Dynamic Code Evolution for the Java HotSpot™ Virtual Machine

Thomas Wuerthinger

April 1, 2009

Contents

1 Introduction

This document is a technical description of the prototype implementation of dynamic code evolution for the Java HotSpot™ Virtual Machine. It explains the basic concepts and contains pointers to the source code in form of method and class references. It also contains instructions on how to execute the unit tests.

In contrast to the current implementation in the Java HotSpot™ Virtual Machine, this approach tries to support all possible types of dynamic code evolution and is not limited to swapping method bodies. The code is currently in prototype state. Gray boxes in this document indicate things that are still on the TODO list.

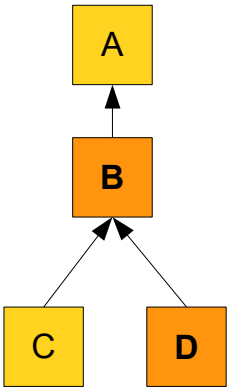

Figure 1: Redefining D and B.

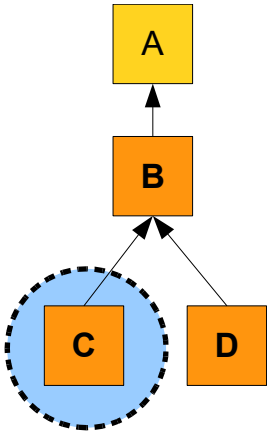

Figure 2: Find all affected classes.

2 Algorithm

2.1 Loading the New Classes

Receive redefinition command. A class redefinition command is always given for a set of classes. Those classes must be already loaded in the VM. Logically they need to be replaced in an atomar operation, as it could lead to an invalid type system when only part of the new classes are redefined. In the current implementation, there must not be any user thread running while the class redefinition is performed. Therefore, we suspend all threads that are not agent threads, and resume them after the class redefinition is completed (see method `JvmtiEnv::RetransformClasses`).

Find all affected classes. Redefining a class can have effects on its subtypes. Therefore, we find all classes that are possibly affected by the class redefinition. Figure 1 introduces an example class hierarchy. When redefining the classes, it is identified that C is possibly affected by the redefinition as shown in Figure 2 (see class `VM_RedefineClasses::FindAffectedKlassesClosure`).

Sort the classes topologically. The full list of affected classes is now sorted topologically such that a super class is always at an earlier position than its subclass (see Figure 3). This is necessary, because we need to ensure that a super class is always redefined before a subclass. This ensures that the new version of the subclass can be loaded with respect to the new version of the super class. The class that does the topological sorting is (see class `VM_RedefineClasses::TopologicalClassSorting`).

The subclass relationships in the new version of the class hierarchy can differ from those in the old class hierarchy. It could be the case that in the old version A is a subclass of B, but in the new version B is a subclass of A. We need to ensure that the topological sorting is based on the *new* class hierarchy. For this purpose, we preparse the new class bytes just to find out about the super types of a class (see method `ClassFileParser::findSuperSymbols`).

Build side universe. Now we know which classes to redefine in which order. The new classes are loaded such that they coexist with the old classes (see Figure 4). We update entries in the system dictionary to point to the new class (see method `Dictionary::update_klass`). For classes not redefined, but only affected by redefinition, we load the same byte codes again to get a duplicated version of the class (see method `VM_RedefineClasses::find_class_bytes`). The different versions of a class are double linked. One can retrieve the old version of a class (see method `Klass::old_version`) or the new version (see method `Klass::new_version`).

> **TODO:** It is planned to add verification code after loading all classes. This code needs to check whether there could be inconstencies as execution proceeds (e.g. deleted method is called, deleted field is accessed, static type / dynamic type relationship is violated).

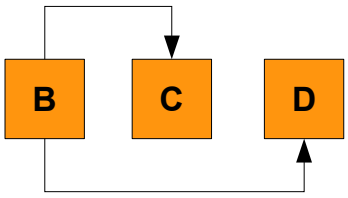

Figure 3: Topological order.

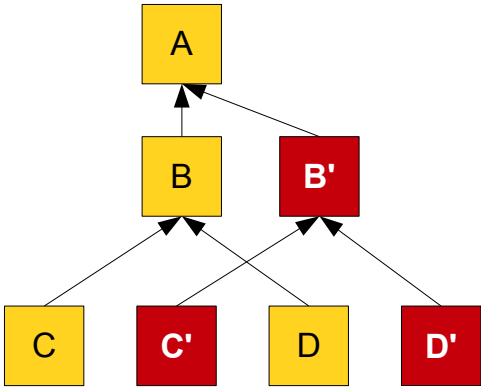

Figure 4: Building side universe.

2.2 Updating the Data Structures and Pointers

Flush dependent code Currently all compiled methods are deoptimized (see method `VM_RedefineClasses::flush_dependent_code`).

