JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 5, No. 3, Special issue: .NET Technologies 2005 Conference, April 2006

An Empirical Study of the Code Pitching Mechanism in the .NET Framework

David Anthony, Witawas Srisa-an, and Michael Leung
Department of Computer Science and Engineering
University of Nebraska-Lincoln, United States

The .NET Compact Framework is designed to be a high-performance virtual machine for mobile and embedded devices that operate on Windows CE (version 4.1 and later). It achieves fast execution time by compiling methods dynamically instead of using interpretation. Once compiled, these methods are stored in a portion of the heap called the code cache and can be quickly reused to satisfy future method calls. While the code cache provides a high-level of reusability, it can also use a large amount of memory. As a result, the Compact Framework provides a "code pitching" mechanism that can be used to discard the previously compiled methods as needed.
In this paper, we study the effect of code pitching on the overall performance and memory utilization of .NET applications. We conduct our experiments using Microsoft's Shared-Source Common Language Infrastructure (SSCLI). We profile the access behavior of the compiled methods. We also experiment with various code cache configurations to perform pitching. We find that programs can operate efficiently with a small code cache without incurring substantial recompilation and execution overheads.

Keywords: Just-in-time compilation, Java virtual machines, .NET CLR, code cache management

1 INTRODUCTION

In both .NET and Java execution systems, Just-In-Time (JIT) compilers have been used to speed up the execution time by compiling methods into native code for the underlying hardware [12, 22, 18]. JIT compilation has proved to be much more efficient than interpretation especially in execution intensive applications [11, 12, 22, 24]. In the Microsoft .NET Framework, a method is compiled prior to its first use. Afterward, the compiled methods are stored in the code cache for future reuse [16]. This code cache is located in the heap region .

The size of the code cache can be increased or decreased depending on the program's behavior. For example, in the default configuration of the *Shared-Source Common Language Infrastructure* (*SSCLI*), frequently referred to as *Rotor* [14, 18], the initial code cache size is set to 64 MB. Once the accumulation of compiled methods reaches this size, the system can choose either to increase the code cache size or to keep the same size and free all the compiled methods not currently in scope (referred to as throw-away compiling [3] or code pitching [18, 16]). There are two possible overheads of the "code pitching" mechanism [18, 16]— the overhead of traversing through all the compiled methods and the overhead of recompiling methods after pitching. However, pitching

Cite this article as follows: : *An Empirical Study of the Code Pitching Mechanism in the .NET Framework*, in Journal of Object Technology, vol. 5, no. 3, April 2006, Special issue: .NET Technologies 2005 Conference, pages 107–127,
http://www.jot.fm/issues/issues_2006_04/article5

provides a means to maintain a small code cache as the memory is periodically reclaimed.

Currently, code pitching is employed in the .NET Compact Framework (CF), which is used to develop applications for smart devices with limited memory resources [16]. Such devices include smart phones, Pocket PCs, and embedded systems running Windows CE. In these devices, a pitching policy can play a very important role since it can determine the amount of memory footprint for the code cache. If pitching occurs infrequently, the code cache would occupy a large amount of memory. If pitching occurs too frequently, a large number of methods would have to be recompiled. The goal of this paper is to take a preliminary step to study the effect of pitching on the overall performance and memory utilization of .NET applications. To date, there have been a few projects that investigate the recompiling decision and method unloading in Java [24, 23, 4]. However, they are implemented into a virtual machine that does not support pitching. With the SSCLI, we have an opportunity to study the mechanism that has been built by a major software maker as a standard feature. Our work attempts to study two important research questions. They are:

RQ1: What are the basic behaviors of the compiled methods?—We investigate the access behaviors, compilation frequency, and commonly used metrics such as size and the number of methods.

RQ2: Can we improve the overall performance and memory utilization by manipulating the code cache configuration?—We experiment with multiple code cache sizes and investigate the impacts of utilizing different cache size enlargement policies.

The remainder of this paper is organized as follows. Section 2 introduces related background information. Section 3 describes our challenges and research questions in detail. It also describes the methodology and constraints used to perform the experiments. Section 4 discusses the experiments and results conducted with regard to the research questions. It also contains the detailed analysis of our findings. Section 5 discusses our finding in details. Section 6 presents the future work. Section 8 discusses prior research work in this area. The last section concludes this paper.

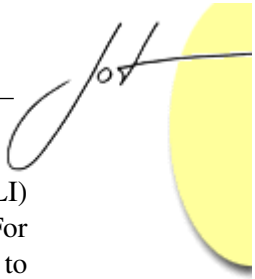
2 BACKGROUND

This section discusses background information related to this work.

Shared-Source Common Language Infrastructure (SSCLI)

The main objective of the CLI is to allow programmers to develop component-based applications where the components can be constructed using multiple languages (e.g. C#, C++, Python, etc.). ECMA-335[1] (CLI) standard describes "a language-agnostic runtime engine that is capable of converting lifeless blobs of metadata into self-assembling, robust, and type-safe software systems" [18]. There are several implementations of this standard that include Microsoft's *Common*

[1] European Computer Manufacturers Association.



Language Runtime (*CLR*), Microsoft's Shared Source Common Language Infrastructure (SSCLI) [14], Microsoft's .NET Compact Framework, Ximian's Mono project [15], and DotGNU [8]. For this research, we use the SSCLI due to the availability of the source code. Moreover, it seems to be the most mature implementation when compared to the Mono or DotGNU projects.

SSCLI is a public implementation of ECMA-335 standard. It is released under Microsoft's shared source license. The code base is very similar to the commercial CLR with a few exceptions. First, the SSCLI does not support ADO.NET and ASP.NET which are available in the commercial CLR. ADO.NET is a database connectivity API and ASP.NET is a web API that is used to create Web services. Second, the SSCLI uses a different *Just-In-Time* (*JIT*) compiler than the commercial CLR uses. The latter provides a more sophisticated JIT compiler with the ability to pre-compile code. However, the commercial CLR does not support code pitching. Notice that both implementations of the CLI adopt JIT compilation and not interpretation mode as in some earlier Java Virtual Machine implementations [19]. Third, it is designed to provide maximum portability. Thus, a software layer called Platform Adaptation Layer (PAL) is used to provide Win32 API for the SSCLI. Currently, the SSCLI has been successfully ported to the Windows, FreeBSD, and MacOS-X operating systems.

