# 3 Implementation / Design

This chapter describes the implementation of the cached profiles functionality for HotSpot, written as part of this thesis. HotSpot is a vital part of the open source Java Platform implementation `OpenJDK` and the source code is available at `http://openjdk.java.net/`.

Most of the code additions are included in two new classes `/share/vm/ci/ciCacheProfiles.cpp` and `/share/vm/ci/ciCacheProfilesBroker.cpp` as well as significant changes to `/share/vm/ci/ciEnv.cpp` and `/share/vm/compiler/compileBroker.cpp`.
The core functionality is located in `/share/vm/ci/ciCacheProfiles.cpp`, a class that takes care of setting up the cached profile data-structure as well as providing public methods to check if a method is cached or not. The class `/share/vm/ci/ciCacheProfilesBroker.cpp` is used before a cached method gets compiled. It is responsible for setting up the compilation environment, so the JIT compiler can use the cached profiles.

A full list of modified files and the changes can be seen in the webrev at `http://mohlerm.ch/b/webrev.01/` or Appendix A.3. The changes are provided in form of a patch for HotSpot version 1aef080fd28d. In the following, the original version is referred to as *baseline*.

I will describe and explain the functionality and the implementation design decision in the following sections, ordered by their execution order.

## 3.1 Creating cached profiles

The baseline version of HotSpot already offers a functionality to replay a compilation based on previously saved profiling information. This is mainly used in case the JVM crashes during a JIT compilation to replay the compilation process and allow the JVM developer to further investigate the cause of this incident. Apart from this automatic process, there exists the possibility to invoke the profile saving manually by specifying the `DumpReplay` compile command option per method.

I introduce a new method option called `DumpProfile` as well as a new compiler flag `-XX:+DumpProfiles` that appends profiling information to a file as soon as a method gets compiled. The first option can be specified as part of the `-XX:CompileCommand` or `-XX:CompileCommandFile` flag and allows the user to select single methods to dump their profile. The second commands dumps profiles of all compiled methods. The profile get converted to a string and saved in a simple

text file called *cached_profiles.dat*.

The system will only consider compilations of Level 3 or Level 4. Level 1 and Level 2 are rarely used in practice and do only include none or little profiling information. The user can also restrict the profiles to Level 4 ones by using the compiler flag: `-XX:DumpProfilesMinTier=4`.

The dumped profiling information consists of multiple `ciMethod` entries, `ciMethodData` entries, and one `compile` entry. They are separated by line breaks and keywords to make sure the data can be parsed correctly. A shortened example of a cached profile can be found in Appendix A.2. The `ciMethod` entries contain information about the methods used in the compilation and Table 3.1 describes it in more detail. The `ciMethodData` (see Table 3.2) includes all profiling data about the methods itself to be able to redo the compilation. The compile entry saves the bytecode index in case of OSR, the level of the compilation and lists all inlining decisions (Table 3.3).

Since method often get compiled multiple times and at different tiers, this results in dumping compilation information about the same method multiple times. This is intentional and is taken care of when loading the profiles (see Section 3.2).

Table 3.1: content of ciMethod entry in cached profile

| **name** | **description** |
|---|---|
| class_name,<br>method_name,<br>signature | used to identify the method |
| invocation_counter | number of invotations |
| backedge_counter | numbe of counted backedges |
| interpreter_invocation_count | number of invocations during interpreter phase |
| interpreter_throwout_count | how many times method was exited via exception while interpreting |
| instructions_size_name | rough size of method before inlining |

Table 3.2: content of ciMethodData entry in cached profile

| **name** | **description** |
|---|---|
| class_name,<br>method_name,<br>signature | used to identify the method |
| state | if data is attached and matured |
| current_mileage | maturity of the oop when snapshot is taken |
| orig | snapshot of the original header |
| data | the actual profiling data |
| oops | ordinary object pointers, JVM managed pointers to object |

Table 3.3: content of compile entry in cached profile

| **name** | **description** |
|---|---|
| class_name, method_name, signature | used to identify the method |
| entry_bci | byte code index of method |
| comp_level | compilation level of record |
| inline | array of inlining information |

## 3.2 Initializing cached profiles

The information dumped in step 3.1 can now be used in a next run of that particular program. To specify that profiles are available, I introduce a new compiler flag `-XX:+CacheProfiles` that enables the use of previously generated profiles. By default, it reads from a file called *cached_profiles.dat* but a different file can be specified using `-XX:CacheProfilesFile=other_file.dat`.

