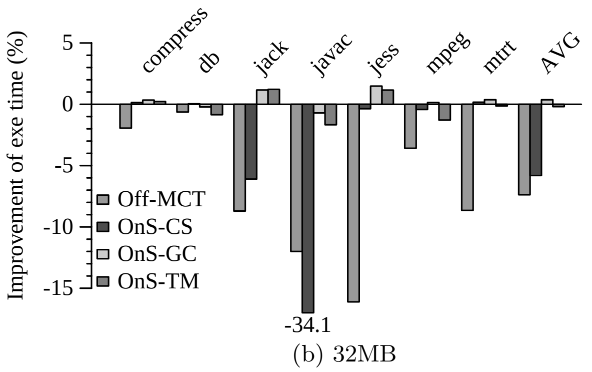


Figure 4: Percent improvement in execution time for the fast JVM configuration using adaptive code unloading. Graph (a) shows results for highly-constrained memory configuration (MIN) and graph (b) shows the results for the unconstrained (32MB) memory configuration.

agement overhead is caused directly or indirectly by the stored compiled code. Thus, unloading code when memory is highly constrained will reduce management overhead to some degree. The significance of such reduction in terms of execution performance, however, depends upon how native code is managed in a JVM.

There are three ways in which native code bodies can be stored by a JVM: Using a dedicated memory area that is not managed by the garbage collector (GC), in a GC-managed heap area separate from the heap used by the application, and in a GC-managed heap area that is used by the application (shared). Storing code in a GC collectable memory area eases the management of memory used by native code since it is managed automatically by the garbage collector. However, this introduces extra GC overhead for managing native code. Storing code in a dedicated memory area reduces the interference between the native code and the applications. However it introduces extra overhead to maintain multiple heaps and prevents the memory reserved for native code from being used by applications. Currently, it is unclear which of the three approaches is the best. For the results in this paper, we evaluated the execution time impact of adaptive code unloading for the third option, i.e., storing code in the same GC-managed heap that is used by the application. The default GC in our implementation system is a semispace copying collector. We plan to investigate other types of GC management strategies (collectors as well as code storage options) as part of future work.

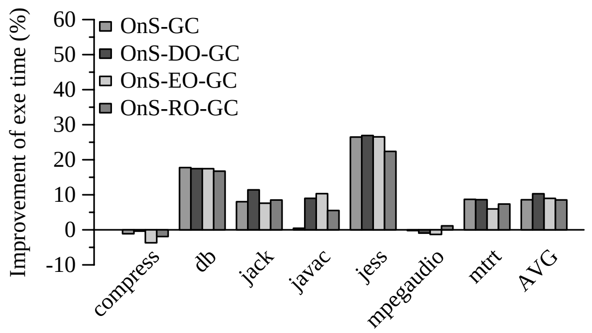
Figure 4 shows the performance results due to code unloading for the minimum (MIN, left graph) and 32MB (32MB, right graph) memory configurations. The y-axis in both graphs is the percent improvement (or degradation) over the clean system. When memory is highly constrained, unloading some code bodies 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 amount of native code in the system significantly improves performance when memory availability is critical since less time is spent in GC. 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.

The overhead of code unloading becomes apparent when memory is unconstrained, as shown in the right graph (32MB) in the figure. This overhead is the result of recompilation and profiling. The best-performing strategy is OnS-GC which uses GC frequency to trigger unloading. The average improvement (or degradation if negative) is -7.4% for Off-MCT, -5.8% for OnS-CS, 0.4% for OnS-GC, and -0.2% for OnS-TM. MCT imposes a high profiling overhead and requires an accurate, input-specific, offline profile, which may not be realistic for mobile programs. CS works well when the size of the method working set is similar to that of the code cache. However, this "perfect" parameter value (code cache size) is difficult to obtain, is not general, and if inaccurate, may cause performance degradation when there is a larger working set size, e.g., as in javac. The code cache size of javac grows to 360KB at the end of its execution, which is much larger than the initial code cache size (64KB). This huge difference between the initial code cache size and the actual working set size causes significant performance degradation for javac since it results in many unnecessary unloading sessions, and thus, introduces large recompilation overhead.

