2 Motivation

I continue with presenting two very simple example methods that illustrate the motivation from using cached profiles. This should provide the reader with an understanding how and why cached profiles can be beneficial for the performance of a Java Virtual Machine. I will omit any implementation details on purpose as they will be discussed in Chapter 3 in detail.

Ideally, being able to reuse the profiles from previous runs should result in two main advantages:

1. **Lower start-up time of the JVM:** Having information about the program flow already, the compiler can avoid gathering profiles and compile methods earlier and directly at higher compilation levels.

2. **Less Deoptimizations:** Since cache profiles get dumped at the end of a compilation, when using these profiles the compiler can already include all optimizations for all different method executions. The compiled code includes less uncommon traps and therefore less deoptimizations occur.

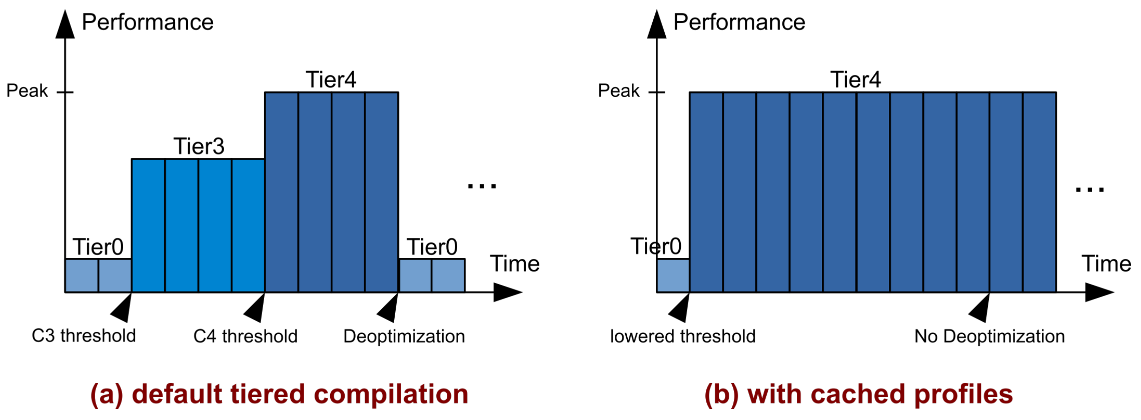

Figure 2.1: schematic visualization of cached profile benefit

Figure 2.1 gives a schematic visualization of the expected effect on performance of a single method when using cached profiles compared to the current state without such a system and standard tiered compilation. The blue bars roughly represent method invocations and higher bars equal higher compilation levels and therefore higher performance. The x-axis represents time since the start of the JVM. The figure shows the ideal case and abstracts away many details and other possible cases. However, it provides a good visualization for the examples provided in this chapter.

Listing 2.1: Simple method that does not get compiled

```java
class NoCompile {
    double result = 0.0;
    for(int c = 0; c < 100; c++) {
      result = method1(result);
    }
    public static double method1(double count) {
        for(int k = 0; k < 10000000; k++) {
            count = count + 50000;
        }
        return count;
   }
}
```

A more detailed performance analysis, also considering possible performance regressions is done in Chapter 4.

I'm using my implementation described in Chapter 3 in CachedProfileMode 0 (see 3.4.1) built into openJDK 1.9.0. All measurements in this chapter are done on a Dual-Core machine running at 2 GHz with 8GB of RAM. To measure the method invocation time I use hprof [8] and the average of 10 runs. The evaluation process has been automated using a couple of python scripts. The error bars show the 95% confidence interval.

2.1 Example 1

For this very first example, on-stack replacement has been disabled to keep the system simple and easy to understand.
Example one is a simple class that invokes a method one hundred times. The method itself consists of a long running loop. The source code is shown in Listing 2.1. Since OSR is disabled and a compilation to level 3 is triggered after 200 invocations this method never leaves the interpreter. I call this run the *baseline*. To show the influence of cached profiles I use a compiler flag to lower the compile threshold explicitly and, using the functionality written for this thesis, tell HotSpot to cache the profile. In a next execution I use these profiles and achieve a significantly lower time spend executing the cached method as one can see in Figure 2.2. This increase comes mainly from the fact that having a cached profile available allows the JVM to compile highly optimized code for hot methods earlier (at a lower threshold) since there is no need to gather the profiling information first.

Since the example is rather simple neither the baseline nor the profile usage run trigger any deoptimizations. This makes sense because after the first invocation, all the code paths of the method have been taken already and are therefore known to the interpreter and saved in the profile.

Enabling OSR vanishes the difference between with and without using cached profiles. This happens because HotSpot quickly realizes the hotness of the method and the simplicity of the method allows the JIT compiler to produce optimal code already. The interpreted version gets replaced

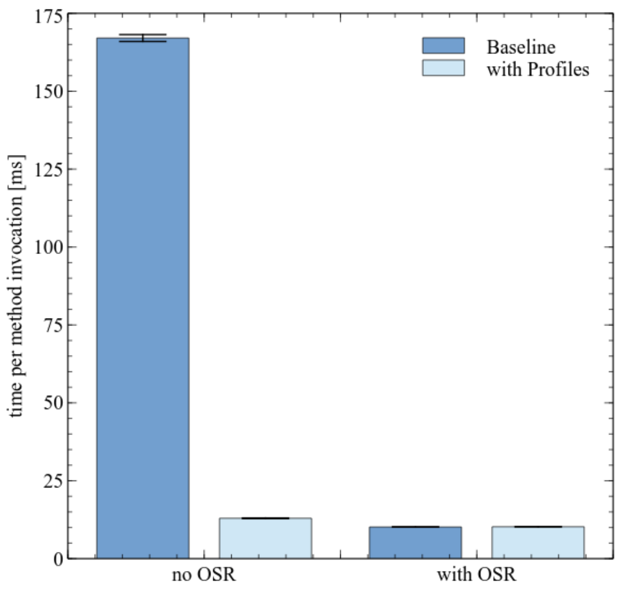

Figure 2.2: NoCompile.method1 - per method invocation time

on the stack by the compiled version during he first method invocation. The optimality of the compiled code is is confirmed by the fact that no deoptimizations occur. This example appears rather artificial since the same performance can be achieved with OSR already but nevertheless shows the influence of early compilation.

