To JIT or not to JIT: The Effect of Code Pitching on the Performance of .NET Framework

David Anthony, Michael Leung and Witawas Srisa-an
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588
{danthony, mleung, witty}@cse.unl.edu

ABSTRACT

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 code-cache and can be reused quickly to satisfy future method calls. While 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 [7, 14, 10]. JIT compilation has proved to be much more efficient than interpretation especially in execution intensive applications [6, 7, 14, 16]. 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 [9]. This code-cache is located in the heap region .

The size of 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*) or frequently referred to as *Rotor*, the initial code-cache size is set to 64 MB. Once the accumulation of compiled method reaches this size, the system can choose to either increase the code-cache size or keep the same size and free all the compiled methods not currently in scope (referred to as pitching) [10]. There are two possible overheads of the "code pitching" mechanism [10, 9]— the overhead of traversing through all the compiled methods and the overhead of recompiling methods after pitching. However, pitching provides a means to maintain a small code-cache as 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 [9]. Such devices include smart phones, Pocket PC, 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 [16, 15, 3]. 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies' 2005 *Conference Proceedings*
ISBN 80-86943-01-1
Copyright UNION Agency — Science Press, Plzen, Czech Republic

used to perform the experiments. Section 4 discusses the experiments and results conducted in regards to the research questions. It also contains the detailed analysis of our findings. Section 5 presents the future work. Section 7 discusses prior research work in this area. The last section concludes this paper.

2. BACKGROUND

This section discusses background information related to this work.

2.1 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" [10]. There are several implementations of this standard that include Microsoft's *Common Language Runtime* (*CLR*), Microsoft's Shared Source Common Language Infrastructure (SSCLI), Microsoft's .NET Compact Framework, Ximian's Mono project, and GNU's dotnet project. 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 Mono or GNU's DotNet 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 from the CLR. 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 [11]. Third, it is designed to provide maximum portability. Thus, a software layer called Portable Adaptation Layer (PAL) is used to provide Win32 API for the SSCLI. Currently, the SSCLI has been successfully ported to 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 [14]. JIT compiler also ensures that every instruction conforms to the specification by 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 [9]. 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 [9, 10].

2.2 Code Pitching Mechanism

The execution engine initializes the code-cache by allocating 8KB. The reserve code-cache size is set to the target code-cache size which is defined by a variable. By default, this variable is set at 64MB by the SSCLI designers. 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 target code-cache size, the allocator will decide whether to allocate more heap space beyond the target cache size or pitch all unused methods. The allocator will not consider code pitching until the target code-cache size, maximum cache size or pitch trigger is reached. The default target cache size is 64 MB whereas the maximum cache size is 2GB.

Once the target code-cache size is reached, the allocator chooses between increasing the cache size or pitching unused code. If the reserved size is less than the target code-cache size or the existing pitch overhead is over the acceptable maximum (default 5ms), the allocator will attempt to increase the code-cache size. During this attempt, if the total needed memory is greater than the reserved size, less than the hard limit, 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 unused code. 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 reach 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 which 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.

2.3 The DNProfiler

Rotor comes packaged with a sample profiler called the DNProfiler. The DNProfiler provides callbacks to the CLR allowing a user to see what is going on without having to hard-code debug statements into the source or develop complicated hooks. The profiler provides callbacks for shutdown and initialization, JIT events, garbage collection, threading, etc... All the user has to do is provide handler code in the DNProfiler to process information during callback events. Once the DNProfiler is coded and compiled, the user has to activate it by turning profiling on and setting the profiling mask to what they want to monitor.

To gather data, the DNProfiler was modified to handle the JIT events. Specifically, the beginning and end of method compilation was monitored along with program initialization and shutdown and pitch events. A high performance counter was used to provide the most accurate time results possible.

The DNProfiler by itself cannot provide enough information to conduct our work. In order to track code-cache usage, we also modify the JIT compiler in the section of code that is responsible for allocating space for compiled code, garbage collection of unused methods, and maintaining the data structures representing the code-cache.

