# Unrestricted and Safe Dynamic Code Evolution for Java[☆]

Thomas Würthinger[a], Christian Wimmer[b], Lukas Stadler[a]

[a] *Institute for System Software, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University Linz, Austria*
[b] *Department of Computer Science, University of California, Irvine*

**Abstract**

Dynamic code evolution is a technique to update a program while it is running. In an object-oriented language such as Java, this can be seen as replacing a set of classes by new versions. We modified an existing high-performance virtual machine to allow arbitrary changes to the definition of loaded classes. Besides adding and deleting fields and methods, we also allow any kind of changes to the class and interface hierarchy. Our approach focuses on increasing developer productivity during debugging, but can also be applied for updating of long-running applications. Changes can be applied at any point a Java program can be suspended.

Our virtual machine is able to continue execution of old changed or deleted methods and also to access deleted static fields. A dynamic verification of the current state of the program ensures type safety of complex class hierarchy changes. However, the programmer still has to ensure that the semantics of the modified program are correct and that the new program version can start running from the state left behind by the old program version.

The evaluation section shows that our modifications to the virtual machine have no negative performance impact on normal program execution. The in-place instance update algorithm is in many cases faster than a full garbage collection. Standard Java development environments automatically use the code evolution features of our modified virtual machine, so no additional tools are required.

*Keywords:* Java, virtual machine, class hierarchy, run-time evolution, dynamic software updating, safe dynamic updates

## 1. Introduction

Updating the code of a running program has been investigated early in programming history [16]. With the introduction of virtual machines (VMs), the possibilities for dynamic code evolution increased because of the additional layer between the executing program and the hardware. Nevertheless, support for this feature in current production-quality VMs is limited. The ability to evolve the code of a running program has advantages in several areas. We distinguish four main applications of dynamic code evolution and their specific requirements:

**Debugging.** When a developer frequently makes small changes to an application with a long startup time, dynamic code evolution significantly increases productivity. After modifying and compiling the program, the developer can resume it directly from where it was suspended instead of stopping and restarting it. For example, modifying the action performed by a button in a graphical user interface no longer requires the whole program to be closed. The main requirement for success is that the code

[☆] This work was supported by Oracle.

*Email addresses:* wuerthinger@ssw.jku.at (Thomas Würthinger), cwimmer@uci.edu (Christian Wimmer), stadler@ssw.jku.at (Lukas Stadler)

evolution step can be carried out at any time and the programmer does not need to perform additional work, e.g., provide transformation methods for converting between the old and new version of object instances or specify update points. The performance of program execution is also important because an application being debugged should behave as in a production environment (full-speed debugging).

**Server Applications.** Critical applications that must not be shut down can only be updated to the next version using dynamic code evolution. The focus lies on the safety and correctness of an update. We believe that this can only be achieved by designing an application with code evolution in mind and restricting updates to certain predefined points. The server applications must not be slowed down before or after performing the code evolution.

**Dynamic Languages.** There are various efforts to run dynamic languages on statically typed VMs (see for example [37]). Dynamic code evolution is a common feature of dynamic languages. VM-level support for dynamic code evolution simplifies the implementation of dynamic languages. The requirement here is that small incremental changes, e.g., adding a field or method, can be carried out fast.

**Dynamic AOP.** Dynamic code evolution is also a feature relevant for aspect oriented programming (AOP). Several dynamic AOP tools use the limited possibilities of the current Java HotSpot™ VM for dynamic code evolution [7, 52]. These tools can immediately benefit from enhanced code evolution facilities.

Our approach to dynamic code evolution focuses on improving developer productivity during debugging, but can also be applied to the other application scenarios. It can carry out the change at any point a Java program can be suspended, i.e., at any point a developer can set a breakpoint. Additionally, a Java program can be paused by requesting that every thread stops at the next *safepoint*. These points are usually used to suspend all threads before a garbage collection run. The Java VM guarantees that at any point during program execution, all threads reach the next safepoint within a finite time span. Once the VM is suspended, code evolution can be performed.

The dynamic modification of a program can yield results that differ from the results we would obtain when we stop the program, recompile the modified version, and restart it. It is not even possible to find out whether the state of the program at the time of the redefinition is a reachable state in the new version of the program [23]. Our approach allows the code evolution to happen at any time. The VM guarantees that the new program executes according to the Java VM specification [34], but the programmer has to ensure that the new program version is capable of starting execution with the state of the old program.

The strong need for advanced dynamic code evolution features is also expressed by the votes for enhancement requests for the Java HotSpot™ VM [40]: The request for improving the current support for code evolution, which allows only swapping method bodies, is within the top five enhancement requests. Additionally, the increased productivity through fast code modifications is considered one of the advantages of dynamic languages compared to statically typed languages such as Java. Dynamic code evolution increases productivity for the Java programming language too. Our VM has been used to improve dynamic AOP [54] as well as for a case study that modifies the GUI builder of the NetBeans IDE to allow for on-the-fly changes to running Java dialogs [53].

The main contributions of this paper are:

- We describe the modifications necessary for dynamic code evolution in a production-quality VM.
- Our approach allows arbitrary changes, including the modification of subtype relationships. Nevertheless, it does not introduce any memory indirections and is without performance loss before and after the code evolution step.
- We allow co-existence of old and new code.
- An update is possible at any point a Java program can be suspended.
- Our modified version of the Java HotSpot™ VM can be used from within any Java IDE that uses the standard Java Debug Wire Protocol (JDWP), e.g., NetBeans or Eclipse.

In addition to our previous conference paper [55], this extended journal version contributes the following new elements:

- A solution that allows accesses to deleted methods and deleted static fields and therefore allows more programs to continue regular execution without throwing runtime exceptions.
- A new configuration that allows the redefinition only if continued execution is guaranteed to not throw exceptions because of deleted fields or methods.
- We present an algorithm that checks the safety of type narrowing changes.
- We implemented a significant performance improvement for changes that do not increase the size of object instances. The evaluation section is updated to account for the improvements.

## 2. Levels of Code Evolution

Several classifications of run-time changes have been published [44, 24]. From the aspect of implementation complexity and impact on Java semantics, we propose the distinction of four levels of code evolution as shown in Figure 1:

**Swapping Method Bodies:** Replacing the bytecodes of a Java method is the simplest possible change. No other bytecodes or type information data depend on the actual implementation of a method. Therefore, this change can be done in isolation from the rest of the system.

**Adding or Removing Methods:** When changing the set of methods of a class, the virtual method table that is used for dynamic dispatch needs to be modified. Additionally, a change in a class can have an impact on the virtual method table of a subclass (see Section 4.2). The virtual method table indexes of methods may change and make machine code that contains fixed encodings of them invalid (see Section 4.8). Machine code can also contain static linking to existing methods that must be invalidated or recalculated.

**Adding or Removing Fields:** Until this level, the changes only affected the metadata of the VM. Now the object instances need to be modified according to the changes in their class or superclasses. The VM needs to convert the old version of an object to a new version that can have different fields and a different size. We use a modified version of the mark-and-compact garbage collector in order to increase the size of objects (see Section 4.7). Similarly to virtual method table indexes, field offsets are used in various places in the interpreter and in the compiled machine code. They need to be correctly adjusted or invalidated.

**Adding or Removing Supertypes:** Changing the set of declared supertypes of a class is the most complex dynamic code evolution change for Java. For a class, this can imply changes to its methods as well as its fields. Additionally, the metadata of the class needs to be modified in order to reflect the new supertype relationships.

When a developer changes the signature of a method or the type or name of a field, the VM sees the change as two operations: a member being added to and another being deleted from the class. Modifications to interfaces can be treated in a similar way as modifications to classes. Adding or removing an interface method affects subinterfaces and the interface tables of classes which implement that interface, but has no impact on instances. Changes to the set of superinterfaces have a similar effect.