2.2 Example 2

However, OSR is one of the core features of HotSpot to improve startup performance of a JVM and disabling that does not give us any practical results. I came up with a second simple example sketched in Listing 2.2, which is slightly more complex but demonstrates the influence of cached profiles without disabling any HotSpot functionalities,

The idea is to create a method that takes a different, long running branch on each of it's method invocations. Each branch has been constructed in a way that it will trigger an OSR compilation. When compiling this method during its first iteration only the first branch will be included in the compiled code. The same will happen for each of the 100 method invocations. As one can see in Figure 2.3 the baseline indeed averages at around 130 deoptimizations and a time per method invocation of 200 ms.

Now I use a regular execution to dump the profiles and then use these profiles. So theoretically the profiles dumped after a full execution should include knowledge of all branches and therefore the compiled method using these profiles should not run into any deoptimizations. As one can see in Figure 2.3 this is indeed the case. When using the cached profiles no more deoptimizations occur

Listing 2.2: Simple method that causes many deoptimizations

```
class ManyDeopts {
    double result = 0.0;
    for(int c = 0; c < 100; c++) {
      result = method1(result);
    }
    public static long method1(long count) {
        for(int k = 0l; k < 100000000l; k++) {
            if (count < 100000000l) {
                count = count + 1;
            } else if (count < 300000000l) {
                count = count + 2;
                .
                .
                .
            } else if (count < 505000000000l) {
                count = count + 100;
            }
            count = count + 50000;
        }
        return count;
    }
}
```

and because less time is spent profiling and compiling the methods the per method execution time is even significantly faster with averaging at 190ms now.

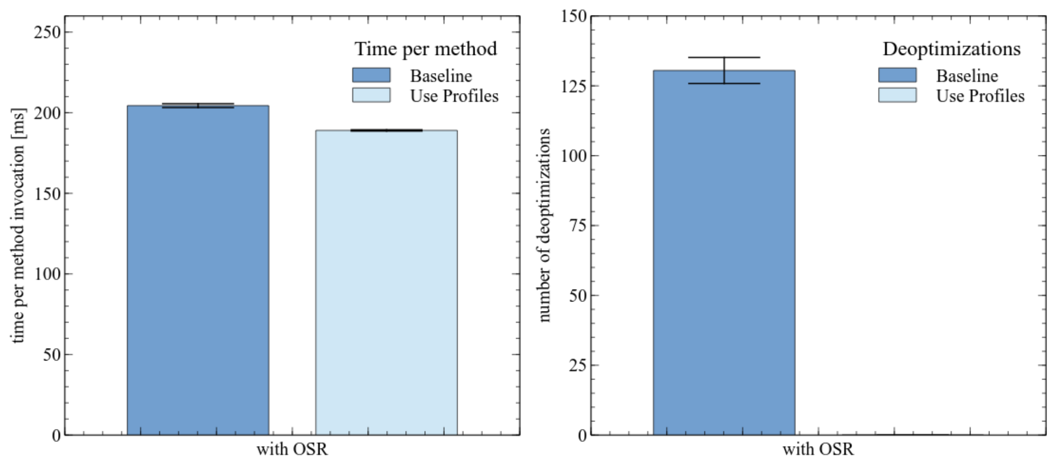

Figure 2.3: ManyDeopts.method1 - per method invocation time and deoptimization count

2.3 Similar systems

In commercially available JVMs the idea of caching profiles is not new. The JVM developed and sold by Azul Systems® called Zing® [2] already offers a similar functionality. Zing® includes a feature set they call ReadyNow!™ [1] which aims to increase startup performance of Java applications. Their system has been designed with financial markets in mind and to overcome the issue of slow performance in the beginning and performance drops during execution. Azul Systems clients reported that their production code usually experiences a significant performance decrease as soon as the market goes live and the clients start trading. The reasons are deoptimizations, that occur for example due to uncommon branch paths are taken or yet unused methods are invoked. In the past their clients used techniques to warm up the JVM, for example doing fake trades prior to market opening. However this does not solve the problem sufficiently, since the JVM optimizes for these fake trades and still runs into deoptimizations once actual trades are meant to happen. That is, because the code includes methods or specific code snippets that differentiate between the fake and the real trades.

ReadyNow!™ is a rich set of improvements how a JVM can overcome this issues. It includes attempts to reduce the number of deoptimizations in general and other not further specified optimizations. As one of the core features Azul Systems® implemented the ability to log optimization statistics and decisions and reuse this logs in future runs. This is similar to the approach presented in this thesis. However they do not record the actual optimization but the learning and the reasons why certain optimizations happen. This gives them the ability to give feedback to the user of the JVM whether or not certain optimizations have been applied. They also provide APIs for developers to interact with the system and allow further fine-grained custom-designed optimizations.

Unfortunately, they do not provide any numbers how their system actually improves performance applied to a real system or any analysis where the speedup originates from in detail.