One of the major runtime components related to this work is the Just-In-Time (JIT) compiler. It is used to compile methods within components into the native code for the underlying hardware [22]. The JIT compiler also ensures that every instruction conforms to the specification by the ECMA standard. Once compiled, these methods reside in the code cache which is located in the heap memory. Instead of recompiling a method each time it is called, the native code is retrieved from the code cache [16]. When more memory is needed by the system or when a long running application is moved to the background, the methods in the code cache are "pitched" to free up memory [16, 18].

Code Pitching Mechanism

There are three important variables in the code pitching mechanism: reserved cache size, target cache size, and maximum cache size. The target cache size is initially set to be the same as the reserved cache size, e.g. 64KB. The maximum cache size is set to be very large to allow the most flexible growth. By default, the SSCLI sets the maximum cache size to 2GB. The execution engine initializes the code cache by allocating 8KB. As program execution continues, additional heap space is allocated to the code cache in 8KB increments as needed to store the compiled methods. The total size of the allocated heap space is called the committed code cache size. As the committed code cache size approaches the reserved code cache size (e.g. 256KB), the allocator will decide whether to allocate more heap space beyond the current reserved cache size or pitch all unused methods. Notice that once the target cache size is reached, the reserved size is adjusted and used to determine the actual cache size. For example, if the initial cache size is 256KB (i.e. the initial value of target and reserved cache sizes) but the system presently needs 512KB to store compiled methods, the reserved size would be adjusted to 512KB while the target cache size remain the same at 256KB.

If the reserved size is less than the maximum code cache size and the existing pitch overhead

(amount of time spent on pitching) is over the acceptable maximum (default 5ms), the allocator will attempt to increase the code cache size. Notice that 5 ms is chosen as the default value in the SSCLI to avoid excessive pitching. During this attempt, if the needed memory is greater than the reserved size, less than the maximum cache size, not at the pitch trigger point, and pitch overhead is too high, it will expand the committed code cache size and the reserved size. Otherwise, it will pitch all the unused compile methods [2]. If there is still insufficient memory after pitching, the code cache size and the reserved size will be increased until enough memory is available. If at any point during the execution, the number of compiled methods reaches the pitch trigger, pitching occurs regardless of other cache conditions.

Currently, code pitching is used in the .NET Compact Framework, which is built for embedded devices. Obviously, it is very important to strike a good balance of memory usage and performance overhead since such devices have a very limited amount of memory. In addition, the Compact Framework is often used in Windows CE that has the maximum virtual process space of only 32 MB. Thus, the amount of code cache has to be small enough to work in this computing environment but yet big enough to provide efficient compilation of methods.

3 EMPIRICAL STUDY

As stated earlier, the behavior of compiled methods in the .NET framework has yet to be studied. In order to design an efficient pitching policy, a thorough understanding of the behavior is needed. The current lack of this knowledge has led us to the first research question.

RQ1: ***What are the basic behaviors of compiled methods?***

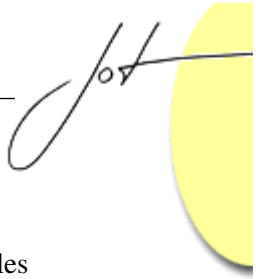
If a large number of methods is frequently used, then it may not be suitable to pitch the code cache frequently. Our contribution is to profile the access behavior of compiled method so that an efficient pitching decision can be made. We conjecture that a significant performance gain or reduction in memory usage can be obtained by utilizing different pitching policies. Thus, our second research question is:

RQ2: ***Can we improve the overall performance and memory utilization by manipulating the code cache configuration?***

In the default configuration of the SSCLI, the policy is to explore other possibilities before pitching is considered. This may not be the most optimal approach especially for the Compact Framework in which the amount of memory available in a system may be limited. Our contribution is to identify a cache size and suggest pitching policies that would result in small cache footprint and minimal compilation overhead.

In terms of experimental platform, we conducted our study on an AMD 64 workstation with 1 GB of memory and running Windows XP Professional.

[2] This is in contrary to [18] in which the authors mentioned that the whole code cache is unloaded. We investigated the source code and found that only methods that are not currently accessed are unloaded.



Variables and Measures

The JIT compiler relies on several variables to control cache size and pitching. These variables are used to control the compiler when to pitch, maximum and minimum cache size, and cache growth characteristics. As will be described in the next subsection, we use existing benchmark programs written in C# to perform our experiments. It is worth noting that we fix the heap size to 800KB (the default value set by the SSCLI). In doing so, we can be certain that any changes in performance are not due to different heap sizes but mainly the changes in code cache sizes.

Throughout the experiment, we monitored the following variables. They provided useful insight into the operation of the JIT compiler, specifically, its caching mechanism.

- *Number of pitch events*
  When the compiler removes compiled code from the cache it is called a pitch event. Pitching will preserve methods that are currently in use (in scope), but will remove the rest.

- *Number of recompilations*
  After a method has been pitched, each time it has to be compiled again is called a recompilation. A method could be pitched and recompiled multiple times.

- *Number of distinct methods*
  This is the number of distinct methods that have been compiled. The number of distinct methods does not include recompilations and does not consider whether the method has been pitched or not.

- *Committed code cache size*
  The amount of heap space requested from the system to store code is called the committed code cache size. The compiler asks for heap in increments of 8KB.

- *Code cache usage*
  Code cache usage is the actual amount of memory used to store compiled methods at a given time.

To address RQ1, we monitor the basic behavior of compiled methods. Our goal is to derive at two important performance metrics based on the results of variables above:

1. compilation frequency—we monitor how often methods are compiled and recompiled.

2. concentration of compiled methods—we monitor which part in the execution methods are compiled the most.

We also observed the average size of compiled methods and compared them to the sizes of typical objects. In order to perform our experiments, we created an environment in which the amount of memory is similar to a typical Java embedded device. To do so, we set the initial code cache size to 256KB. However, we would allow the SSCLI to enlarge the code cache as necessary.

Application	Minimum (bytes)	Maximum (bytes)	Average (bytes)	Standard Deviation	Number of Methods
LCSC	52	27024	1044.93	2587.04	632
AHC	52	6320	317.04	474.21	514
PNG Decoder (PNGD)	52	6320	324.71	486.52	556
CLisp	52	44008	425.66	1424.96	324
SharpSATbench (SAT)	52	6772	355.35	597.71	412

Table 1: Basic characteristic of the compiled methods in our benchmarks.