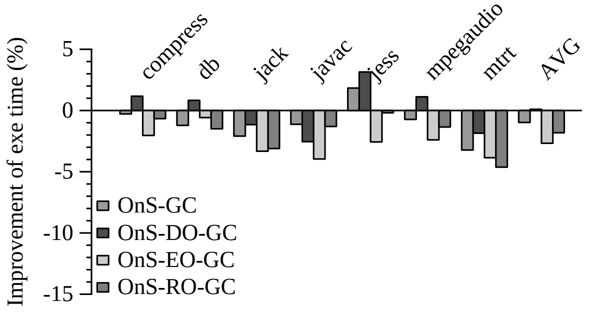
The performance impact using the adaptive configuration is similar; as such we omit the results for brevity. We present optimizations for the adaptive configuration in the next section and provide summary results for both configurations in Section 5.2.2.

5.2.1 Adaptive JVM Configuration

As we discussed in Section 3.3, in the adaptive JVM configuration how to recompile an unloaded, previously optimized method will affect the balance of recompilation overhead and performance benefits. To investigate impact of unloading *optimized* code, we evaluate three additional variants of OnS in the adaptive optimization JVM configuration. They are: delay unloading of optimized code for an additional unloading session (*OnS-DO*), exclude optimized code when unloading (*OnS-EO*), and unload optimized code and re-optimize it at the same level if reloaded (*OnS-RO*). The default OnS fast-compiles unloaded optimized methods upon reloading. We use the best WHEN strategy, i.e., GC (garbage collection triggered) to evaluate all variants. Figure 5 shows the performance results of these four variants for both MIN (a) and 32MB (b) configurations.



(a) MIN



(b) 32MB

Figure 5: Comparison of four variants of online sample-based profile for adaptive configuration

The figure indicates that OnS-DO-GC (delay unloading) works best for most of benchmarks. This is because this strategy gives optimized code an extra chance to stay in the system; in addition, unlike OnS-EO-GC, OnS-DO-GC exploits opportunities to unload outdated optimized code. OnS-RO-GC is intended to save the learning time of a hot method when it is reloaded. However, our results show that in most cases, this strategy does not work well since many of hot methods do not remain **constantly** hot following unloading/reloading. For example, method *spec.benchmarks._201_compress.Input_Buffer.getbyte()* is really hot (invoked more than 1.5 million times) before it is unloaded at the first time. Then, due to the phase shift of the application, it becomes less hot in the following period of execution. However, it is still invoked periodically: about 10 invocations between two unloading sessions. Since it is not hot enough to be recognized by the sampling profiler, it is unloaded upon every unloading session. Then it is optimized upon its next invocation because of the RO strategy, which introduces large, unnecessary optimization overhead.

In summary, the average performance improvement (or degradation), when memory is highly constrained (MIN), for each of the strategies, is 8.6% for OnS-GC, 10.3% for OnS-DO-GC, 9.0% for OnS-EO-GC, and 8.5% for OnS-RO-GC. When resources are unconstrained (32MB), it is -1.0% for OnS-GC, 0.1% for OnS-DO-GC, -2.7% for OnS-EO-GC, and -1.8% for OnS-RO-GC.

5.2.2 *Adaptivity to heap sizes*

We next summarize the improvements enabled by our techniques in terms of code size and overall performance, for a range of heap sizes; these results indicate the adaptivity of our strategies. We focus on the best-performing combination of *What* and *When* strategies: online, sample-based profiling using an unloading trigger of GC invocation count (OnS-GC) for the fast JVM configuration, and OnS-DO-GC for the adaptive JVM configuration. Figures 6 (for OnS-GC) and 7 (for OnS-DO-GC) show the results. For both figures, the x-axis is heap size. The y-axis of the left graphs in both figures is the average code size normalized to the clean version; the y-axis of right graphs is the execution time normalized to the clean version.