Adding a new Java class is not considered a code evolution change because the dynamic class loading capabilities are part of the base functionality of every Java VM. Another possible kind of change in a Java class is modifying the set of static fields or static methods. This does not affect subclasses or instances, but may invalidate existing code, e.g., when it contains static field offsets. Additionally, a code evolution algorithm needs to decide how to initialize the static fields: either run the static initializer of the new class or copy values from the static fields of the old class.

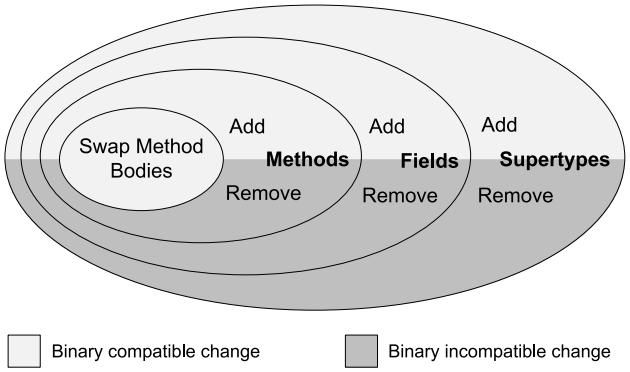


Figure 1: Levels of code evolution.

Changes to Java programs can also be classified according to whether they maintain binary compatibility between program versions or not [11]. The light grey areas of Figure 1 represent binary compatible changes; the dark grey areas indicate binary incompatible changes. With binary compatible changes, the validity of *old code* is not affected. We define old code as bytecodes of methods that have either been deleted or replaced by a method with different bytecodes in the new version of the program. When an update is performed at an arbitrary point, a Java thread can be in the middle of executing such a method. Therefore, old code can still be executed after performing the code evolution step.

Binary incompatible changes to a Java program may break old code. The semantics of instructions that were valid in the old version but are no longer valid in the new version of the program are not clear as neither the Java language specification [19] nor the Java VM specification [34] takes code evolution into account. We classify binary incompatible changes as follows. Section 5 and Section 6 describe how we handle the case where old code is not compatible with new types.

**Removing Fields or Methods:** The bytecodes of deleted or replaced methods can contain references to class members that no longer exist in the new version of the program. When the VM reaches such bytecodes during continued execution of old code, it needs to decide what to do when deleted methods are called or deleted fields are accessed.

**Removing Supertypes:** When narrowing the type of a class, an important invariant during Java program execution can be violated: The static and dynamic type of a variable may no longer have a subtype relationship. Additionally, the receiver object of a dynamic call need no longer be compatible with the class of the called method.

## 3. Architecture of the Java HotSpot™ VM

We implemented our approach as a modification to the Java HotSpot™ VM [39]. This section gives a brief overview and introduces the main subsystems of the current product version. The overall architecture is shown in Figure 2.

Java class files are parsed and loaded by a class loader and then inserted into the *system dictionary*. The VM maintains this system dictionary to look up classes based on their name and class loader. The qualification using name and class loader is necessary because two class loaders can load two different classes with the same name. The class loader is also the granularity for class unloading, i.e., when a class loader and all its loaded classes are no longer reachable, they can be unloaded from the VM.

The execution of a Java program starts in the interpreter, which steps through the bytecodes of a method and executes a code template for each instruction. Only the most frequently called methods, referred to as *hot spots*, are scheduled for *just-in-time (JIT) compilation*. As most classes used in a method are loaded

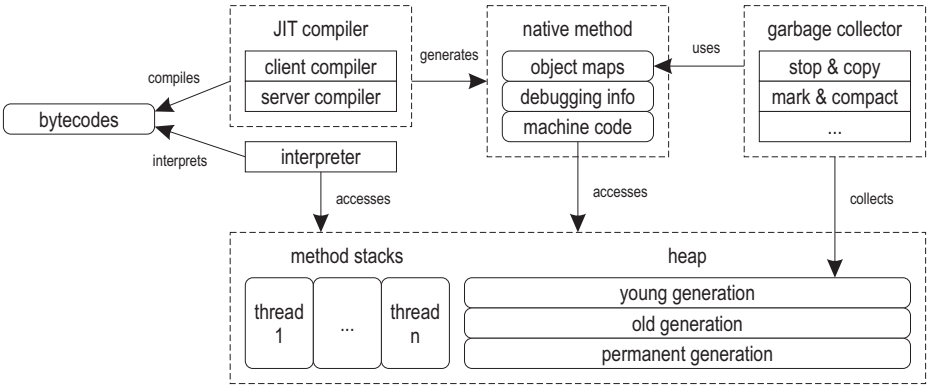


Figure 2: Architecture of the Java HotSpot$^{TM}$ VM

during interpretation, information about them is already available at the time of JIT compilation. This information allows the compiler to inline more methods and to generate better optimized machine code.

The Java HotSpot$^{TM}$ VM has two alternative just-in-time compilers: the *server compiler* and the *client compiler*. The server compiler [42] is a highly optimizing compiler tuned for peak performance at the cost of compilation speed. Low compilation speed is acceptable for long-running server applications, because compilation impairs performance only during the warm-up phase and can usually be done in the background if multiple processors are available. For interactive client programs with graphical user interfaces, however, response time is more important than peak performance. For this purpose, the client compiler [31] achieves a trade-off between the performance of the generated machine code and compilation speed.

The generational garbage collector [50] of the Java HotSpot$^{TM}$ VM manages dynamically allocated memory. It uses exact garbage collection techniques, so every object and every pointer to an object must be precisely known at GC time. This is essential for supporting compacting collection algorithms. The memory is split into three generations: a young generation for newly allocated objects, an old generation for long-lived objects, and a permanent generation for internal data structures.

New objects are allocated sequentially in the young generation. Since each thread has a separate *thread-local allocation buffer* (TLAB), allocation operations are multithread-safe without any locking. When the young generation fills up, a *stop-and-copy* garbage collection is initiated. When objects have survived a certain number of collection cycles, they are promoted to the old generation, which is collected by a *mark-and-compact* algorithm [29].

Exact garbage collection requires information about pointers to heap objects. For machine code, this information is contained in *object maps* (also called *oop maps*) created by the JIT compiler. Besides, the compiler creates *debugging information* that maps the state of a compiled method back to the state of the interpreter. This enables aggressive compiler optimizations, because the VM can *deoptimize* [28] back to a safe state when the assumptions under which an optimization was performed are invalidated. The machine code, the object maps, and the debugging information are stored together in a so-called *native method object*. Garbage collection and deoptimization are allowed to occur only at some discrete points in the program, called *safepoints*, such as backward branches, method calls, return instructions, and operations that may throw an exception.

The interface between a Java VM and a Java debugger is specified in the Java Debug Wire Protocol (JDWP) [38]. A VM implementing this interface is immediately usable from within standard Java development environments that use the JDWP protocol for debugging Java applications, for example NetBeans or Eclipse. One of the JDWP commands redefines classes that were previously loaded by the VM. The current implementation of this command in the Java HotSpot$^{TM}$ VM allows only the swapping of method bodies [10], i.e., the first level of changes defined in Section 2. It is not possible to add or remove methods and fields, and it is not possible to change the class hierarchy. The VM rejects such changes, i.e., the JDWP command fails with an error message and the loaded classes remain unmodified. We extend this command to

allow arbitrary changes to loaded types. This approach makes our implementation accessible to all existing JDWP clients.

## 4. Implementation

Our approach focuses on implementing unrestricted code evolution in the Java HotSpot VM while keeping the necessary changes small. In particular, we do not modify any of the just-in-time compilers or the interpreter. Our changes affect the garbage collector, the system dictionary, and the class metadata. However, they are small and do not influence the VM during normal program execution.

Figure 3 gives an overview of the modifications to the VM that are described in the following subsections. The code evolution is triggered by a JDWP command. First, the algorithm finds all affected classes and sorts them based on their subtype relationships. Then, the new classes are loaded and added to the type universe of the VM, forming a side universe. A heap iteration swaps pointers and also updates instances if their size did not increase. The modified full garbage collection is only necessary if at least one instance has an increased size. After invalidating state that is no longer consistent with the new class versions, the VM continues executing the program.