Application	% of space needed in the code cache				
	15%	30%	45%	60%	75%
LCSC	0.29%	0.31%	0.33%	0.36%	0.41%
AHC	0.005%	0.006%	0.009%	99.95%	99.96%
PNG Decoder	0.24%	0.34%	0.39%	0.52%	99.87%
CLisp	7.98%	12.17%	16.10%	36.33%	82.89%
SharpSATbench	0.04%	0.06%	98.28%	98.30%	99.95%

Table 2: Code-cache usage at different points in execution time.

To address RQ2, we went a step further and prevented the SSCLI from enlarging the code cache. The goal of our experiment is to observe the behavior of compiled methods under a hard-limit and explore different code cache configurations to improve the overall performance. We also compared the execution time among different configurations that result in different number of pitch events.

Benchmark Programs

To address our research questions, we need a set of programs that compile a large number of methods. In addition, we must be able to manipulate the way these programs are operated. As of now, there are very few benchmark programs available for the .NET platform. We have gathered 3 different programs that compiled a reasonable amount of methods (over 1000). We also want to observe how the code cache would perform during the execution of smaller applications. Therefore, we also experiment with using the classic HelloWorld and Adaptive Huffman Compression to get some insights on how many methods are needed to execute such programs. To our surprise, HelloWorld still requires over 300 compiled methods. This section describes the experimental objects:

- **LCSC**
  This benchmark is based on the front end of a C# compiler. The program parses a given C# input file with a generalized LR algorithm. The benchmark is available from Microsoft's research web site [13], along with the inputs that were used in performing the analysis.

- **AHC**
  This program uses an adaptive Huffman compression algorithm to process files. For this

program there were three separate inputs for use as test cases. This benchmark is also available from Microsoft's research web site [13].

- **PNG Decoder**
  This program shows "how fast a Java implementation can decode a PNG photo image of a typical size used on a mobile phone." [9]

- **CLisp Compiler**
  This is a small compiler that converts a Lisp source file to an executable. The compiler was used to compile two sample source files, a Fibonacci series generator and a numerical sorting algorithm. This compiler is found in the sscli/compilers/clisp directory.

- **SharpSATbench**
  SharpSATbench is a "clause-based satisfiability solver where the logic formula is written in Conjunctive Normal Form (CNF)." [13]

4 RESULTS

In the following subsections, we present the results of our experiments that answer two research questions proposed in Section 3.

RQ1: Access Behavior

In this section, we discuss the basic behavior of these compiled methods. The issues that will be discussed in this section include the number of compiled methods in each application, the number of methods recompiled, and the size of the compiled methods. Table 1 depicts the size information of compiled methods in our benchmark programs.

It is worth noticing that typical objects in object-oriented languages such as Java and C# only have an average object size of less than 100 bytes [7, 21]. However, the average size of the compiled methods in each application range from 300 bytes to over 1100 bytes. It is also worth noting that the smallest size for a compiled method is 52 bytes (see Section 5 for more information). This is true across all applications. For the largest size, a method can be as large as 44KB bytes. We also conducted experiments with no pitching and found that 1.4MB of memory is needed to store the compiled methods in LCSC.

In our experiment, we first study the code cache usage of every application. We set the cache size to be large enough so that pitching does not occur. With the proposed set of benchmarks, the size is set to 2 MB. We then monitor the fraction of the code cache usage at different points in the program's execution. For example, LCSC requires 1.4 MB of space to store all compiled methods. When the program consumes 15% of all the needed cache space or 212KB, we observe the percentage of execution. In this case, the program has only completed 0.29% of the total execution time (see Table 2). It is worth noting that in three out of five applications, over 45% of all the space needed for the code cache are consumed with in the first few percent of the execution time.

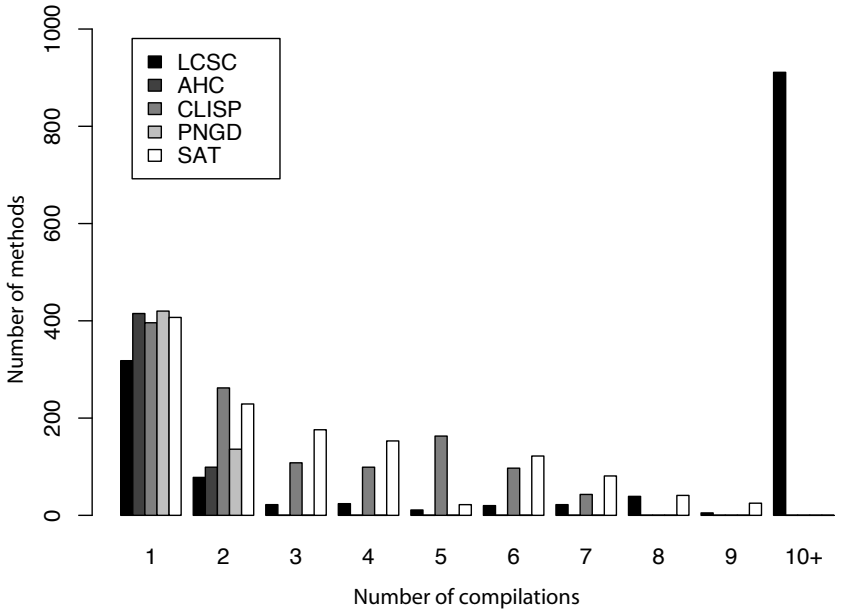


Figure 1: Distribution of compiled methods based on the number of compilations (128KB cache size).

We also monitor the distribution of methods based on the number of compilations. We set the code cache size to 128KB to emulate embedded devices environment and to induce some pitch events. We find that in two applications AHC and PNGD, no methods are compiled more than twice. However, in larger applications, such as the compilers and the constraints satisfaction program, methods are compiled multiple times. Notice that CLisp requires at most 7 compilations while SAT and LCSC require 9 and more than 10 compilations, respectively. Since most of these applications execute repetitive tasks, many compiled methods are reused. If pitch events are forced to occur more often, these programs may need to have methods recompiled more frequently. Figure 1 illustrates our findings.