Figure 6(a) summarizes the effect of OnS-GC on code size as the heap size grows from the minimum to 32MB. In this figure, we can see that when memory is limited, OnS-GC triggers more aggressive unloading, which results in code size reductions of 61% on average. When memory availability grows, the aggressiveness with which OnS-GC triggers unloading decreases quickly since fewer garbage collections are invoked. However, the separate startup strategy guarantees that even when memory is not critical, e.g., 32MB, dead startup code will be unloaded. The code size reduction using a 32MB heap is 43% on average.

This code size reduction enables significant execution time benefits while imposing very little overhead. Figure 6(b) shows the normalized execution time of OnS-GC strategy for different heap sizes. In this figure, we can see that when memory is constrained, code unloading can not only reduce the size of cached code, but also improve execution time by trading off GC time for compilation overhead. When memory grows, the execution time improvements gained decrease quickly because the GC overhead decreases and unloading becomes unnecessary. These results indicate that our framework and the OnS-GC strategy can adapt to dynamic heap memory availability.

Similar to Figure 6, Figure 7(a) summarizes the effect of OnS-DO-GC on code size and (b) depicts its impact on execution time, for the adaptive JVM configuration. Notice that for the adaptive configuration, the minimal heap size that a benchmark can run with is larger than that in fast configuration due to larger memory requirements for code optimization. As such, the curves in Figure 7 start from a larger initial heap size. On average, the code size reduction achieved by OnS-DO-GC is 43% for minimal heap size and 38% for 32MB; while the performance improvement is 10.3% and 0.1%, respectively.

6. RELATED WORK

Several code cache management techniques have been proposed in the area of dynamic binary translation [19, 2, 9, 6]. The technique most related to our work is *code pitching* used in Microsoft .NET Compact Framework [19]. In this framework, all code in the code buffer is discarded when the buffer exceeds some threshold. A similar strategy is used in the Dynamo [2] dynamic optimizer from HP. In Dynamo, when the cache fills, all fragments are “flushed” to free up space for new traces. Code pitching and cache flushing can both be configured using our *NP-CS* strategy. This simple strategy is chosen by these systems due to ease of implementation, low profiling overhead, or no linking problems. We found,

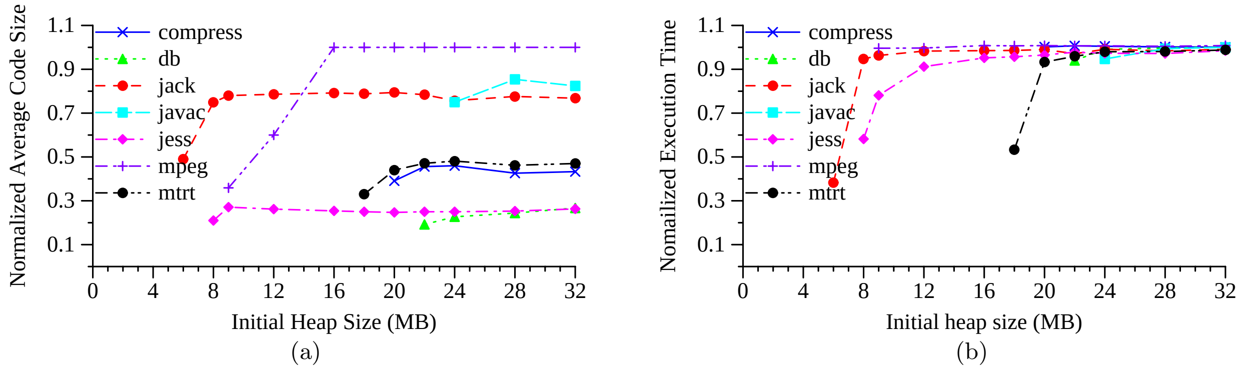


Figure 6: Summary of the improvements of OnS-GC strategy for Fast JVM configuration