[1]European Computer Manufacturers Association

3. EMPIRICAL STUDY

As stated earlier, the behavior of compiled methods in .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 perform pitching as the last resort. This may not be the most optimal approach especially in the Compact Framework where the amount of memory available on 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.

3.1 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 utilize existing experimental objects written in C# to perform our experiment.

Throughout the experiment, we monitor 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, but will remove the rest.
- *Number of Recompilations*
  After a method has been pitched, each time it has to be complied again is called a recompilation. A method could be pitched and recompiled multiple times.
- *Number of Different Methods*
  This is the number of unique methods compiled. The number of unique 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 8k.
- *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 observe the average size of compiled method and compared them to the sizes of typical objects. In order to do our experiments, we need to create an environment where 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.

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

3.2 Experimental Objects

To address our research questions, we need a set of programs that compiled 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 as simple 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 [8], 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 [8].
- **Hello World**
  This is the classic "Hello World" program written in C#. It simply prints "Hello World" to the console and exits. Using such a simple program provided insight into how many methods were needed just to start and stop program execution. The specific file used is available in the sscli/samples/hello directory.
- **CodeToHTML**
  CodeToHTML is an example program found in the sscli/samples/utilities/codetohtml directory. This program parses a given C# or Jscript file and converts

Application	Minimum (bytes)	Maximum (bytes)	Average (bytes)	Standard Deviation	Number of Methods
LCSC	52	27024	1044.93	2587.04	1351
AHC	52	6320	317.04	474.21	514
Hello World	52	6320	299.95	472.37	327
CLisp	52	44008	425.66	1424.96	1168
CodeToHTML	52	44008	460.67	1543.39	1665

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	0.65%	1.08%	1.41%	1.77%	2.16%
AHC	0.03%	0.05%	0.07%	99.95%	99.96%
Hello World	19.81%	35.54%	49.28%	55.03%	79.38%
CLisp	6.27%	11.58%	16.66%	40.12%	94.85%
CodeToHTML	0.08%	0.18%	0.24%	0.28%	0.33%

Table 2: Code-cache usage based on percentage of execution

it to an HTML file. The generated HTML file displays formatted C# in a clearly organized manner. The test cases used were the C# files from the LCSC benchmark and are available for download from the Microsoft web site.

- **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.

4. RESULTS

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

4.1 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 that are recompiled, and the size of the compiled methods. Table 1 depicts the size information of compiled methods in our benchmark programs.

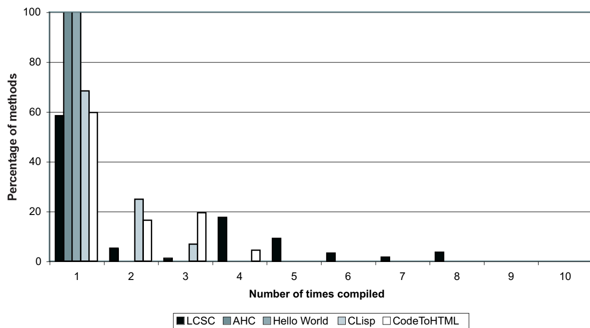

Figure 1: Distribution of compiled methods based on the number of compilations

It is worth noticing that typical objects in Object-Oriented Languages such as Java and C# only have the average object size of less than 100 bytes [4, 13]. However, the average size of the compiled methods in each application range from 300 bytes to 1000 bytes. It is also worth noting that the smallest size for a compiled method is 52 bytes. This is true across all applications. For the largest size, a method can be as large as 44K bytes. Since the SSCLI commits memory in increments of 8K bytes, five requests to increment must be made just to hold the largest compiled method in our applications. If no pitching is used, 1.5 MB of memory is needed to stored the compiled methods in LCSC (LCSC needs the largest amount of memory at 1.4 MB).

It is also worth noticing that even small applications such as HelloWorld, a significant number of methods is still needed to complete the execution (i.e. 327 methods in this case). However, we also find that complex applications such as compilers or HTML generator only require about 1500 methods. We suspect that both compilers and HTML generator perform repetitive routines, many of the methods can be reused over the length of execution.