> **TODO:** Add a system that allows to decide which methods need to be deoptimized. For this purpose, we need to know all methods that call a certain method or access a certain field. An intermediate step could be to at least deoptimize only methods that depend on a redefined class.

Update constant pool cache entries. We iterate over all classes to update their constant pool cache (see method `VM_RedefineClasses::adjust_cpool_cache`). We clear all field entries (see method `constantPoolCacheOopDesc::adjust_entries`) and adjust method entries. For method entries we either change the pointer to the method or adjust the virtual table index (see method `ConstantPoolCacheEntry::adjust_method_entry`).

> **TODO:** Improve performance by only updating those field entries that are affected by redefinition. Do not clear fields, but adjust the field offset.

> **TODO:** Improve performance by providing the method with a more efficient data structure for performing a method (or field) lookup (e.g. a hash table of method oops). Additinally, the entries should be adjusted only once and not for each class.

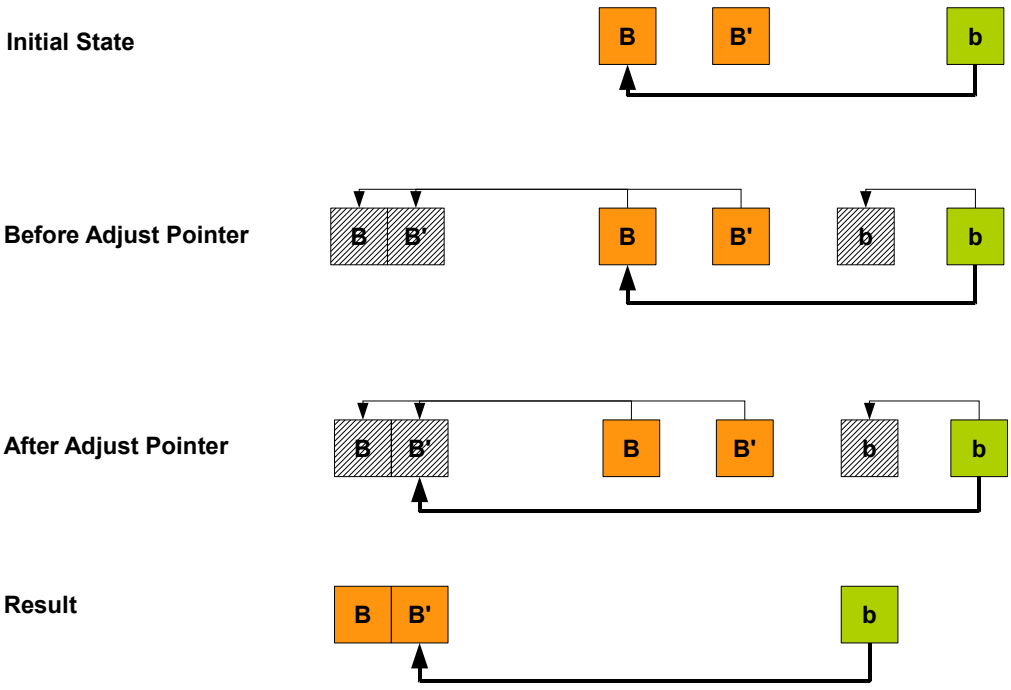

Figure 5: Swap pointers (B, B').

Swap pointers. In a mark and compact garbage collection run, we change pointers to the old class to point to the new class. The pointer replacements are registered before the GC run (see method `Universe::add_oop_replacement`). Figure 5 shows the steps performed in order to swap the pointers. In the adjust pointer phase, the pointer of b to B would usually be adjusted to the new location of B after the garbage collection. We adjust it to the new location of B' such that after garbage collection b points to B'.

We do not adjust all pointers, as we want to keep certain pointers pointing to the old class (e.g. pointers from an old class to another old class). Therefore there is one version of adjusting pointers that keeps them (see method `MarkSweep::adjust_pointer_no_replacement`) and one that swaps them (see method `MarkSweep::adjust_pointer`).

> **TODO:** Use a better data structure for saving the pairs that should be swapped (e.g. an oop hashtable). Currently a linear scan over all saved pairs is necessary at each pointer adjustment.

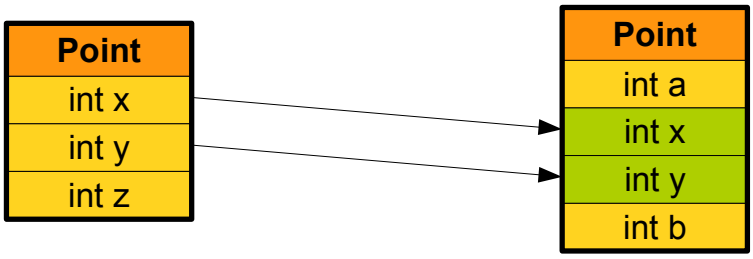

Figure 6: Matching fields of old and new version.