Before any cached profiles can be used the virtual machine has to parse that file and organize the profiles and compile information in a data-structure. This data-structure is completely kept in memory during the whole execution of the JVM to avoid multiple disk accesses. The parsing process is invoked during boot up of the JVM, directly after the compileBroker gets initialized. This happens before any methods get executed and blocks the main thread of the JVM until finished.

As mentioned in Section 3.1, the file consists of method informations, method profiles, and additional compile information. The parser scans the file once and creates a so called `CompileRecord` for each of the methods that include compilation information in the file. This compile record also includes the list of method information (`ciMethod`) and their profiling information (`ciMethodData`). As mentioned previously, a method's compile information could have been dumped multiple times, which results in multiple `CompileRecords` for the same method. In this case, HotSpot will only keep the `CompileRecords` based on the latest data written to the file but never overwrite an existing higher level profile. Because a profile dumped by the C1 compiler can not be used by the C2 compiler and vice verca, the level of the profile matters as it influences the compile level transitions described in Section 3.4. And since profiling information only grow, the compilation that happened last contains the richest profile and is considered the best. This is based on the fact, that the richer the profile, the more information about the method execution is known and influences the compiled version of that method. For example, a profile for a method might include data for all its branches and can therefore avoid running into uncommon traps and trigger deoptimizations.

The `CompileRecord` as well as the lists of methods information and profiles are implemented as an array located in HotSpot's heap space. They get initialized with a length of 8 and grow when needed. This choice has been done for simplicity and leaves room for further improvements.

## 3.3 Using cached profiles

The implementation offers three different modes `mode 0`, `mode 1`, and `mode 2`, that differ in the way they use the cached profiles. The following paragraph applies to all three modes and I will discuss the differences of the modes in detail in Section 3.4.

The idea is to modify the compiler to use cached profiles when available and if not continue as usual. A simplified graphical overview of the program flow for compiling a method with the changes introduced in this thesis can be found in Figure 3.1.