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 percentage of execution and the percentage of the consumption of the code-cache. 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 212 KB, we observe the percentage of execution. In this case, the program has only completed 0.65% of the total execution time (see Table 2). It is worth noting that in three out of five applications, about 50% of all the space needed for the code-cache are consumed with in the first few percents of execution.

We also monitor the distribution of methods based on the number of compilations. We set the code-cache size to 256KB to emulate embedded devices environment and induce some pitch events. We find that in two applications AHC and HelloWorld, all methods are compiled only once. However, in larger applications, such as compilers

and HTML converter, about 40% of methods are compiled multiple times. Notice that CLisp and CodeToHTML require at most 3 and 4 compilations, respectively. However, LCSC requires methods to be compiled as many as 8 times. As stated earlier, most of these applications execute repetitive tasks. Thus, 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.

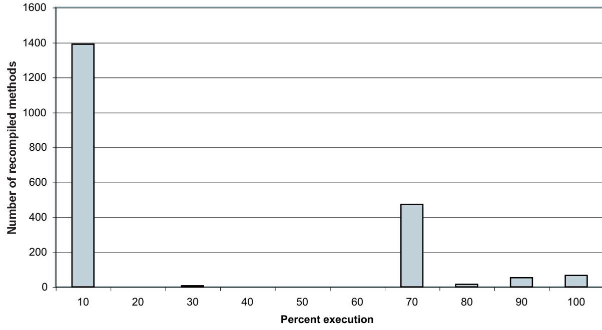

Figure 2: Distribution of recompiled methods over the execution time

In terms of access behavior, we find that in **all applications**, methods are heavily accessed within the first 6% of execution time. Then they are accessed moderately from 6% to about 30% of execution time. Afterward, they are infrequently accessed. To investigate the number of recompiled methods, we set the code-cache size to 256KB to force pitching. We find that about 70% of recompilation occur during the first 6% of execution time (depicted in Figure 2) in all benchmark programs that perform recompilation (excluding AHC and HelloWorld). 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 [5, 12]. This behavior may provide an opportunity for optimization by dynamically adjusting the heap size as needed. For example, the heap size can initially be set to be larger and then reduced after the first 6% of execution. We are currently experimenting with this approach and will report the result in the subsequent publication.

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 still require at least 300 methods to execute.
- In larger programs, a large number of methods is 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.
- The reuse often occurs toward the beginning of the program execution.

4.2 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 (shall be referred to as *Approach 1*) is to set the initial code-cache to a certain size (e.g. 256KB). This however, is not the highest possible value. When the amount of compiled methods reach 256KB for the first time, the system will pitch all methods that are not in scope but it will also consider whether to increase the cache size. Thus, if the cache size is doubled, the next pitch event will occur when the accumulation of methods in the code-cache approaches 512KB. Figure 3 depicts the pitch events using Approach 1. The initial code-cache is set to 256KB.