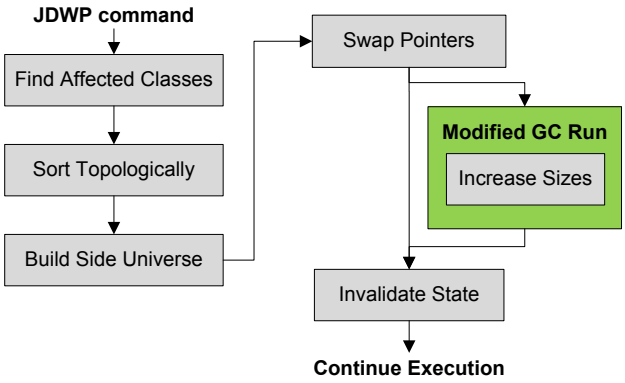


Figure 3: Steps performed by the code evolution algorithm.

### *4.1. Class Redefinition Command*

We use the existing JDWP command for class redefinition to trigger dynamic code evolution. The command requires that all redefined classes are already loaded in the VM. If a class is not yet loaded, redefinition is not necessary. The new version can be loaded as the initial version of the class. For each class, a number identifying the class and an array with the new class bytes is transmitted. Our modified VM implements this command exactly based on its specification and does not require additional information to perform the code evolution.

The first steps of the redefinition (described in the next three sections) can be done concurrently to normal program execution. Only the subsequent garbage collection run that performs the instance updates needs to stop all running Java threads. We use the same safepoint mechanism as the garbage collector to suspend all active threads.

### *4.2. Finding Affected Types*

When applying more advanced changes than just swapping method bodies, classes can be indirectly affected by the redefinition step. A field added to a class is implicitly also added to all its subclasses. Adding a method to a class can have effects on the virtual method tables of its subclasses.

Therefore, the algorithm needs to extend the set of redefined classes with all their subtypes. Figure 4 gives an example with three classes `A`, `B`, and `C`. Class `A` and `C` are redefined, but this also affects class `B` as

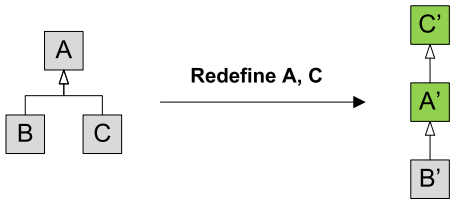


Figure 4: Code evolution example changing the subtype relationship between class `A` and `C`.

it is a subclass of `A`. Class `B` is added to the set of redefined classes and is replaced by `B'`, which has the same class file data as `B`, but possibly different properties inherited from its superclasses. We need to fully reload `B`, because its metadata including the virtual method tables need to be initialized based on the new supertype.

The same rule applies when redefining interfaces. All subinterfaces and also all classes implementing the interface need to be redefined, because adding or removing methods of the superinterface has effects on their interface method tables.

### *4.3. Ordering the Types*

The redefinition command does not specify an order in which the classes must be redefined. From the user's perspective, the classes must be swapped atomically. Our algorithm does a topological sorting of the classes based on their subtype relationships. A class or interface always needs to be redefined before its subtypes can be redefined. The new version of a class could be incompatible with the old version of its superclass. In that case, class loading only succeeds if the superclass was already replaced by its new version.

In order to make changes to the class hierarchy possible, we need to order the types based on their relationship *after* the code evolution step and cannot use the information about their current relationship. Subtype relationship information is available in the VM only after a class has been loaded. Therefore, we parse parts of the class files prior to class loading in order to find out about the new subtype relationships. In the example of Figure 4, we need to first redefine `C` to `C'` and subsequently `A` to `A'`, because in the new version of the program `A` is a subclass of `C`.

### *4.4. Building a Side Universe*

We keep both the old and the new classes in the system. This is necessary to be able to keep executing old code that depends on properties of the old class. It would also open the possibility to keep old and new instances in parallel on the heap. Additionally, it is the only way to solve the problem of cyclic dependencies between code evolution changes, e.g., when one change requires class `A` to be redefined before `B`, but the other one `B` to be redefined before `A`: While adding the new classes, the type universe is always kept in a consistent state, because we build a separate side branch for the new classes. Therefore, the old version of a class does not affect the loading and verification of the new version of another class.

The Java HotSpot™ VM maintains a system dictionary to look up classes based on their name and class loader. We replace the entry for the old class with the entry for the new class immediately after loading the new class. The pre-calculated order in which we redefine classes ensures that the side universe is created correctly. When we load class `A` in the example of Figure 4, the lookup for class `C` returns `C'`, because class `C` was redefined before `A`. The VM copies the value of static fields of the old class to the static fields the new class if both name and signature match. We do not execute the static class initializer of the new class. Figure 5 shows the state of the universe after building the side universe for the new class versions.

We keep the different versions of the same class connected in a doubly linked list. This helps navigating through the versions during garbage collection. The system dictionary, however, always contains just a reference to the latest version of a class.

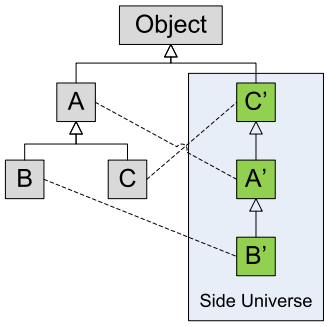


Figure 5: The new state of the type universe after code evolution.

*4.5. Swapping Pointers*

When updating a class `C` to `C'` we must ensure that all instances of class `C` are updated to be instances of class `C'`. The instance of an object on the heap contains a reference to its class. The Java HotSpot™ VM does not keep track of the instances of a given class, therefore a heap traversal is necessary to find all existing instances. Additionally, other parts of the system (e.g., native code) can have references to the old class that need to be updated too. This is an irrevocable step, so all checks whether the redefinition is valid have to be done beforehand. Additionally, this step needs to be carried out atomically.

During the heap traversal, we distinguish three different types of objects:

**Should remain unchanged:** Meta-data class objects should retain their pointers, because updating their pointers to the new version would lead to old classes pointing to new classes (e.g., at the super class field) and therefore destroy the sanity of the old side class hierarchy. Also, the holder of the constant pool should always be the class version corresponding to the constant pool and not the latest version.

**Check object's type:** Java instance objects cannot have fields pointing to VM meta-data objects. Therefore it is sufficient to only check whether the object's header field points to a redefined class.

**Full check:** All other objects can potentially contain pointers to class meta-data objects at any place. The VM performs a full iteration over the object's pointer. Also, the VM performs the pointer update on all root pointers.

The VM also collects information about redefined objects. In particular, it finds out whether there is at least one object instance that has an increased size due to the redefinition. Additionally, it collects a list of redefined objects.

*4.6. Updating Instances*

For updating instances, we need a strategy how to initialize the fields of the new instance. We have a simple algorithm that matches fields if their name and type are the same. For the matching fields, we copy the values from the old instance to the new instance. All other fields are initialized with `0`, `null`, or `false`.

With this approach, we can efficiently encode the memory operations necessary for an instance update (filling an area with zero or copying bytes from the old version of the object). The information is calculated once per class and temporarily attached to the class meta object. The modified garbage collector reads the information and performs the memory copy or clear operations for each instance. This makes instance updates faster compared to other approaches that work with custom transformation methods for converting between the old and new version of object instances. We believe that the programmer wants to provide as little additional input as possible during debugging and so the lost flexibility compared to transformation methods is balanced by the ease of use.