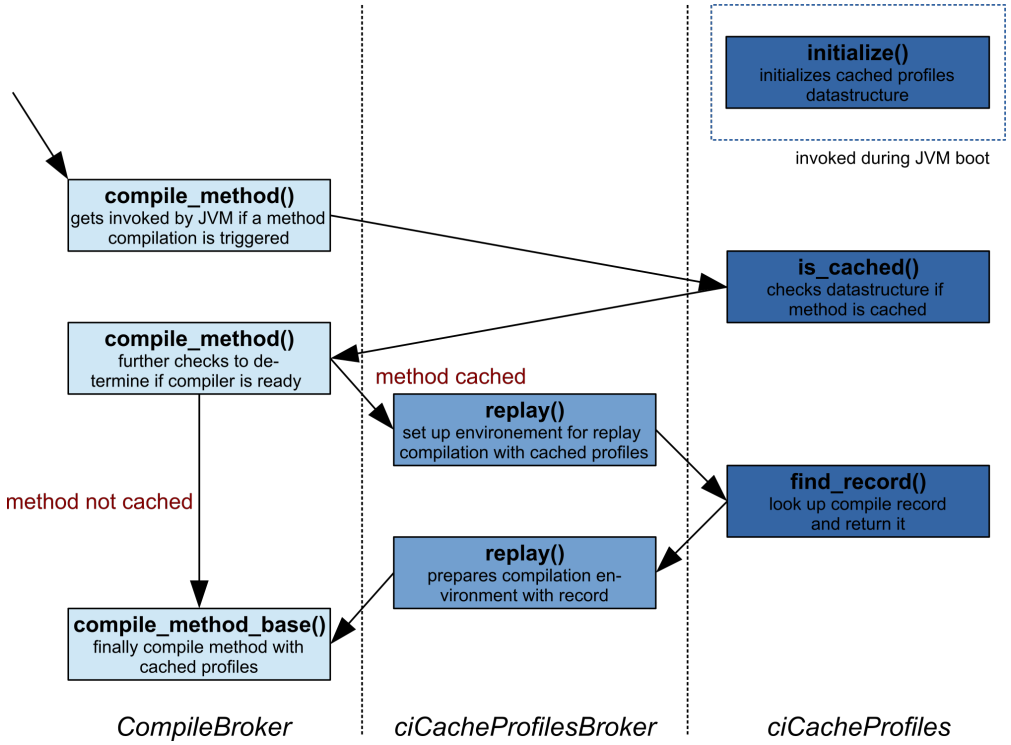


Figure 3.1: program flow for compiling a method

As mentioned before, once compilation thresholds are exceeded a method gets scheduled for compilation. This means that the JVM will invoke a method called `compile_method()`, located in the `compileBroker` class. This method tests if certain conditions hold, for example checks if the compile queue isn't full or if there is already another compilation of that particular method running. I extended this method with a call to `ciCacheProfiles::is_cached(Method* method)` which does a linear scan through the `CompileRecord` array data-structure. The method returns either 0 if the method is not cached or returns an integer value, reflecting the compile level, in case a cached profile of this method is available. Because only methods compiled with level 3 or 4 get cached, this call only gets executed if the compilation request is also of level 3 or higher.

Depending on the compilation level of the profile, the level of the requested compilation, and the `CacheProfileMode`, the `compileBroker` then schedules either a compilation using recently

freshly gathered profiles or calls into `ciCacheProfilesBroker` to replay the compilation, based on a cached profile. Since these decisions are different in each mode, I describe them in detail in the next section. In case the method is not cached, the execution continues like in the baseline version. Otherwise, the `ciCacheProfilesBroker` class then initializes the replay environment and retrieves the compile record from `ciCacheProfiles`. Subsequently, the needed cached profiles get loaded to make sure they are used by the following compilation. `ciCacheProfilesBroker` then returns the execution to the `compileBroker`, which continues with the steps needed to compile the method. Again some constraints are checked (e.g. if there is another compilation of the same method finished in the meantime) and a new compile job is added to the compile queue. Eventually the the method is going to be compiled using the cached profiles.

Since the implementation is only invoked by the static class `compileBroker`, `ciCacheProfiles` and `ciCacheProfilesBroker` are static classes as well. The `compileBroker` is solely called by the JVM main thread, therefore there is no need to make the compileRecord data-structure or any of the new implementations thread safe.

## 3.4 Different usage modes for cached profiles

The implementation of cached profiles offers 3 different modes, which distinguish from each other in the transitions between the compilation tiers. The motivation as well as the advantages and disadvantages of the modes are described in the following three subsections. While `mode0` and `mode1` are similar except for the compile thresholds, `mode2` differs significantly. Figure 3.2 provides a graphical overview of the differences in the compilation tier transitions of the modes.

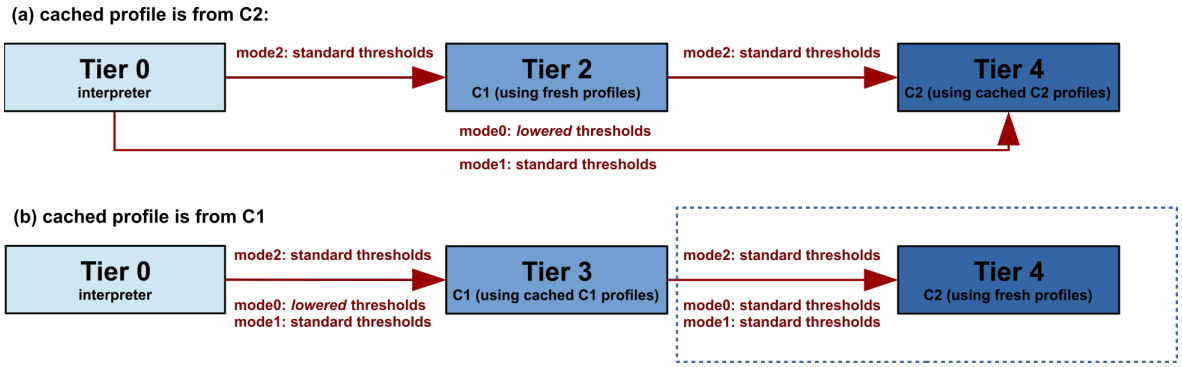


Figure 3.2: Tier transitions of different modes

All three modes have advantages and disadvantages and their performance is evaluated in Chapter 4. Since depending on the methods, different modes might perform best, it is up to the JVM user to decide which mode he or she wants to use. Because `Mode 2` is the most conservative one, which does not modify thresholds it is considered the default mode and used if not further specified.