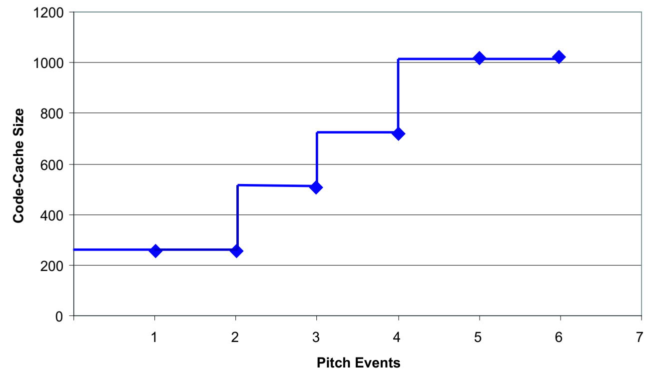

Figure 3: Monitoring pitch events using Approach 1

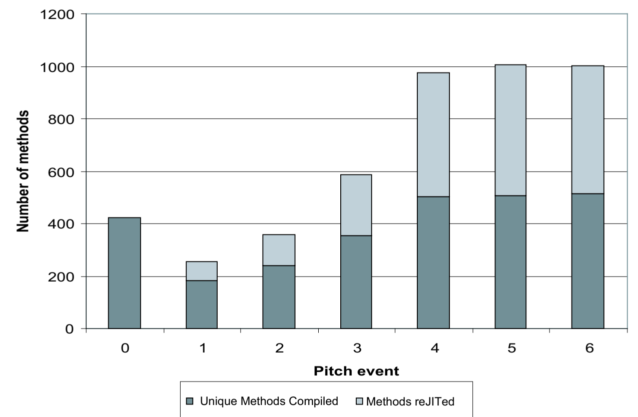

Figure 4: Ratios between new methods and recompiled methods based on pitch events

Figure 3 illustrates the basic behavior of code-cache expansion in Approach 1. The diamonds in the figure represent the all the pitch events that occur in the system. In this example, we have 6 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 in the reduction of the number of pitch events during the initial execution period. For example, by increasing the initial cache size from 256 KB to 512 KB, the number of pitch events decrease by two in LCSC.

These two events occur during the first five percent of the execution.

Figure 4 depicts the number of methods that are recompiled by applying Approach 1 in which the cache size can be increased as needed. Notice that there are more methods rejitted 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 accessed and must be recompiled.

It is worth noting that the initial target size can greatly affect the number of pitch events in the 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 heap size be advantageous by facilitating more reuse at the beginning.

Figure 4 initially appears to be contradicting Figure 2 as the amount of recompiled (reJITed) 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 the 33rd percent and the last event occur at the 80th percent. Thus, most of the recompilation events occur during the initialization of the system.

The second method (shall be referred to as *Approach 2*) 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. Table 4 provides the information about the pitch events and the total execution time in LCSC when the Approach 2 is applied. Again, we monitor the number of pitch events with respect to the different cache sizes.

Notice that excessive pitching (as in the cases of 256K and 512K cache size using Approach 2) can result in significant runtime overheads (864 seconds with 6700 pitch events versus 66 seconds with no pitching). We also find that a small amount of pitching does not significantly affect the overall performance; however, it can lead to a very significant reduction in memory usage. For example, if the cache size is set to 2MB, there is no pitching in the system. The execution time of this scenario is about 67 seconds. On the other hand, if we set the heap size to 1MB (50% saving in memory usage), there are 4 - 5 pitch events (depending on whether Approach 1 or 2 is used), but the execution times only increase by about 1 second or 1.5%. Thus, in the memory constrained systems pitching can be used to reduce the memory footprint without incurring a substantial amount of overhead.

Figure 5 depicts the usage of code-cache as LCSC is executed. The x-axis represents the percentage of execution completion 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 heap size using Approach 1, 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.

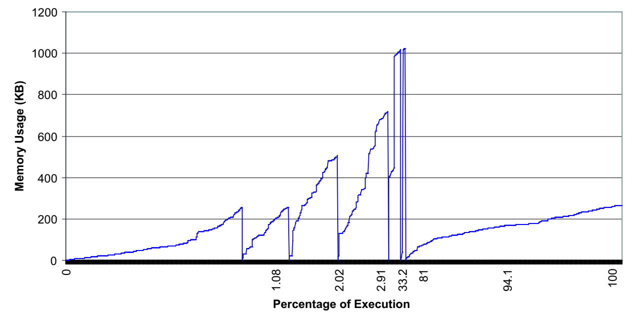

Figure 5: Code cache usage (256KB)

In addition, after the pitch event at the 33rd percent of the execution time, the next pitch events does not occur until the 81st percent. 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 can also be obtained.