If the size of an instance remains the same (e.g., fields are only reordered), we update the instance data in-place during the heap iteration for updating pointers. In case the instance size is decreased, we also

update the instance data and place a filler object at the end in order to account for the decreased size. Every object in the VM is aligned on 8-byte boundaries, therefore the smallest possible size decrease is 8 bytes. In this case, we can put a never referenced instance of class `java.lang.Object` in the deprecated area, because such an instance fits exactly. For any decrease larger than 8 bytes, we put in a byte array. The length of the array is chosen such that the array exactly fits the gap between the old and new object size.

### *4.7. Increased Instance Sizes*

Increasing the sizes of existing class instances (e.g., because a field was added to the class) is implemented as a modification of the mark-and-compact garbage collection algorithm. The mark-and-compact algorithm calculates a *forward pointer* for each live heap object, which points to the address of the object after the heap compaction. In the following heap traversal, all references are adjusted to point to this newly calculated address instead of the referenced object itself. In the final compaction phase, the objects are moved to their new addresses. As the update from the old to the new version of an instance is carried out during the compaction phase of the garbage collector, we do not need any additional memory for co-existence of old and new instances. The old version of an instance is deprecated immediately after the new version has been created and filled with values.

We adjusted the forward pointer calculation algorithm in order to consider the new object sizes. An additional modification of the garbage collector is necessary to support increasing object sizes. In this case, the instances are not necessarily always copied to lower addresses, which is a necessary condition for the compaction phase of a mark-and-compact garbage collector. Therefore, every instance that would be copied to a higher memory address must first be rescued to a side buffer. Otherwise the garbage collector would overwrite objects that are not yet copied and destroy their field values. After all instances have been either copied or rescued, the side buffer is processed and used to initialize the new versions of the rescued instances. To reduce the number of objects that need to be copied to a side buffer, our forward pointer calculation algorithm places objects that are copied to the side buffer automatically at the very end of the heap. This makes space for other objects to increase their size while still being copied to lower addresses.

In Figure 6, the size of `x` is increased and therefore the object `x` at its destination address overlaps other objects such as `y` and overwrite their contents before they are copied. Our modified forward pointer algorithm detects that `x` is an instance that needs to be rescued and therefore places it at the end of the heap. This makes free space for the destination addresses of the instances `y` and `z` such that they need not be rescued. The optimization to place rescued objects at the end of the heap significantly reduces the number of rescued objects and therefore the necessary size of the side buffer. In the compaction phase, the object `x` is copied to the side buffer, objects `y` and `z` are processed normally. Afterwards, the new version of `x` is constructed based on the data of the old version in the side buffer.

### *4.8. State Invalidation*

The changes performed by code evolution violate several invariants in the VM. The Java HotSpot™ VM was not developed with code evolution in mind and makes assumptions that no longer hold, e.g., that field offsets never change. In this section we outline different subsystems of the VM that need changes in order to prevent unexpected failures due to broken assumptions.

#### *4.8.1. Compiled Code*

Machine code generated by the just-in-time compiler before code evolution needs to be checked for validity. The most obvious potentially invalid information are virtual method table indexes and field offsets. Additionally, assumptions about the class hierarchy (e.g., whether a class is a leaf class) or calls (e.g., whether a call can be statically bound) become invalid.

The Java HotSpot™ VM has a built-in mechanism to discard the optimized machine code of a method, called *deoptimization* [28]. If there is an activation of the method on the stack, the stack frame is converted to an interpreter frame and execution is continued in the interpreter. Additionally, the machine code is made *non entrant* by guarding the entry point with a jump to the interpreter. We can use it to deoptimize

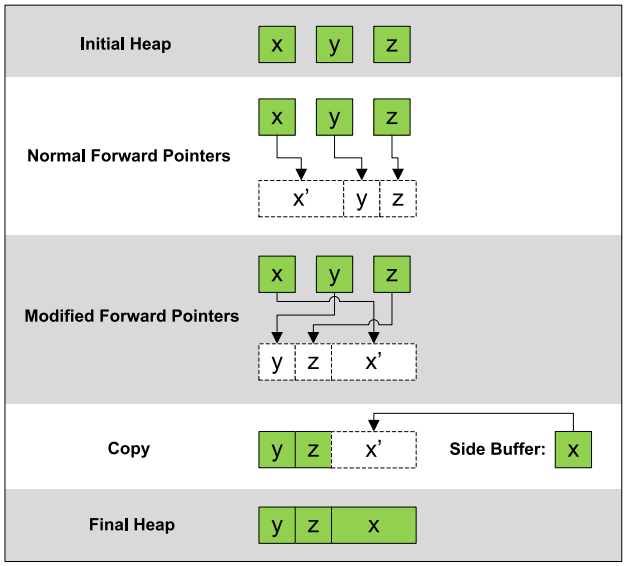


Figure 6: Increasing object size during garbage collection.

all compiled methods to make sure that no machine code produced with wrong assumptions is executed. Analyzing the assumptions made for compiled methods could reduce the amount of machine code that has to be invalidated. However, the evaluation section shows that the time necessary to recompile frequently executed methods is low.

*4.8.2. Constant Pool Cache*

The Java HotSpot™ VM maintains a cached version of the constant pool for a class. This significantly increases the execution speed of the interpreter compared to working with the original constant pool entries stored in the Java class files. The original entries only contain symbolic references to fields, methods, and classes, while the cached entries contain direct references to metadata objects. The entries relevant to code evolution are field entries (the offset of a field is cached) and method entries (for a statically bound call a pointer to the method meta object, for a dynamically bound call the virtual method table index is cached). We iterate over the constant pool cache entries and clear those entries that are possibly affected by code evolution (i.e., that correspond to members of a redefined class). Cleared entries are resolved again by the interpreter. The lookup in the system dictionary automatically returns the new version of the class and therefore the entry is reinitialized with the correct field offset or method destination information.

*4.8.3. Reflection*

Reflection allows a program to query the fields and methods of an object and access them by name. In Java, the reflection system creates objects that represent fields (`java.lang.reflect.Field`) and methods (`java.lang.reflect.Method`). These objects allow to access the fields and to invoke the methods, respectively. Internally, the VM uses optimized strategies to implement this functionality that cache information such as field offsets. These caches need to be invalidated in a similar way to compiled code. Therefore, the VM resolves the information again at the next reflective member access.

After a dynamic code evolution, reflective objects can refer to fields and methods that no longer exist. In this case, a `NoSuchFieldError` or `NoSuchMethodError` is thrown accordingly. This might come as a surprise to the programmer, who expects a reflective access to never fail once the reflective object is created. We consider this a problem that cannot be solved at the VM level. It is necessary to employ an update-friendly programming style where reflective information is not stored across update points.

*4.8.4. Class Pointer Values*

Several data structures in the Java HotSpot™ VM depend on the actual addresses of the class meta objects, e.g., a hash table mapping from classes to JVMTI objects. We need to make sure that these data structures are reinitialized after code evolution. While class objects may also be moved during a normal garbage collection run, our pointer swapping potentially also changes the order of two class objects on the heap. The just-in-time compiler uses a binary search array for compiler interface objects that depends on the order of the class objects and therefore must be resorted after a code evolution step.

## 5. Deleted Fields and Methods

Changes are binary incompatible if they may break old code. As long as only method bodies are swapped or fields and methods are added to classes, the old code can execute normally. It does neither call new methods nor access new fields. However, when a field or method is deleted, old code is possibly no longer valid. In our system, old code may still be executed when old methods are on the stack, so it can happen that old code accesses a deleted field or tries to call a deleted method.

Figure 7 shows an example for this case. The program is paused in method `foo` between the calls to `print` and `bar`. Method `foo` is redefined to a new version `foo'` while method `bar` is deleted. Subsequent calls to method `foo` immediately target the new code, but the old activation of `foo` continues to execute in the old code. Therefore, it reaches the call to the deleted method `bar`.