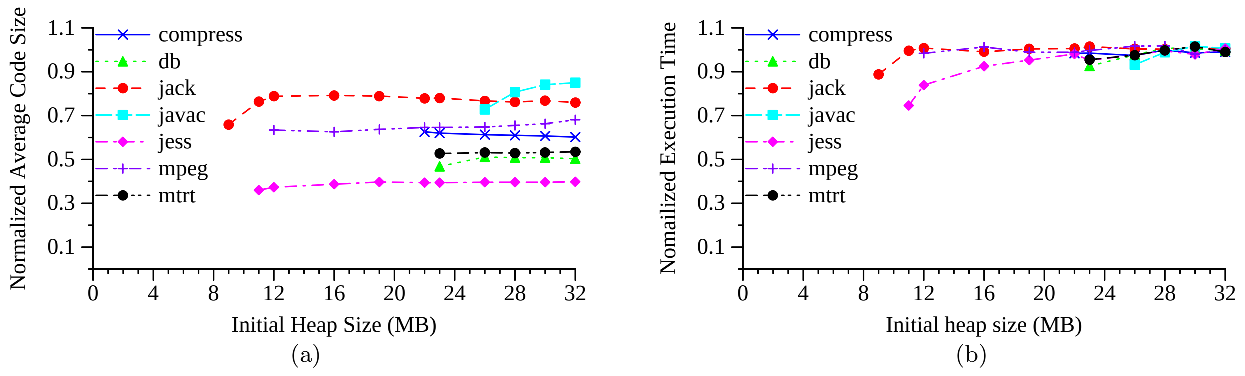


Figure 7: Summary of the improvements of OnS-DO-GC strategy for Adaptive JVM configuration

however, that in our target systems (JVMs), where cached code is commonly method-based, selective unloading of code using lightweight profiling techniques and a trigger that is able to adapt to changes in heap memory requirements can achieve better balance between execution performance and memory use. We plan to investigate partial method unloading, i.e., code region-based unloading, as part of future work.

Other dynamic binary translation systems use other forms of cache management. For example, DynamoRIO [6], uses an unbounded code cache by default. Users, however, can specify a limit to the code cache, and in this case, DynamoRIO uses a circular buffer mechanism similar to that proposed in [11]. The DAISY project, a multi-architecture VLIW emulator from IBM [9], employs a form of generational garbage collection to manage a large code cache (100MB or more). Similar generational cache management mechanism is investigated in [12] using DynamoRIO [6] and a generational cache simulator. The strategies that we describe in Section 3.3, e.g., DO and EO, handle optimized code separately and can be considered simple forms of generational cache management. We do not investigate other, more general, generational code cache management and circular buffer strategies in this paper.

Another related area is code size reduction in JVMs for restricted resource environments. The HotSpot JVM from Sun Microsystems [13] limits the size of compiled code by only compiling the hottest methods and interpreting all others. Other work employs *profile-driven deferred* compilation or optimization [5, 23] 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. Even in these "never cache cold methods" systems, code unloading techniques can also be used to manage "hot" methods.

7. CONCLUSIONS

In this paper, we first discuss the opportunity for dynamically unloading compiled code in JIT-based JVMs for mobile and embedded devices. We found that, for most benchmarks that 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 is consuming memory resources needlessly. As such, we developed a dynamic code unloading framework that can be integrated into any compile-only JVM to dynamically and adaptively unload code to relieve memory pressure in resource-constrained systems. In our framework, a code unloader uses information about the system memory status to determine when to initiate unloading and utilizes both online and off-line profile information to select methods for unloading.

We implemented our code unloading framework in the IBM JikesRVM and investigated a number of different unloading strategies. Through experimentation using both an unoptimized JIT configuration and an adaptive optimization configuration, we found that by adaptively unloading dead and infrequently used code, we can significantly reduce the memory consumed for native code. When this code is stored on a garbage collected heap, these reductions in code size translate into significant performance improvements when memory is highly constrained. Overall, our best strategies enable an average reduction in code size of 43% while introducing less than 1% overhead, when memory is unconstrained. When memory is highly constrained, our system reduces code size by 61% and execution time by 23% on average across benchmark programs and JVM configurations.

8. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for providing useful comments for the final version of this paper. This work was funded in part by NSF grant No. EHS-0209195, and an Intel/UCMicro external research grant.

9. REFERENCES

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

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

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

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

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

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1th Annual International Symposium on Code Generation and Optimization*, pages 265–275, March 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] L. Zhang and C. Krintz. Profile-driven Code Unloading for Resource-constrained JVMs. In *3rd International Conference on the Principles and Practice of Programming in Java*, June 2004.