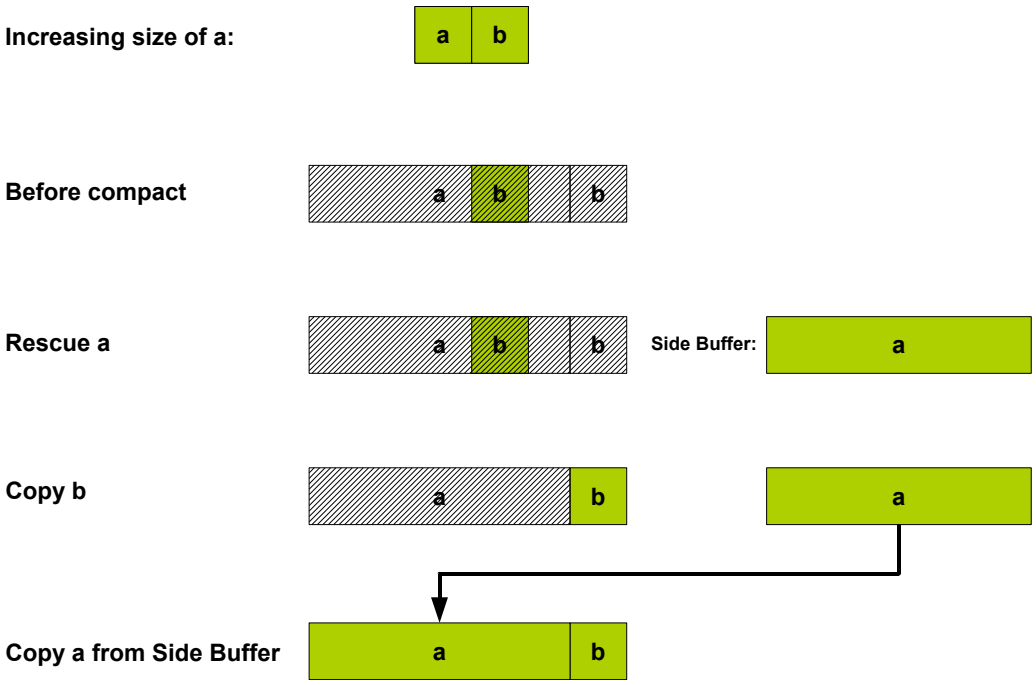

Figure 7: Garbage collection run with increased object sizes.

Update instance fields. When the fields of a class change, we need to updated the instances of a class. This is done during the same garbage collection run that swaps the class pointers. There is a simple strategy for perfoming the update: Generally all new fields are initialized to zero. Only when there is a field with the same name and signature in the old class, the field value is copied from the old version of the object. Figure 6 illustrates this. There is a method to calculate the field matching (see method `instanceKlass::do_fields_evolution`). During garbage collection, we apply the algorithm to each object (see method `MarkSweep::update_fields`).

TODO: Cache the field matching such that subsequent calls to the method do not have to scan over all fields. Currently this scan is performed for each object instance, while it has to be performed only once per redefined class. There is a placeholder that should store the information (see class `CodeEvolution`).

Rescue big objects As objects can get bigger (e.g. when a field is added to a class), we need to rescue objects during the compact phase of the garbage collector. Otherwise objects would be overwritten by other objects. Figure 7 shows how this is handled. In the example, the size of a was increased such that in the compact phase, the next object b would be overwritten. We copy a to a side buffer (see method `CompactibleSpace::rescue`) and copy it back after the garbage collection run finished.

TODO: Currently this can mean in the worst case, that the whole heap needs to be rescued. When the rescued objects are later copied to the end of the heap, the other objects can move further in front. This would decrease the probability for later objects to need a rescue too and also gives a better worst case scenario.

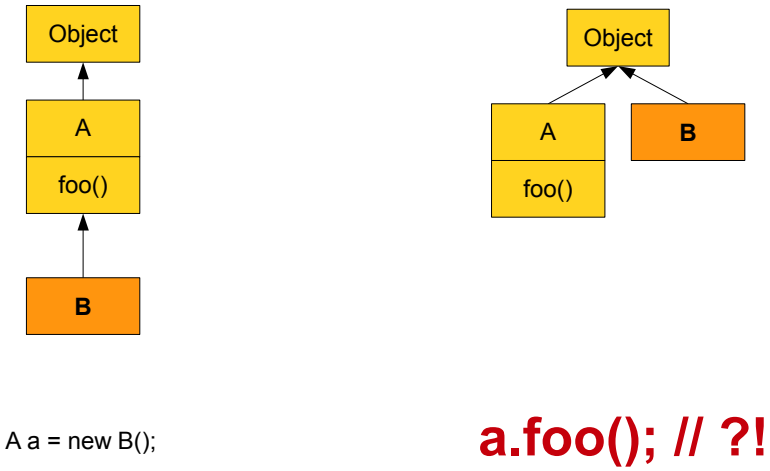

Figure 8: Problems with structural changes.