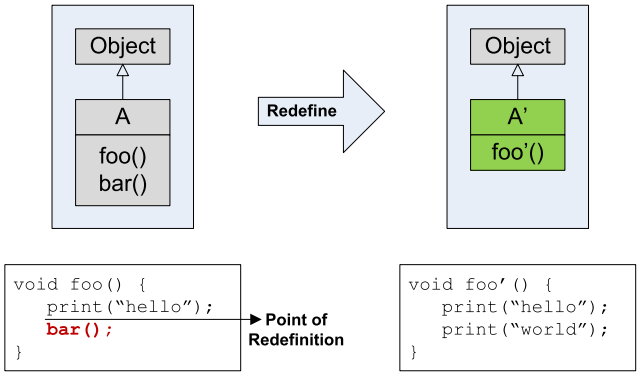


Figure 7: Deleting a method using dynamic code evolution.

The new version of `foo` is correct because it no longer calls `bar`. It is possible to develop heuristics for translating from the stack values and bytecode position in the old method to new stack values and a new bytecode position in the new code. In the general case, however, it is impossible to find a match that is both valid and intuitive for developers.

In our implementation, the old method continues execution in the interpreter. When it reaches the bytecode for the call to `bar`, the reference to `bar` needs to be resolved (because we cleared the constant pool cache during the redefinition step, see Section 4.8). In an earlier version of our system, the resolution fails to find the method and throws a `NoSuchMethodException`. Now we use a more fail-safe way of handling this situation as described in the next section.

*5.1. Executing Multiple Versions*

We distinguish four different possibilities of dealing with the problem of deleted members. The user can set the requested behavior on a per-class level by specifying a custom bytecode attribute in the new version of a redefined class. The behavior is then applied when executing an old method of that class.

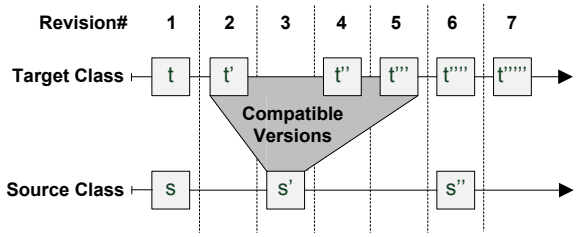


Figure 8: Compatible versions when a method of the source class accesses a member of the target class.

**Static check:** A static reachability analysis checks whether the continued execution of the program can reach an instruction referring to a deleted member. If this can happen, then the class redefinition is rejected. The VM looks at the stacks of all threads at the time of redefinition. If a method is executing a redefined method, then it starts a reachability analysis starting from the currently executed bytecode index. This behavior can block redefinition if a member is deleted that is accessed from a long-running loop.

**Dynamic check:** The redefinition is always performed and the execution of old methods resumes in the interpreter. When the interpreter reaches the bytecode for the call to `bar`, the reference to `bar` needs to be resolved (because we cleared the constant pool cache during the redefinition step, see Section 4.8). The resolution fails to find the method and throws a `NoSuchMethodException`.

**Access deleted members:** This behavior avoids throwing an exception, but accesses the old deleted member. It is only applied to methods and static fields. Accessing deleted instance fields still causes an exception. This is because we remove deleted instance fields from all active instances and therefore cannot access their data after the redefinition step.

**Access old members:** Calling a method of a different version is dangerous if the semantics of the method changed. Therefore our VM also offers a behavior where a member access is always performed from the viewpoint of the version of the executing method. An old method calls the old method version based on the compatible class version even if the method also exists in the latest class version. This behavior can delay the redefinition, because a thread only starts executing the new program version after returning from the last modified method. However, possible functional changes on a method cannot cause semantic problems for the caller.

### *5.2. Access Deleted Members*

We implemented the functionality to access old class members in the interpreter. Old code is always executed in the interpreter, so we do not need to modify the compiler for this enhanced functionality. After class redefinition, the interpreter caches for resolved methods and fields are cleared. When resolving a member again, the interpreter looks for the old deleted version of the member. If the policy is that old members should only be accessed when they do no longer exist in the new version of the program, the interpreter first tries to resolve the member based on the new version. When resolving a member based on the old version, it uses the latest class version of the target class that is compatible with the class of the executed method as shown in Figure 8. Two specific versions of two classes are compatible if they coexisted at any time as the latest version of their class.

In the example figure, $t'$, $t''$, and $t'''$ are all compatible target classes of the source class version $s'$. The class $t'$ is compatible with both $s$ and $s'$ because it coexisted as the latest version with both of them (with $s$ in revision 2 and with $s'$ in revision 3). The interpreter looks up the latest compatible target class. If the class $s'$ declares the method then $t'''$ is used; if the class $s$ declares the method then $t'$ is used as the compatible target class. If the accessed member is also not found in the compatible target class, then throwing an exception is the correct behavior.

For virtual calls and interface calls, the class versions are looked up based on the concrete class of the receiver. Usually, the interpreter would save the table index for such calls in the constant pool cache to be faster on subsequent executions of the bytecode. However, in this case receiver objects with different concrete types may resolve to different table indexes. Therefore, the table index must be resolved every time.

## 6. Type Narrowing

When the set of implemented interfaces or the set of all superclasses of a class increases, old code can execute as normal. It does not use the newly added interfaces or classes, but executes as before. On the other hand, when the set is narrowed, old code is possibly no longer valid.

Figure 9 gives an example for this case. Class `B` is redefined to no longer extend class `A` but directly inherit from `Object`. Now, instances of `B` must no longer be treated as instances of `A`. There could already be variables of type `A` referencing instances of `B` as shown in the code listing. After code evolution, the values of such variables become invalid, because `B` is no longer a subtype of `A`. In the listing of Figure 9, the call `a.foo()` no longer makes sense, because `B` does neither declare nor inherit a method `foo`.

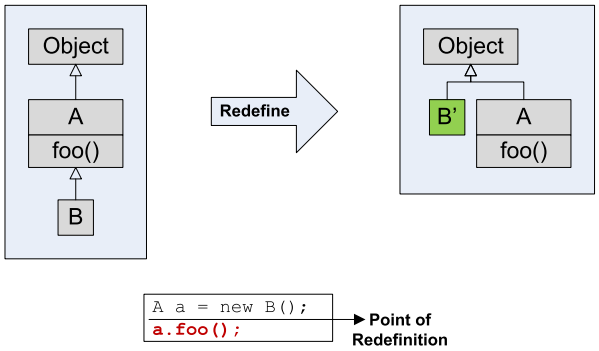


Figure 9: Type hierarchy change using dynamic code evolution.

An old version of our system correctly performs the code evolution step, but the call to `foo` leads to termination of the VM. Now we have a dynamic check that guarantees the safety of a type narrowing change.

In order to allow the safe removal of a supertype, the VM needs to guarantee that the subtype relationship between the static and dynamic type of values is valid after the change. Additionally, it has to re-verify methods to make sure that the relationship is not violated in continued execution. For increasing the performance of the algorithm, the VM places two boolean flags on the types: A type is *narrowed* if one of its original supertypes is no longer a supertype in the new version of the type. A type is *relevant* with respect to the type narrowing check, if it was removed from the set of subtypes of a narrowed type.

Figure 10 shows a type narrowing example with four Java types `A`, `B`, `C`, and `D` involved. The classes `C` and `D` are redefined. The set of supertypes of `C` is narrowed, because `B` is no longer a supertype of `C` in the new version. The set of supertypes of `D`, however, remains unchanged. In both versions `A`, `B`, and `C` are supertypes of `D`. Therefore, instances of class `D` can never violate the relationship between dynamic and static type after the redefinition. For checking values, interface `B` is marked as relevant, because it was removed from the supertype set of `C`. All other classes were not removed from any supertype set.

To check all objects on the heap, the VM calculates a check map for each class that specifies at which field offset a pointer needs to be checked. Only fields with a declared type that is relevant are added to the map. A class with an empty check map means that its instances are ignored in the heap iteration. When iterating over the fields of an object, the dynamic type of a field can also be used as a shortcut: If the dynamic type was not narrowed, then the check is not needed. The VM performs a costly subtype check only if it is still necessary after taking the declared type and the dynamic type into account. Object arrays are processed by iterating over their elements if their declared element type is relevant.