To investigate the number of recompiled methods, we set the code cache size to 256KB to force pitching. We find that about 70% of recompilations occur during the first 10% of execution time (depicted in Figure 2) in all benchmark programs. The remaining 30% of compilation occur during the remaining 94% of execution time. Thus, many of these methods are short-lived but during their lifetimes seem to have many accesses. This is similar to typical objects where the majority are short-lived [10, 20]. This behavior may provide an opportunity for optimization by dynamically adjusting the cache size as needed. For example, the cache size can initially be set to be larger and then reduced after the first 10% of execution. We are currently experimenting with this approach.

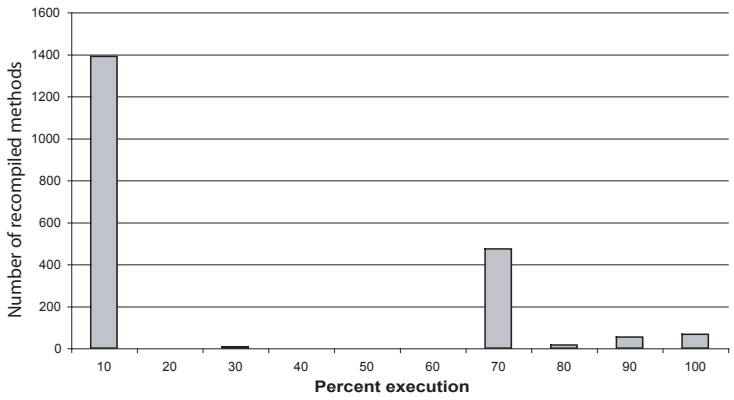


Figure 2: Distribution of recompiled methods over the execution time (all applications combined using 256KB cache size).

In summary, we find that compiled methods have the following behavior:

- The average size of a method is much larger than the average size of a typical object.
- Even the simplest applications require at least 300 methods to execute.
- In larger programs, a large number of methods are reused. This conclusion is based on the fact that large programs recompile a large amount of methods when the cache size is small and pitching occurs frequently.
- Methods are compiled more frequently toward the beginning of a program execution.

RQ2: Optimizing Code-Cache Configuration and Pitching Policy

In this section, we will apply different pitching policies to LCSC and monitor the differences in the runtime behavior. We choose LCSC because it accesses a large number of methods and requires the largest number of pitch events. In the SSCLI, there are two ways to set the size of the code cache. The first method (which shall be referred to as the *soft limit approach*) is to set the target code cache to a certain size (e.g. 256KB). This however, is not the highest possible value. When the volume of compiled methods reach 256KB for the first time, the system will pitch all methods that are not reachable, but it will also consider whether to increase the target cache size.

Typically, the cache size is doubled. Thus, the next pitch event will occur when the accumulation of methods in the code cache approaches 512KB. The second method (shall be referred to as the *hard limit approach*) is to set the initial code cache size to be the limit. Notice that the limit must be big enough to contain the initial method working set that can initialize the application. If the cache size is too small to contain all methods during initialization, the program may crash. Figure 3 depicts the pitch events using the soft limit approach. The initial target code cache is set to 256KB.

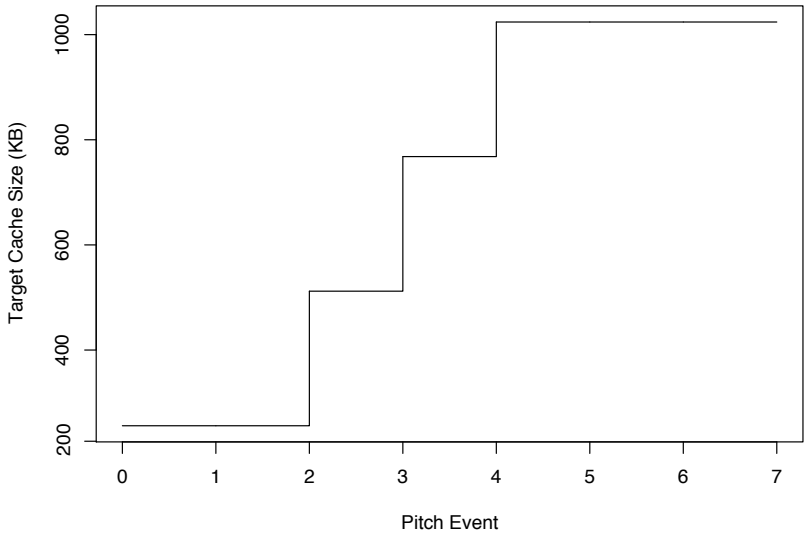


Figure 3: Monitoring pitch events in LCSC using the soft limit approach.

Figure 3 illustrates the basic behavior of code cache expansion in the soft limit approach. The x-axis in the figure represents all the pitch events that occur in the system. In this example, we have 7 pitch events throughout the execution of LCSC. Table 3 depicts the number of pitch events in all applications with different target cache sizes (256KB, 512KB, 1MB, and 2MB). It is worth noting that the benefit gained through this approach is the reduction of the number of pitch events during the initial execution period. For example, by increasing the initial cache size from 256KB to 512KB, the number of pitch events decreases by two in LCSC. These two events occur during the first 5% of the execution.

Figure 4 depicts the number of methods that are recompiled by applying the soft limit approach in which the cache size can be increased as needed. Notice that there are more methods recompiled after the later pitch events (4 to 6). This is corresponding to Table 2 as methods are compiled during the early part of the execution. As we continue to pitch late into the execution, the methods that were compiled and have recently been pitched are still being invoked and must be recompiled.

It is worth noting that the initial target size can greatly affect the number of pitch events in a

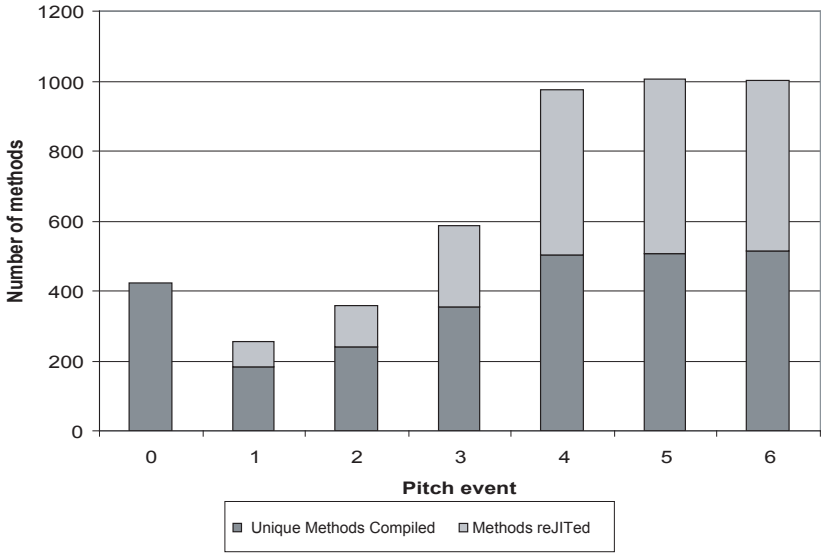


Figure 4: Ratios between new methods and recompiled methods based on pitch events.