2.3 Logical Problem with Structural Changes

Figure 8 shows a problem that can occur, when arbitrary changes are allowed. The Java statement schon on the left is correct in the old version of the class hierarchy. In the new hierarchy the classes A and B are no longer related, but the variable a will still contain a reference to an instance of class B.

TODO: Find a solution such that the virtual machine does react appropriately. One possibility could be to disallow the class redefinition in this case. A second possibility could be to set all such variables to `null`.

2.4 Modifications of the VM

This section describes modifications in the VM that might have an effect on normal execution.

Constant Pool Cache Modifications When creating an entry for a virtual method call, usually only the virtual table index is stored (see method `ConstantPoolCacheEntry::set_method`). We changed this such that also a reference to the statically known method is stored. This is necessary to be able to update virtual table indices later on.

Garbage Collector Modifications When adjusting pointers, we need to check, whether the adjusted pointer is one that needs to be swapped with another one. A simple check is necessary even, when no pointers are swapped (see method `MarkSweep::adjust_pointer`).

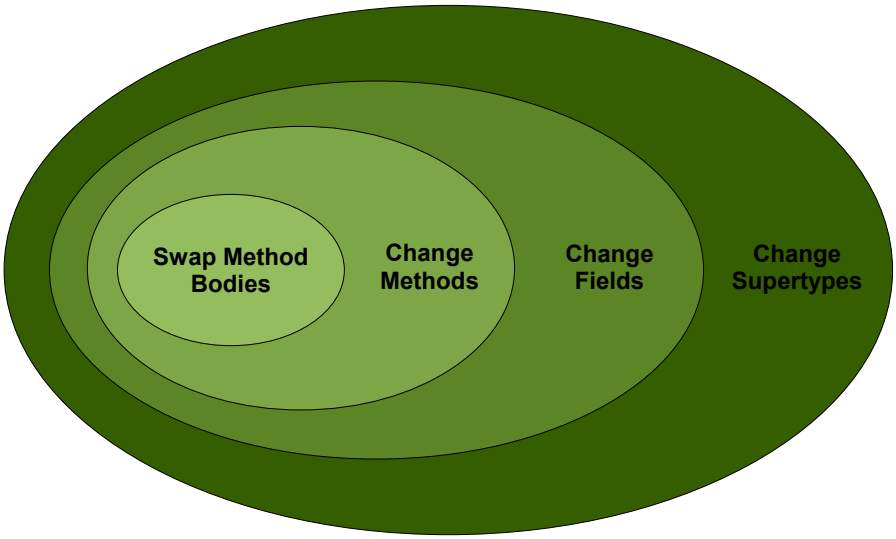

Figure 9: Levels of dynamic code evolution.

3 Tests

There exist different levels of unit tests as shown in Figure 9. The base level contains only tests that do not change any class signature, but only swap method bodies (see class `BodyTestSuite`). The next level modifies only the methods of a class (see class `MethodTestSuite`). The third level allows also modifications to the fields, but does now allow changes to the super types of a class (see class `FieldTestSuite`). The final level allows arbitrary changes (see class `StructuralTestSuite`).

3.1 Running Tests

To run the tests, the following command should be executed:

```
java
-classpath .;%JUNIT%/junit-4.5.jar;%BCEL%/bcel-5.2.jar;%JAVA_HOME%/lib/tools.jar;
           ../../../HotSwapTool/dist/HotSwapTool.jar
-Xdebug
-Xrunjdwp:transport=dt_socket,server=y,address=4000,suspend=n
at.ssw.hotswap.test.Main
```

The working directory should be the subdirectory `hotswaptest/HotSwapTests/build/classes` of the hotspot sources. The variables `JUNIT`, `BCEL` and `JAVA_HOME` must be set accordingly. The Java debug agent must be started on port 4000 as this is currently the port that the class redefinition utility tries to connect to.

3.2 Class Redefinition Utility

There is a utility for redefining classes (see class `HotSwapTool`). This class can be used to redefine all inner classes of a given outer class to a certain version number. The version number must be encoded in the class name with three preceding underscores (e.g. `A___2` for class `A` version 2). If a class name does not contain the three preceding underscores, the class is treated as version 0. Now a call to method `HotSwapTool.toVersion(Class, int)` with a reference to the outer class and a version number, redefines all inner classes to a certain number. Byte code rewriting is used to remove the encoded version number in the class files such that `A` and `A___2` are really treated as the same classes.