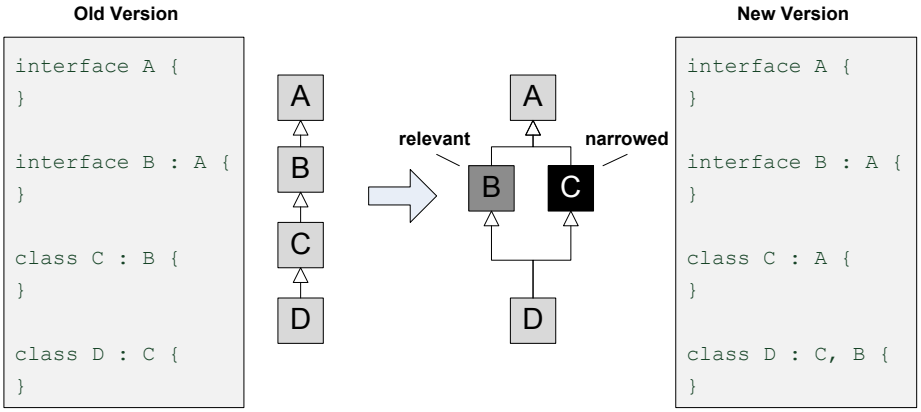


Figure 10: Type narrowing example.

For checking the local variables and the expression stack, the VM walks over the execution stacks of all threads. Each stack frame value of object type is checked. If its dynamic type is a narrowed type, the VM tries to determine the declared type of the value. This is only possible for local variables listed in the method's `LocalVariablesTable` attribute. If the declared type is not available, then the VM conservatively rejects the redefinition when an instance of a narrowed type is found in a stack frame.

In addition to validating values, the VM also has to make sure that loaded bytecodes still pass verification taking the new class hierarchy into account. The verifier relies on subtype relationships (e.g., when assigning a local variable to a field) and therefore a change in the hierarchy can also change the verification result. In order to avoid the expensive verification of all methods, we do a quick check of the constant pool of a class. If its constant pool does not contain a literal reference to a narrowed type, the re-verification of the methods of this class can be skipped.

Only if all checks are positive, the type narrowing class redefinition is performed. Otherwise, the VM gracefully fails by rolling back any changes and returning an error code.

## 7. Semantics of Dynamically Modified Applications

Our approach is primarily designed for being used during debugging where occasional rejected updates or unexpected program semantics are acceptable. However, we claim that our approach is also suitable for being used in production systems. The VM provides safe and predictable semantics for binary incompatible changes. As shown in Section 5, the developer can select between different behaviors for dealing with deleted members. Only the "dynamic check" behavior can result in exceptions and is therefore restricted to debugging scenarios. The "static check" behavior suppresses dynamic code evolution altogether if the continued execution of the program can reach an instruction referring to a deleted member. Also, the other two behaviors rule out exceptions caused by deleted methods and static fields.

Dynamic modification of a program can yield results that differ from the results one would obtain when stopping the program, recompile the modified version, and restart it. But this is inherent to dynamic code evolution and not a restriction of our approach. At the time of a dynamic modification, the program is in a state that was established by the old version of the program. Dynamic code evolution allows running the new program version against the old state, which is sometimes even necessary in long-running server applications. Of course, the code of the new version has to be written in such a way that it can cope with the old state. For this, it is helpful when the old code was already written with dynamic updates in mind. The rule of thumb should be to avoid duplication of state—because dynamic updates could affect duplicates in different ways and make them inconsistent—as well as to avoid caching of intermediate results—because dynamic updates cannot transform them. Additionally, introspection of the application itself, e.g., using

reflection (see Section 4.8.3), is problematic since updates change structures that the application might consider invariant.

Our VM provides the programmer with the necessary tools to perform the dynamic modification safely and predictably. The programmer needs to be aware that the change to the new version of the program could happen at any time during execution of the old program. However, when the features provided by our VM are used in a controlled way, they allow updates of long-running applications without any negative impact on the peak performance of the application before and after the update.

## 8. Evaluation

We evaluate our implementation by looking at it from three sides: First, we discuss our support for different levels of code evolution. Second, we show that our modified VM produces benchmark results similar to the unmodified baseline. After performing a code evolution step the original peak performance is reached again. Finally, we present timing results for micro benchmarks to discuss the performance characteristics of our garbage collector modifications.

### *8.1. Functional Evaluation*

Our implementation supports all possible modifications to a set classes. There are two restrictions: When a deleted instance field is accessed by old code, an exception is thrown. Additionally, a type narrowing change is not carried out if the dynamic type of a variable would no longer match its static type after redefinition. In all cases, the continued execution of the Java program is fully compliant with standard Java semantics. Table 1 gives an overview of the supported changes classified as discussed in Section 2.

| Method | Restrictions |
|---|---|
| Swap Method Body | |
| Add Method | |
| Remove Method | |
| Add Field | |
| Remove Field | `NoSuchFieldError` when accessing a deleted instance field |
| Add Supertype | |
| Remove Supertype | only allowed when dynamic type safety verification succeeds |

Table 1: Supported code evolution changes.

When debugging an application, possible problems after resuming the program are more acceptable than when updating a server application. The worst case scenario is that the developer needs to restart the application. Without code evolution, this would be necessary anyway. The semantics of the problematic instructions are not clear as the Java standard was not designed with code evolution in mind. Therefore, we believe that throwing an exception (in case a deleted instance field is accessed) is an acceptable behavior and leads to less confusion compared to trying to hide and absorb the problem. Our algorithm for accessing old deleted methods and old deleted static fields (see Section 5.2) can safely handle many cases of binary incompatible changes.

Type narrowing changes where a supertype is removed from a class are rare during development compared to other kinds of changes. Our type safety verification performs a safe rollback and gives an error message when such a change turns out to be unsafe (see Section 6). This is a significant enhancement compared to an earlier VM version that could not guarantee continued execution without a VM crash [55].

Since version 1.4 of Java, the JPDA (Java Platform Debugger Architecture) defines commands for class redefinition. A VM specifies three flags to inform the debugger about the code evolution capabilities: `canRedefineClasses` if class redefinition is possible at all, `canAddMethod` if methods can be added to

classes, and `canUnrestrictedlyRedefineClasses` if arbitrary changes to classes are possible. To the best of our knowledge, our modified version of the Java HotSpot™ VM is the first VM that can return `true` for all three flags. Based on the considerations in Section 2 about the implementation complexity of changes in the VM, we propose a more fine grained distinction between different levels of code evolution. The step between adding methods and allowing arbitrary changes is too coarse grained.

It is difficult to measure the usage characteristics of code evolution as it heavily depends on the application domain and also on the developer behavior. Gustavsson [24] published a case study in which the updates to a web server between different versions are examined. The result is that 37% of the modifications only redefine method bodies, 16% only add or remove methods, 33% are arbitrary code evolution changes, and 14% are changes that cannot be performed, e.g., because of code that never becomes inactive or a need to change things outside the VM. Our implementation can therefore increase the percentage of possible changes in this case study from 37% to 86%.

### *8.2. Effects on Normal Execution*

We use two benchmarks selected from the DaCapo benchmark suite [5] with different characteristics regarding the warm-up phase. We show that our modifications to the VM have no effects on normal program execution. Additionally, after a code evolution step, the application reaches the peak performance again. We measure the times of 24 subsequent benchmark runs within the same VM. A garbage collection between two subsequent runs is performed in order to reduce the random noise introduced by the garbage collector. In our modified VM, we redefine a single class between the 8th and 9th and between the 16th and 17th run, causing the VM to discard previously compiled machine code as described in Section 4.8.

