# Accurate Profiling in the Presence of Dynamic Compilation Howto

Yudi Zheng, Lubomír Bulej, Walter Binder

October 8, 2015

## 1 Getting Started

Our approach to accurate bycode-instrumentation-based profiling is implemented in Oracle's Graal compiler[1], and now integrated into openjdk.

The API to our approach is folded into com.oracle.graal.api.directives.GraalDirectives:

| Method | Description |
|---|---|
| instrumentationBegin(int) | Marks the beginning of the instrumentation boundary. It requires an argument indicating the length of the path between the inspected base program node and the instrumentation in the control flow graph. |
| instrumentationToInvokeBegin(int) | Similar to instrumentationBegin(int) but ensures the instrumentation is valid only if the inspected base program node is an invocation. Graal intrinsifies invocation during bytecode parsing, and may mislead the inspected base program node. |
| instrumentationEnd | Marks the end of the instrumentation boundary. |
| inCompiledCode | Returns a boolean value indicating whether the method is executed in compiled code. |
| rootName | Returns the name of the root method for the current compilation task. |
| isMethodInlined | Returns a boolean value indicating whether the enclosing method is inlined. Valid only within the instrumentation (i.e. surrounded by invocations to instrumentationBegin and instrumentationEnd). |
| runtimePath | Returns an integer representing a taken path at runtime. This merges getAllocationType and getLockType. Valid only within the instrumentation, and requires the inspected allocation or lock acquisition to precede the instrumentation. |

To setup a Graal development environment, run:

```
$ hg clone https://bitbucket.org/allr/mx
$ export PATH=$PWD/mx:$PATH
$ mkdir graal
$ cd graal
$ mx sclone http://hg.openjdk.java.net/graal/graal-compiler
$ cd graal-compiler
$ mx build
```

During the compilation, you will be asked to choose Java distributions for JAVA_HOME and EXTRA_JAVA_HOME. Set them to point to the location of JDK 8. **You should select jvmci when being asked for the default VM.**

**Note** For details, visit https://wiki.openjdk.java.net/display/Graal/Instructions

[1] http://openjdk.java.net/projects/graal/

Once the compilation is finished, you can launch a JVM with Graal compiler using "mx vm". For instance, instead of running "java -version", you can run "mx vm -version", and it will display the following message:

```
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
OpenJDK 64-Bit JVMCI VM (build {IDENTIFIER}, mixed mode)
```

Various VM flags (passed to "mx vm") should be taken into account while applying our approach:

| **Flag** | **Description** |
|---|---|
| -G:+UseGraalInstrumentation | Enable this flag to activate our approach. |
| -G:-RemoveNeverExecutedCode | Disable this flag if the instrumentation code is guarded by inCompiledCode. Otherwise, the compiler will insert a de-optimization, and the instrumentation will never be executed. |
| -G:-InlineDuringParsing | Disable this flag if you want to inspect the inlining behavior using our approach. Otherwise, the inlining will be performed during bytecode parsing, and cannot be observed through our approach. |
| -G:SmallCompiledLowLevelGraphSize | Set to 0 to prevent compiled-code's size contributing to inlining heuristics. |
| -XX:-TieredCompilation | Disable this flag to disallow tiered compilation. Currently, our approach is only implemented in Graal and the C1-compiled code will be treated like in interpreting mode. |
| -XX:+BootstrapJVMCI | Enable this flag to perform a boot strapping compilation for the Graal compiler's code. This will limit the interference from the dynamic compilation of Java library classes and Graal itself. |

## 2 A Close Look

If you would like to test our approach more extensively, the easiest way is to add a new test into the Compiler Testing Framework. We several many test cases that target partial escape analysis and inlining for illustration. The source code of these test cases can be found at `https://github.com/mur47x111/CompilerTesting`. Checkout and run using the following commands:

```
$ git clone https://github.com/mur47x111/CompilerTesting.git
$ cd CompilerTesting && ant
$ export GRAAL_HOME=/path/to/graal-compiler
$ ./runTest pea.TestSuite inlining.TestSuite query.TestSuite
```

Specifically, the test cases inlining.PolyMethod* demonstrate the two polymorphism-related reasons for not inlining call sites mentioned in Table 3 in our paper. Starting with these test cases, you may create similar test cases to validate Graal's default inlining policy. You can also create new test cases targeting other Graal optimizations using the following steps:

1. Create a sub class extending ch.usi.dag.testing.JITTestCase
2. Override warmup() and isWarmedUp(). During the warmup phase, warmup() is called repeatedly until isWarmedUp() returns true.
3. Create test methods annotated by @Test.

**Note** You must carefully assign the argument to instrumentationBegin when applying our approach in the Java source code. For instance, the new statement will be compiled into a new bytecode and an invokespecial to the class's constructor. You should pass -2 to instrumentationBegin such that the instrumentation will be associated with the directly preceding new statement.

## 3 Tool Support

The source code of the profilers mentioned in our paper can be found at `https://github.com/mur47x111/graal-assisted-profilers`. The following steps allow you to execute the profiler and to observe the produced output messages:

1. Check out and compile DiSL-graal

```
$ git clone https://github.com/mur47x111/DiSL-graal.git
$ cd DiSL-graal && ant
$ export DISL_HOME=/path/to/DiSL-graal
```

2. Check out graal-assisted-profilers

```
$ git clone https://github.com/mur47x111/graal-assisted-profilers.git
$ cd graal-assisted-profilers
```

3. Enter one of the profiler directories and compile.

```
$ cd AllocationProfiler && ant
```

4. Execute the run script to launch the profiler with your application.

```
$ ../scripts/run /path/to/profiler.jar -cp /path/to/application/classes YourAppName
```

5. Execute the profile script to launch the profiler with the DaCapo benchmarks.

```
$ ../scripts/profile /path/to/profiler.jar /path/to/dacapo-9.12-bach.jar
```

   The scope of the profiling can be configured:

   - The choice of benchmarks is determined by graal-assisted-profilers/config/bench.txt, which is a symlink to bench-sanity.txt by default. Linking it to bech-all.txt will cause the profiler scripts to use the benchmarks from the evaluation setup of our paper.
   - The choice of classes and methods to profile is determined by graal-assisted-profilers/config/excl.txt. Classes that match the expressions in this file will be excluded from instrumentation and hence will not be profiled. By default the JDK classes are excluded. Removing these expressions will cause the JDK classes, the GraalVM, and all its dependencies to be profiled as well, which will significantly increase the time needed to collect the profiles. Moreover, infinite recursive invocations may occur if you profile the profiler itself. For solving such problem, check out DiSL's dynamic bypassing feature or ShadowVM.
   - The number of DaCapo benchmark iterations is controlled by the ITERATIONS variable in each profile script. Adjust the number following the "-n" option in the ITERATIONS variable to change the number of iterations.
   - The workload size of DaCapo benchmarks is controlled by the WORKLOAD variable in each profile script. The variable is empty by default, which selects the default workload size. Use the "-s" DaCapo option to change the workload size, by setting the WORKLOAD variable to, e.g., "-s small", or "-s large", etc.

Below we briefly describe each profiler and the corresponding outputs. Please consult our paper for additional details on each profiler.

- **AllocationProfiler** tracks types (TLAB/heap/stack allocation) of object allocations for each allocation site.
- **InliningProfiler** profiles hotness of call-sites that are not inlined.
- **ReceiverProfiler** profiles receiver types for each call site keyed to 0, 1, 2, and 3 levels of calling context.