Applications	256k	512k	1024k	2048k	4096k	8192k	16384k	65536k
LCSC	7	5	3	0	0	0	0	0
AHC	0	0	0	0	0	0	0	0
PNGD	0	0	0	0	0	0	0	0
CLisp	2	1	0	0	0	0	0	0
SAT	2	1	0	0	0	0	0	0

Table 3: The number of pitch events with different code cache sizes.

system. This is because the first pitch event will take longer to occur with larger cache size. As shown in Figure 2, a majority of repeated invocations occurs within the first 10% of execution. Thus, a larger initial cache size is more advantageous because more reuse occurs at the beginning.

Figure 4 initially appears to be contradicting Figure 2 as the volume of recompiled methods do not become significant until the fourth pitch event. However, we find that 4 out of 6 pitch events occur in the first 3% of execution. The fifth event occurs around 33% and the last event occurs at the 80%. Thus, most of the recompilation events occur during the initialization of the system.

Similar to the *soft limit approach*, we investigate the number of distinct methods and previously used methods that have to be compiled after each pitch event using the hard limit approach. Figure 5 illustrates the number of pitch events occurring when this approach is used. When the code cache size is limited to 128KB, we have nearly 25000 pitch events. As expected, the number of pitch events decreases as the code cache size is enlarged (256KB and 512KB, respectively). We also discover that when the code cache is set to 1024KB, there are only 4 pitch events (not in the figure). Figure 5 also compares the number of distinct methods to the number of recom-

piled methods in the three configurations of code cache (128KB, 256KB, and 512KB). In most instances, there are no distinct methods created during the execution—only at the beginning and at the end.

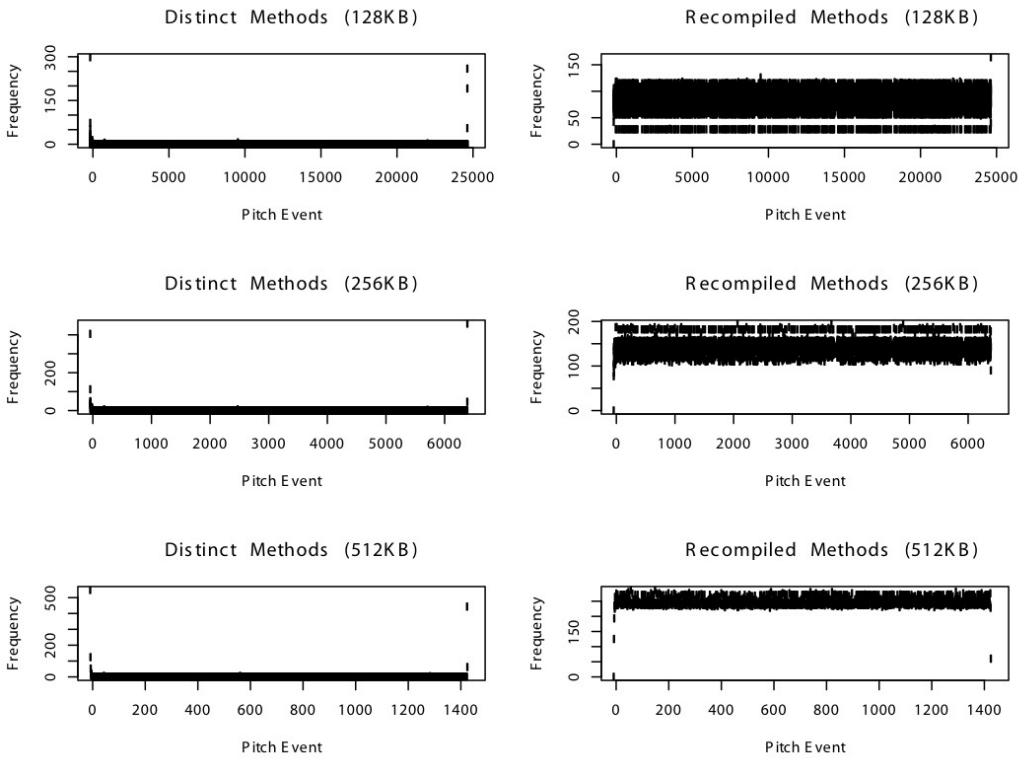


Figure 5: New methods v. recompiled methods in LCSC based on pitch events using the hard limit approach. Note that the y-axis represents the number of times that the two types of methods, distinct and recompiled have been compiled.

On the other hand, a small code cache in applications that invoke a large number of methods can cause excessive pitching, as in the cases of 64KB to 256KB cache sizes in the hard limit approach (see Figure 6) and significant runtime overheads. LCSC in our experiment can take as much as 35 times more execution time than the soft limit approach when the code cache is set to 64KB. We also find that in all applications except LCSC, the code cache requirements are moderate. Thus, the number of pitch events does not increase significantly. For example, adjusting the code cache size in AHC, PNGD, SAT, and CLisp has very little effect on the execution time (see Figure 7). As a reminder, these applications were executed using the same initial heap size (800KB). Therefore, the differences in performance are mainly due to the changes in code cache size. With the observation, we conjecture that pitching may be used to reduce the memory footprint without incurring a substantial amount of overhead in memory constrained systems. We will conduct experiments to further validate our conjecture as part of future work.