Our implementation is based on a recent early access development build of the Java HotSpot™ VM for Java 7. The baseline run is performed by the unmodified version of the same early access Java HotSpot™ VM. We use the same command line flags for both VMs. All performance measurements were performed on an Intel Core™2 Quad CPU with 2.40 GHz per core and 8 GByte memory. The operating system is a 32-bit version of Windows Vista.

The heap size is specified with 1 GByte and the client compiler is used as the just-in-time compiler. Additionally, we use the following command line flag:

```
-Xrunjdwp:transport=dt_socket,
server=y,address=4000,suspend=n
```

This starts a JVMTI agent for receiving JDWP commands. The agent is used for debugging the Java program running in the VM. Later, we can connect to this agent and send the commands for triggering code evolution.

We executed the test setup 20 times and calculated the mean. Figure 11 shows the results for the unmodified reference VM without code evolution (dotted line) and our modified VM (solid line) when executing two DaCapo benchmarks.

The performance characteristics after a code evolution step are similar to the warm-up phase. The first run after the code evolution is significantly slower. This is because the VM needs some time to recompile the frequently executed methods again. In the second run, the performance difference is hardly measurable and subsequent runs do not show any difference. As the profiling information is reused, the first run after code evolution is faster than the first run overall. For the `chart` benchmark, the slowdown of the first run compared to the peak performance is about 15%, but the slowdown of the first run after code evolution is only about 3%.

### *8.3. Micro Benchmarks*

To measure the performance of instance updates and our garbage collector adjustments, we use three micro benchmarks in which we increase and decrease the field count and reorder the fields of a class and update all instances of this class to the new version. We compare the performance of this redefinition to the performance of a normal garbage collection run. Table 2 describes the different class versions used for the benchmarks. The leftmost column contains the initial version of the class. It has three `int` fields and three

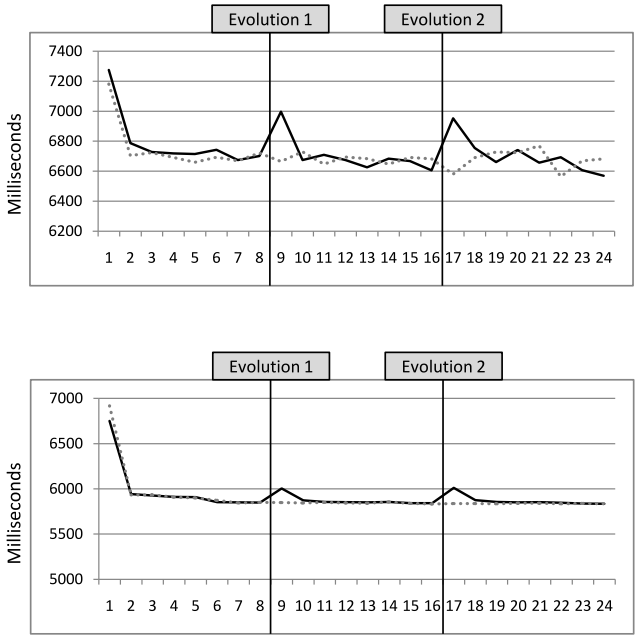


Figure 11: Executing 24 runs of the `bloat` (top) and the `chart` (bottom) benchmark. The dotted line represents an unmodified reference VM, the continuous line our code evolution VM.

`Object` fields resulting in a total object size (including 8 bytes object header) of 32 bytes. Note that the size of the object header can vary between different platforms and virtual machines. All three modifications are applied to the initial version, i.e., they are not cumulative. The three benchmark configurations are:

| Initial | Increase | Decrease | Reorder |
|---|---|---|---|
| `class C {`<br>`int i1;`<br>`int i2;`<br>`int i3;`<br>`Object o1;`<br>`Object o2;`<br>`Object o3;`<br>} | `class C' {`<br>`int i1;`<br>`int i2;`<br>`int i3;`<br>`Object o1;`<br>`Object o2;`<br>`Object o3;`<br>`Object o4;`<br>} | `class C' {`<br>`int i1;`<br>`int i2;`<br>`int i3;`<br>`Object o1;`<br>} | `class C' {`<br>`int i3;`<br>`int i1;`<br>`int i2;`<br>`Object o3;`<br>`Object o1;`<br>`Object o2;`<br>} |
| 32 bytes | 40 bytes | 24 bytes | 32 bytes |

Table 2: The initial and the three redefined versions of class `C`

**Increase:** An object field is added to the class resulting in an increased instance size of 40 bytes (because the size of an object is rounded up to 8 bytes). Therefore, the changed objects need 25% more heap area.

**Decrease:** Two object fields are removed resulting in a decreased instance size of 24 bytes. Therefore, the changed objects need 25% less heap area.

**Reorder:** All fields of the object are reordered to be in a different position than before. The size of the heap area of updated objects remains unchanged.

We create a total of 4,000,000 objects, resulting in an approximate size in memory of 128 MByte. For our benchmarks, we create fractions between 0% and 100% of the objects as instances of the redefined class. The rest of the objects are created as instances of another class with the same fields, but this class is kept unmodified. For comparison, we also provide the time for a full garbage collection without code evolution. As there are no dead objects on the heap, this garbage collection run only marks all objects, but does not need to copy memory in the compaction phase. We execute each configuration 10 times and report the mean of the runs. Figure 12 shows the results.

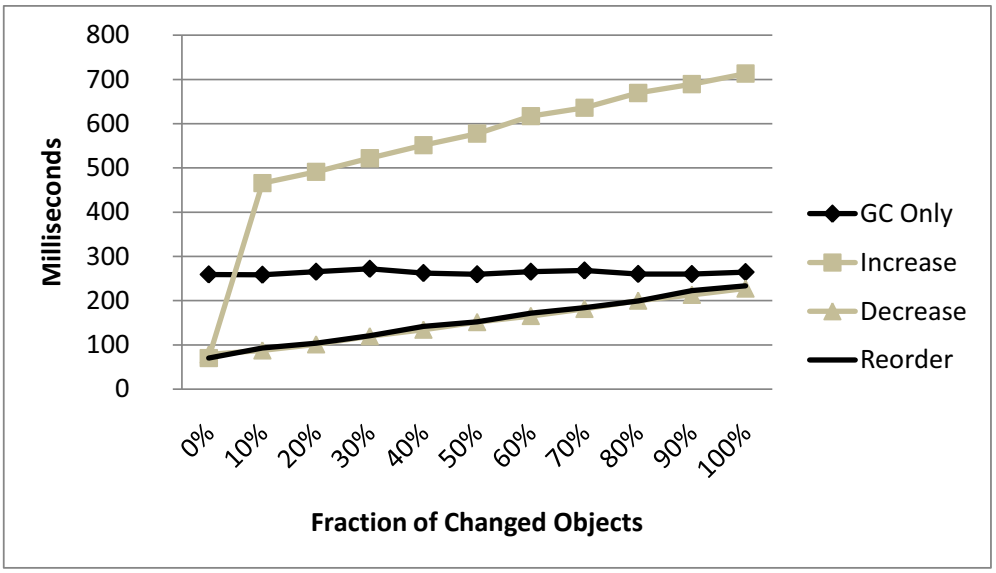


Figure 12: Micro benchmark results for changing the fields of a class compared against a garbage collection run.

When no objects are affected, the cost of a class redefinition is about a third of the cost of a full garbage collection run. Most of the cost comes from the heap iteration to update pointers and check for instances of redefined classes. The time also includes issuing the JDWP command and loading the new classes.

Increasing the size of all objects on the heap needs about three times more time than the no-load garbage collection run. About 20% of the objects need to be copied to a rescue buffer, because one object (32 bytes) makes room for four objects to increase their size by 8 bytes each. The performance of this benchmark is improved when there are dead objects at the beginning of the heap. The dead objects provide space for instances to increase their size and lead to less objects being copied to the rescue buffer.