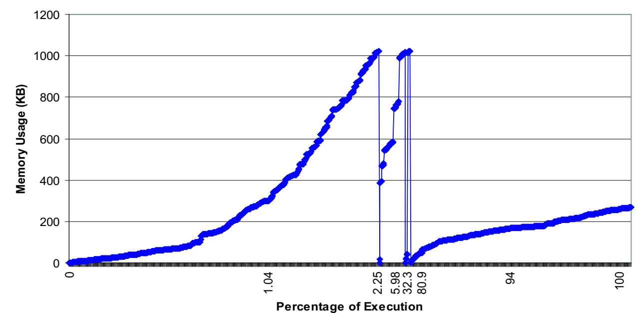

Figure 6: Code cache usage (1024KB)

Figure 6 depicts the usage of code-cache for LCSC with 1024KB cache size applying approach 2. 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 the 33rd and 81st percents. As a reminder, this is similar to the number of pitch events in Figure 5 after 4% of the program have been executed. Thus, a larger cache size clearly reduces the number of pitching activities during the initial state of execution.

In summary, we conclude that the following policies can 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 overheads. Thus, the policy should favor reducing memory usage over a moderate increase in pitching activities.
- Larger initial cache size can significantly reduce the number of pitch events during the program initialization. Thus, the policy should allocate a large enough cache at the beginning.
- 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

Applications	256k	512k	1024k	2048k	4096k	8192k	16384k	65536k
LCSC	6	4	3	0	0	0	0	0
AHC	0	0	0	0	0	0	0	0
CodeToHTML	3	2	0	0	0	0	0	0
Hello World	0	0	0	0	0	0	0	0
CLisp	2	1	0	0	0	0	0	0

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

Cache Size	Approach 2		Approach 1	
	Execution Time (sec)	Pitches	Execution Time (sec)	Pitches
256k	864.32	6774	68.81	6
512k	412.17	1470	68.64	5
1024k	69.45	5	69.44	3
2048k	66.88	0	68.38	0
4096k	66.78	0	68.16	0
8192k	67.52	0	68.38	0
16384	67.59	0	67.98	0
65536	67.58	0	68.19	0

Table 4: The number of pitch events and execution times with Approach 2

and not result in a substantial run-time overhead. Thus, the policy should include reducing the cache size after the initialization phase.

5. FUTURE WORK

Better benchmarks are needed that utilize more methods that force the execution engine to pitch more frequently especially for larger cache sizes. Ideally, pitching should occur with heap sizes that are close to the default target size. 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 be focused on two 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 removes 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.

6. RELATED WORK

In [2], multi-level recompilation technique was introduced as part of the Jalapẽno Virtual Machine. The basic idea is to use 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 accesses) 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 separately compartmentalized from the main heap memory region. This is different than work conducted by Zhang [16, 15]. In their work, the IBM's Research Virtual Machine (RVM) [1] was modified to incorporate code pitching. Unlike the .NET CF and the SSCLI, the RVM intermixed objects with compiled methods and therefore, the regular garbage collector is used to unload compiled methods. Their framework attempted to adaptively balance the compilation overhead and memory usage in the environment where objects and compiled code are stored together. Their main strategy is to identify what to unload and when to unload compiled methods. They reported that their strategy can reduce the code size by 43% without incurring substantial overhead in memory unconstrained system. If the memory is constrained, they can reduce the code size by as much as 61%. They also claimed that a significant reduction in execution time (22%) can be obtained due to less time spent in garbage collection.

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 [16].

7. 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 1000 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. Even the simplest programs such as HelloWorld still require as many as 300 methods to execute.

Based on the above finding, we conduct multiple experiments using different code-cache configurations. First, we set the initial cache size to different values ranging from 256KB to 64MB. 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 33% (from 6 events to 4 events). Having a large initial cache size can be advantageous since most of the method reuse occur within the first few percents 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 incur 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, 4 to 5 pitch events would occur but the overall execution time only increase by 1.5%. Thus, we conclude that a well designed pitching policy can greatly reduce the amount of 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.

8. 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.

9. REFERENCES

[1] B. Alpern, M. Butrico, T. Cocchi, J. Dolby, S. Fink, D. Grove, and T. Ngo. Experiences porting the jikes rvm to linux/ia32. In *Proceedings of 2nd Java(TM) Virtual Machine Research and Technology Symposium (JVM'02)*, San Francisco, California, August 1-2, 2002.

[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] 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.

[4] 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.

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

[6] 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.

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

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

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

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

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

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

[13] 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.

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

[15] 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.

[16] 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.