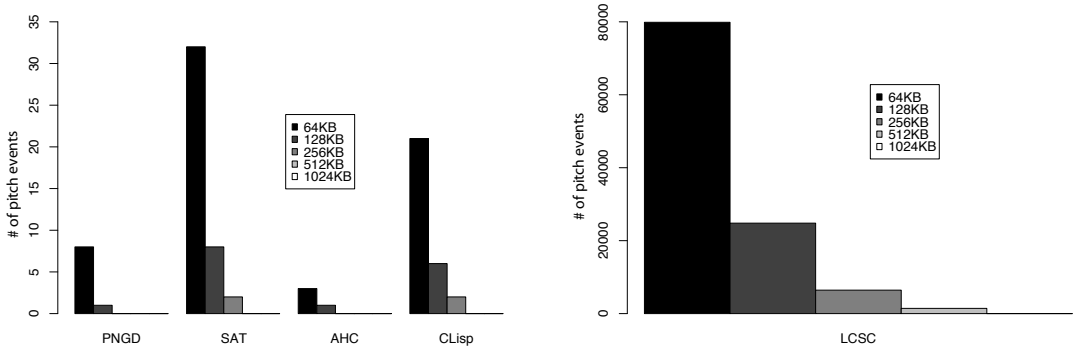


Figure 6: Pitching overhead when the hard limit approach is used.

Figure 8 depicts the usage of code cache as LCSC is executed. The x-axis represents the percentage of execution completed and the y-axis represents the amount of memory in the code cache used by the program. It is worth noting that with 256KB initial cache size using the soft limit approach, the size of the code cache increases to 1024KB within the first 3% of execution. However, it will take another 30% of execution to accumulate the compiled methods that would result in another pitching. In this situation, it may not be necessary to increase the cache size from 768K to 1024K. In addition, after the pitch event at 33% of execution time, the next pitch events do not occur until 81 %. One possible improvement to the pitching policy is to reduce the cache size after the programs are fully initialized. This may result in a few more pitch events but a significant reduction in memory usage may also be obtained.

Figure 9 depicts the usage of code cache for LCSC with 1024KB cache size applying the hard limit approach. It is worth noting that there are no pitch events at all until after 2.25% of execution. The figure also shows that after the first pitch event, there are only two more pitch events at 33 and 81% of execution. As a reminder, this is similar to the number of pitch events in Figure 8 after 4% of the program has been executed. Thus, a policy that favored a larger cache size clearly reduces the number of pitching activities during the initial state of execution and would be effective in memory-constrained environments.

In summary, we conclude that the following policies may be used to improve the pitching performance.

- Moderate pitching activities have very little effect on the overall performance of the system. However, excessive pitching can incur a large amount of overhead. Thus, a policy that favors reducing memory usage over a moderate increase in pitching activity would be effective in memory-constrained environments.

- Larger initial cache sizes can significantly reduce the number of pitch events during the program initialization. Thus, the policy should allocate a large enough cache at the beginning.

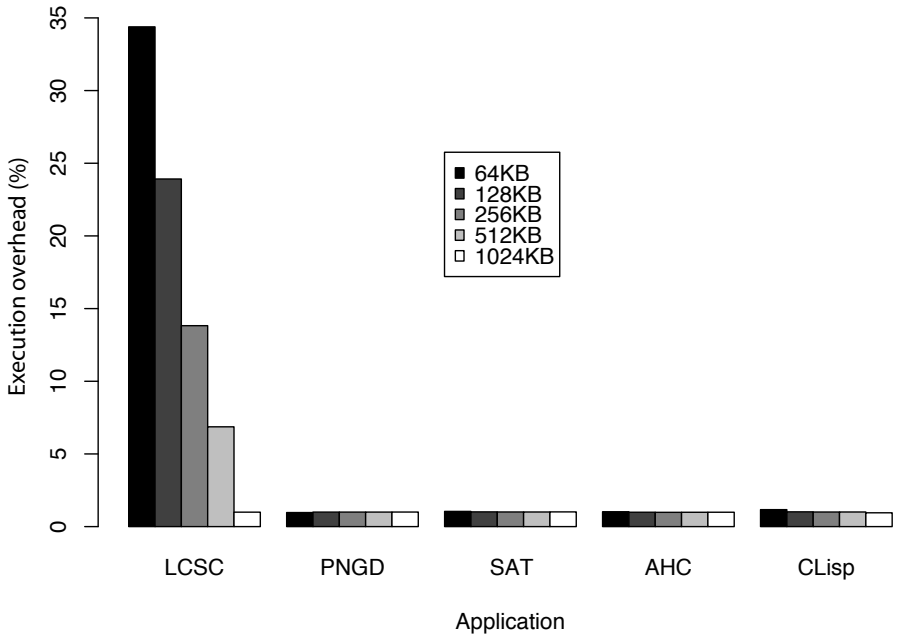


Figure 7: Execution time overhead in the hard limit approach.

- Once stabilized, the system compiled fewer methods which means that we can potentially reduce the cache size at the expense of more pitching activities. However, the number of pitch events should be moderate and not result in a substantial run-time overhead. Thus, the policy should include reducing the cache size after the initialization phase.

5 CLASSIFICATION OF FREQUENTLY RECOMPILED METHODS

As shown in Figure 5, a large portion of methods are compiled and recompiled over and over again. In this section, we further identify the types of methods that exhibit this behavior. This insight may open more avenues to further optimize code cache management strategies.

We conducted experiments by setting the maximum code cache to 64KB to force regular pitching. We then identified a subset of methods that are called at least once by all five benchmarks. We investigated the distribution of such methods and the frequency that these methods are invoked. For example, the distribution for *call by all* is calculated by observing the number of distinct methods that are invoked by all five applications. In our experiments, there are 85 methods that fall into this criteria. We then took the summation of all methods in all applications (last column in Table 1) and the sum is 2438 in this case. The distribution of *call by all* is calculated by $\frac{5\times85}{2438}$ or 16%.

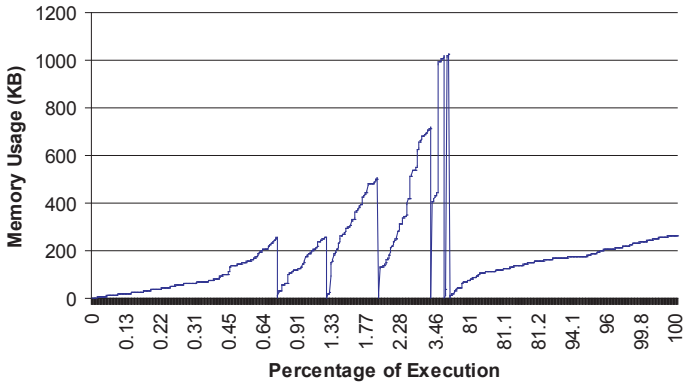


Figure 8: Code cache usage for LCSC (256KB). Notice that the scale of the x-axis is not linear.