### 3.4.1 Compile Thresholds lowered (mode 0)

The first mode is based on the consideration that a method which has a profile available does not require profiling anymore. Therefore, the compile thresholds (see Section 1.4) of these methods are lowered automatically. If the cached method has a C2 profile, all thresholds are lowered, in case of a C1 profile only the thresholds that affect the tiers below and equal to Tier 3. This differentiation has been done to prevent an increase in early C2 compilations using fresh profiles. Since these fresh profiles only contain little profiling information many costly deoptimizations could be triggered. By default, the thresholds are lowered to 1% of their original values but the threshold scaling can be modified with the JVM parameter: `-XX:CacheProfilesMode0ThresholdScaling=x.xx`. 1% results in the level 3 invocation counter being reduced from 200 to 2. This means that the method will be interpreted once but then directly trigger a compilation on the next invocation. Since the interpreter also handles class loading, this decision has been made to avoid the need of doing class loading in C1 or C2 which was considered out of the scope for this thesis.

In `mode 0`, the JIT compiler will always use a cached profile for compilations of Level 3 or Level 4 in case there is a cached profile which has been generated by a compiler of the same level. However, in case a method to be compiled on Level 3 has a cached profile available for Level 4, the compiler will skip the C1 compilation completely and immediately compile with C2. In this case, HotSpot directly uses the highly optimized version generated by C2 and ideally this results in a lower time to reach peak performance.

However, since the thresholds of all methods with cached profiles get lowered and some of the C1 compilations get promoted to C2 compilations the C2 compiler is put under heavy load. Especially during startup of a program, where many compilations happen naturally, C2 might not be able to handle all these requests at the same time and the compile queue fills up. This might negatively affect performance and is analyzed in Section 4.4.

### 3.4.2 Unmodified Compile Thresholds (mode 1)

`Mode 1` is doing exactly the same as `mode 0` but does not lower the compilation thresholds of methods with cached profiles. This is done to decrease the load increase on C2 as mentioned in Subsection 3.4.1. Apart from this change `mode 1` has the same behaviour as `mode 0`.

### 3.4.3 Modified C1 stage (mode 2)

Both modes mentioned before use cached profiles as soon as a compilation of level 3 and 4 are triggered. Since the thresholds for level 3 are smaller than the level 4 thresholds (see Appendix A.1) a method reaching a level 3 threshold could actually trigger a level 4 compilation, if the cached profile is one of level 4. So even if `mode 1` is used and the thresholds are untouched, C2 might get overloaded since compilations occur earlier.

`Mode 2` has been designed to make as little changes as possible to the tiered compilation and

prevent C2 being more used than usual. It does so by keeping the original tiered compilation steps and compilation thresholds and compiles methods with C1 prior to C2. But since there are already profiles available there is no need to run at Tier 3 to generate full profiles but instead it uses Tier 2. Tier 2 does the same optimizations but offers only limited profiles like method invocation and backbranch counters. They are needed to know when to trigger the C2 compilation and therefore Tier 1 can not be used. Tier 2 is considered about 30% faster on average than Tier 3 [7]. Eventually, if the Tier 4 thresholds are reached, the method is compiled using C2 and the cached profiles.

The above only makes sense if the cached profile is a C2 profile. If only a C1 profile is available, C1 should gather fresh, full profiles since they might be needed in C2 later. HotSpot will then only use the cached profile during the C1 compilation and then use the generated profiles for possible C2 compilations. In theory this transition is considered rare, because if a method has not been compiled with C2 when creating the profile it is unlikely to get compiled with C2 in the future.

## 3.5 Issues

If the profiles generated by multiple runs of the program deviate sharply it is likely that a cached profile does not fit to the current execution. In this case the compiled version would still trigger many deoptimizations and the method could end up having even worse performance since it's going to use the profile over and over again. To circumvent that behavior, only methods which have been deoptimized less then 10 times already will get compiled using cached profiles. If they are above that limit a standard compilation will be used instead. The limit is 10 to allow a small number of recompilations. This could for example be useful when the method is deoptimized due to classes not being loaded.

## 3.6 Debug output

For debugging and benchmarking purposes four debug flags are implemented, that can be used along with `-XX:+CacheProfiles`.

| **flag** | **description** |
|---|---|
| -XX:+PrintCacheProfiles | enable command line debug output for cached profiles |
| -XX:+PrintDeoptimizationCount | prints amount of deoptimizations when the JVM gets shut down |
| -XX:+PrintDeoptimizationCountVerbose | prints total the amount of deoptimizations on each deoptimization |
| -XX:+PrintCompileQueueSize | prints the total amount of methods in the compile queue each time a method gets added |