We then investigate the invocations frequency of these methods. There are over 3.5 million total invocations in all applications when the maximum code cache is set to 64KB. We find that the methods invoked by all applications account for 23.70% of invocations. This is 7% higher than the distribution depicted in Figure 10, which translates to heavier usage of these methods. We also find that all of these methods belong to the *System* library. We then combine the methods invoked by 5 and 4 applications and find that they only account for 24.90%, which means that methods used in four applications are not as heavily invoked as ones used by all applications.

6 FUTURE WORK

Better benchmarks that utilize more methods and force the execution engine to pitch more frequently especially for larger cache sizes are needed. In addition, the benchmarks used in this experiment do not demonstrate the diversity of applications the typical end user runs. More practical benchmarks are definitely needed to better simulate a real world system. On the other hand, some of the chosen experimental objects compile reasonable amounts of methods.

With that said, many of our results derive from experimenting with these few benchmark programs. Thus, our conclusions or suggestions should not be viewed as generalized ones. Instead, they should be viewed as potential solutions to improve the performance of the code-pitching mechanism in the SSCLI and .NET Compact Framework. Obviously, experiments with more benchmark programs are needed.

Future work will focus on four primary goals. The first goal is to develop better benchmarks in order to better simulate real world uses of the SSCLI. These benchmarks should focus on what a more average user would be expected to run. New benchmarks should have networking and other communication methods that are inherent to their proper execution.

The second major goal is to develop a better code pitching mechanism that selectively re-

Figure 9: Code cache usage (1024KB). Notice that the scale of the x-axis is not linear.

moves code from the cache, as opposed to the all or nothing approach taken in the current Rotor implementation. This improved collection mechanism will likely correlate method usage and size to enable the pitching mechanism to make a better decision as to its usefulness in the future. In addition, the current Rotor implementation does not decrease the size once the code cache has been expanded. We plan to investigate the performance gain of decreasing the cache size after the initial phase of execution.

The third goal is to further investigate the affects of code cache management strategies on performance metrics not studied in this paper such as start-up time. Typically, applications for mobile embedded devices are user-interactive. This means that start-up time plays a very important role in enriching users' experience. Once the programs have started, the overall performance is less noticeable by the users as most of the execution time will be spent on input/output. It is possible to tune the code cache manager to minimize start-up time.

Our last goal is to utilize an on-line profiling to create application-specific code cache management policies that perform as well as those created using off-line analysis [23]. It will determine the proper size, the proper increment, and the proper increment frequency dynamically. Note [23] introduced these issues but did not study them. The system initializes a small code cache and lets it quickly grow to a sufficient size to contain all methods needed to start the program. In this approach, no methods are unloaded during this period. However, when the number of methods needed has declined (e.g. after 10% of execution), the methods are unloaded and the code cache is reinitialized to a small size. During this time, the code cache grows very slowly or does not grow at all as there are only a few methods compiled and used. Toward the end when the number of methods compiled dramatically increases, the growth rate of the code cache is once again adjusted to match the applications' needs. Thus, the system will allow the application to start faster but will maintain a very small cache size throughout most of the execution.

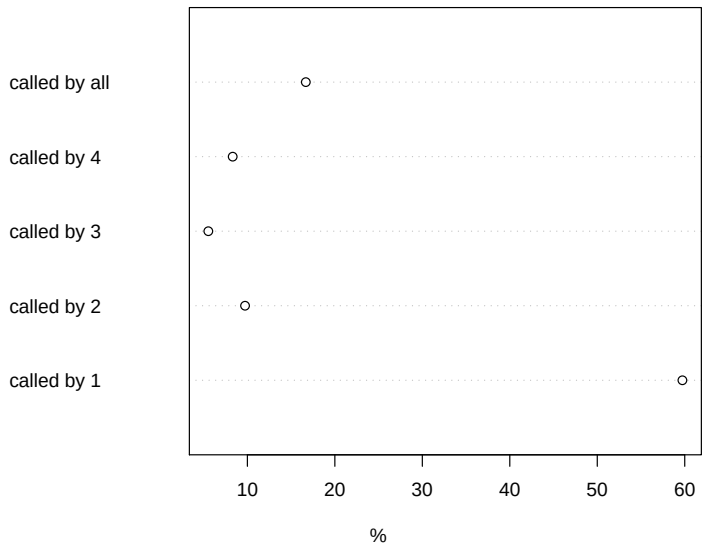


Figure 10: Distribution of methods based on the number of applications that used them.

7 RELATED WORK

In [2], a multi-level recompilation technique was introduced as part of the Jalapeño Virtual Machine. The basic idea is to use a non-optimized compiler to compile a method the first time it is called. During the execution, the virtual machine would keep track of all the ”hot” methods (frequently invoked) and recompile them with higher optimization levels.

Currently, the code pitching mechanism in .NET compact framework as well as the SSCLI discards all compiled methods that are not in scope. The code cache itself is separate from the main heap memory region. This type of strategy is often referred to as “flush when full” [17]. One of the first systems to use the approach to managed code caches or translation-cache is *Shade*—an instruction set simulator for execution profiling [5, 6]. In this system, *translation cache* (*TC*) is a separate memory area used to stored translations. When the TC is full, the system flushes all the entries in the TC. The authors claimed that flushing is advantageous over other approaches because methods' chaining makes selective freeing tedious. They initialized the TC to a large size to minimize the number of flushing [5].

Recent efforts by [24, 23] mainly investigated the performance of a code unloading mechanism in virtual machines that store both compiled methods and regular objects in the same heap space. Specifically, they identified what methods to unload and when to unload. They used off-line and on-line profiling techniques to improve the performance. They reported a code size reduction of up to 61%. One interesting observation that 70% of compiled methods are dead within the first

10% of execution time was also reported and confirmed by [1], in which .NET applications were studied. However, [23] reported difficulties in developing mechanisms to detect a point in time that represents 10% of execution. Therefore, they used two triggering techniques: unload every 10 GC cycles and unload every 10 seconds.

In addition, they also conducted a preliminary study of a similar scheme used in the .NET compact framework that stores compiled method bodies in a separate code cache; unloading is performed when the cache is full. Their experiment consisted of initializing the code cache size to 64KB and enlarging it by 32KB every 10 unloading cycles. They anticipated two difficulties that were not studied: dynamically determining the size of the code cache and determining how often and by what increments to grow. Our proposed activity further investigates and improves the performance of this code cache management scheme.

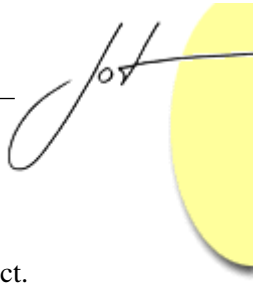
It is worth noticing that they reported in their earlier work that native IA32 code tends to be 6 to 8 times larger than the bytecode written in Java. They also reported that on average 61% of compiled methods are no longer accessed after the first 10% of execution [24].

8 CONCLUSIONS

We have performed experiments to demonstrate the effects of code-pitching on the overall performance of .NET applications. We find that the compiled methods have the following properties. First, they are much larger than typical objects with averages ranging from 300 bytes to 1100 bytes. Second, a large number of methods are repeatedly accessed. Third, these accesses often occur within the first 6% of execution time. Fourth, methods are compiled prolifically. At 64KB cache size, the applications compiled over 3.3 million methods.

Based on the above finding, we conduct multiple experiments using different code cache configurations. First, we set the initial cache size to 256KB. We allow the system to expand the cache as needed. By setting a larger initial cache size (e.g. 512KB versus 256KB), we can reduce the number of pitch events by 29% (from 7 events to 5 events). Having a large initial cache size can be advantageous since most of the method reuse occurs within the first few percent of execution. Larger cache size may defer pitching and promote more reuse. Second, we also find that excessive pitching can cause significant overhead. However, a moderate amount of pitching barely incurs overhead. In our experiment we find that when the cache size is set at 2MB, no pitching occur. However, if we reduce the cache size by half, only 3 pitch events would occur (in LCSC) but the overall execution time increases only slightly. Thus, we conclude that a well designed pitching policy can greatly reduce the amount of the code cache footprint without incurring substantial overheads. In addition, a policy to reduce the code cache size after the initial state can also be employed to further reduce the code cache footprint.

We also find that there is a significant number of methods that are shared by all applications. These methods account for 16% of all methods. This methods are invoked prolifically and when the cache size is set to 64KB, they account for nearly 24% of all method invocations. By looking at the type of these methods, they are all part of the *System* library. One possible optimization based on this insight is not to pitch these methods to minimize recompilation efforts.



9 ACKNOWLEDGEMENT

We would like to acknowledge Tyson Stewart for helping during the initial phase of this project. This project is partially supported by the National Science Foundation under grant CNS-0411043, University of Nebraska UCARE program, and University of Nebraska Layman's Award.

REFERENCES

[1] D. Anthony, M. Leung, and W. Srisa-an. To JIT or not to JIT: The effect of code-pitching on the performance of .NET framework. In *Proceedings of 3rd International Conference on .NET Technologies*, pages 165–173, Plzen, Czech Republic, May 30-31, 2005.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, New York, NY, USA, 2000. ACM Press.

[3] P. Brown. Throw-away compiling. *Software Practice and Experience*, 6:423–434, 1976.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[5] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, New York, NY, USA, 1994. ACM Press.

[6] R. F. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. Technical report, Mountain View, CA, USA, 1993.

[7] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1999.

[8] DotGNU Project. DotGNU Project - GNU Freedom for the Net. http://www.dotgnu.org, 2006.

[9] Embedded Microprocessor Benchmark Consortium. Grinderbench. http://www.grinderbench.com/.

[10] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, 1998.

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

[12] A. Krall and R. Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[13] Microsoft. Ben's CLI benchmark. http://research.microsoft.com/

[14] Microsoft. Microsoft shared source CLI. http://www.microsoft.com/downloads.

[15] Mono. What is Mono? http://www.mono-project.com/Main_Page, 2006.

[16] S. Pratschner. information available from http://weblogs.asp.net/stevenpr/archive/2004/07/26.aspx.

[17] J. Smith and R. Nair. *Virtual Machines : Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[18] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly and Associates, 2003.

[19] Transvirtual. Kaffe virtual machine. http://www.kaffe.org.

[20] D. Ungar. The design and evaluation of a high performance Smalltalk system. *ACM Distinguished Dissertations*, 1987.

[21] Q. Yang, W. Srisa-an, T. Skotiniotis, and J. M. Chang. Java virtual machine timing probes: A study of object lifespan and garbage collection. In *Proceedings of 21st IEEE International Performance Computing and Communication Conference (IPCCC-2002)*, pages 73–80, Phoenix Arizona, April 3-5, 2001.

[22] F. Yellin. Just-in-time compiler interface specification. ftp://ftp.javasoft.com/docs/jit_interface.pdf, 1999.

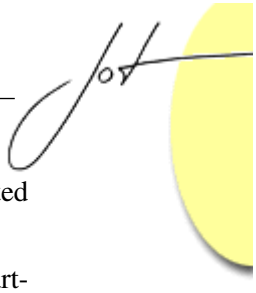
[23] L. Zhang and C. Krintz. Adaptive code unloading for resource-constrained JVMs. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 155–164, New York, NY, USA, 2004. ACM Press.

[24] L. Zhang and C. Krintz. Profile-driven code unloading for resource-constrained JVMs. In *International Conference on the Principles and Practice of Programming in Java*, Las Vegas, NV, June 2004.

ABOUT THE AUTHORS

David Anthony is a senior and an undergraduate research assistant in the Department of Computer Science and Engineering at University of Nebraska-Lincoln, USA. He can be reached at danthony@cse.unl.edu.

Witawas Srisa-an is an Assistant Professor in the Department of Computer Science and Engineering (http://www.cse.unl.edu) at University of Nebraska-Lincoln, USA. His research interests



include computer architecture, Object-Oriented systems, programming languages, and distributed systems. He can be reached at witty@cse.unl.edu.

Michael Leung graduated in 2005 with a BS degree in Computer Engineering from the Department of Computer Science and Engineering at University of Nebraska-Lincoln, USA. He can be reached at mleung@cse.unl.edu.