Reordering the fields of all objects (and therefore copying the objects field by field instead of as a whole) is comparable to a full garbage collection run if all objects are affected. As the size of the objects does not change for this micro benchmark, the objects are updated in-place during the heap iteration and so the modified garbage collection run is not necessary. We need to make a copy of each object and then copy the field values one-by-one. Note that the VM really performs a reordering of fields in this example, although it would be allowed to keep the old order (because the VM is allowed to keep fields in any order). We could detect that this case is a pure reordering of fields and optimize it, but this would only benefit micro benchmarks and not real-world update scenarios.

Redefining a class such that all object sizes decrease is also slightly faster than a full garbage collection run. Decreasing the sizes can be done during the heap iteration using filler objects to fill the deprecated heap areas (see Section 4.6). This is significantly faster than adjusting the object sizes in a full garbage collection run like in the case where object sizes are increased. The filler objects are automatically removed by the garbage collector in subsequent garbage collection runs, because there are no pointers to them.

Compared to the performance numbers of an earlier VM implementation [55], we can show significant improvements on the micro benchmarks. The speed up for the 0% case where no heap object instances are

affected is more than a factor of 5. The benchmark scenarios that reorder fields or decrease the number of fields are improved by a factor of 3 to 4.

## 9. Related Work

### *9.1. General Discussions*

Several classifications of run-time changes of programs have been published [44, 24]. Ketfi et al. outline the adaptation of component-based applications [30]. Ebraert et al. present a general discussions on the problems and pitfalls of software evolution [15] and examine dynamic run-time changes from the point of view of the user [14].

Kramer and Magee investigate the problem of how to design applications to allow a consistent state transfer between new and old programs [32, 33]. Vandewoude et al. extend their work and introduce the notion of tranquillity [51]. Run-time evolution in the context of aspect-oriented programming is discussed by Gustavsson et al. [25].

### *9.2. Procedural Languages*

Early work on dynamic code evolution was done by Fabry [16]. The dynamic code aspect is achieved by jump indirection to procedures. Data conversion routines can be specified and the old and new version of a module can execute in parallel. Lee and Cook implemented a dynamic modification system for the StarMod language, which is an extension of Modula-2. Their system is called DYMOS [9]. It includes a command interpreter that can perform update actions based on certain conditions, e.g., when certain procedures are inactive.

Frieder and Segal developed a procedural dynamic updating system called PODUS [17, 46]. They require a binding table for methods that is updated accordingly. The concept of semantic dependency between methods is introduced.

Gupta implemented a hot code replacement mechanism for C programs on a Sun workstation based on state transfer [22, 21]. The system supports adding and deleting of functions. For adding global data, extra pointers must be declared in advance. Interprocedures are installed in case the return value or parameters of a method change. Those interprocedures map between a call from a method of the old program version to a new method of the new version. Gupta also developed a formal framework for program evolution [23]. Different formalizations of dynamic software updates were published by Bierman et al. [4] and Stoyle et al. [48].

Hicks et al. present a dynamic modification system for C-like languages [26]. They apply patches to the running program that are mostly automatically generated and can contain verification code. The patches contain the new code as well as the code needed to do the state transformation from the old to the new program. Neamtiu et al. extend this work to support the update of procedures with long-running loops and the ability to transform data of local variables [36].

In contrast to the work described in this subsection, our algorithm targets the challenges of code evolution for object-oriented languages. Also, we can leverage the advantages of dynamic compilation in a virtual machine and do not need to insert hooks into the statically compiled program. The modification of the garbage collector gives our algorithm maximum flexibility when performing instance updates.

### *9.3. Object-Oriented Languages*

Class evolution in object-oriented systems is discussed by Casais [6]. Clamen published about type evolution and instance adaptation in the context of object-oriented databases [8]. They dynamically connect system components. Bialek et al. developed a framework for the evolution of distributed component-based applications [3].

Hjalmtysson et al. [27] present an approach for dynamic classes in C++. They use the C++ template mechanism and proxy classes to realize the dynamic code evolution aspect. There has to be an interface definition for every dynamic class and it is only possible to change the implementation behind this interface.

Therefore, it is not possible to add or remove any public members of a class and it is also not possible to change the class hierarchy.

Duggan et al. suggest to solve the problem of coexisting old and new version objects by introducing the concept of equal types [13]. The old and new version objects can be used interchangeably. Conversion functions are used whenever the object is not of the required version.

The Common Lisp Object System (CLOS) [47, 18] includes the possibility to redefine a class. They allow the definition of transformer methods that update the instances from the old to the new version. A conceptual difference to our approach is that in CLOS the classes can only be redefined one-by-one, while we atomically redefine a set of classes. In the former case, the programmer is responsible for performing the redefinitions in the correct order. Additionally, our approach can deal with static typing and also with methods being defined as class members. This is not necessary for a class redefinition algorithm for the Lisp language. The class redefinition command for Java is designed for debugger use, while the method for redefining a class in Lisp can be called from user code. We believe that parts of our algorithm (e.g., the garbage collection modifications) can be used for efficiently implementing the CLOS class redefinition mechanism.

### *9.4. Java*

There are various approaches of dynamic code evolution for Java based on proxy objects and bytecode rewriting [41, 45, 43]. The main advantage of this approach is that it requires no change of the runtime system and can therefore be applied to any Java VM. Disadvantages are the performance penalty introduced by the indirection and the limitations of flexibility, e.g., changes of the class hierarchy are not supported. Additionally, support for triggering the code evolution from within development environments is not available or requires special plugins. Furthermore, the reflection facilities of the VM are affected (e.g., stack traces are obscure because of generated proxy methods). Gregersen et al. [20] advocate the idea of having a dynamic-update-enabled virtual machine and outline its advantages.

The project JDrums [1] is an implementation of a dynamic Java VM based on JDK 1.2. Its main limitations are that the just-in-time compiler must be disabled, active methods cannot be updated, and superclasses cannot be changed.

Malabarba et al. [35] present an implementation of code evolution based on JDK 1.2. They require that only the interpreter is used and cannot handle code evolution in the context of just-in-time compilation. In case of instance changes, they perform a global update using a mark-and-sweep algorithm during which all old version objects are converted to new version objects. In contrast to our solution, their modifications to version 1.2 of the JDK impose a significant performance penalty on normal program execution. Additionally, they allow only binary compatible changes and their VM uses object handles instead of direct object references.

Subramanian et al. [49] implemented code evolution for the Jikes RVM. They support adding and removing methods and fields, but do not support changes to the class hierarchy like we do. A special tool is used to generate update specification files. A transformation method is executed every time an object is converted between two versions. In contrast to our algorithm, their implementation is not focused on code evolution for debugging and can therefore neither perform an update at an arbitrary point nor be used from within standard Java development environments.

Dimitriev et al. [12] present a class evolution algorithm for the persistent object system for the Java programming language called PJama [2]. They introduce transformer methods for updating the stored objects. While some principles of class evolution also apply when updating the schema of an object-oriented data store, our main contribution is to perform dynamic class evolution and update heap objects while the user application is running.

The work most closely related to ours was done in attempt to apply PJama principles to run-time evolution of Java applications by Dmitriev [10, 11]. His implementation is part of the Java HotSpot$^{TM}$ VM since JDK 1.4. While instance and schema changes are discussed in the thesis, the actual implementation is only capable of swapping method bodies. It does change the original class metadata object and uses a constant pool merging algorithm to make sure that the new and the old methods can work with the constant

pool associated with the old class. Our approach of using a side universe (see Section 4.4) requires less code and is the key design decision that enables us doing more complex changes to classes. Also, we do not need a custom class loader for loading the new class versions for validating the changes. Our code is based on the current implementation of swapping method bodies in the Java HotSpot™ VM and significantly enhances it to allow arbitrary changes, including changes to the instance format and the class hierarchy.

## 10. Future Work

The current system always continues executing an old method until it returns to its caller. We want to reduce the time spent executing the old code. In some cases a method could even be never updated, e.g., because it contains an endless loop. Therefore, we want to explore the problem of replacing the current activation of an old method by an activation of a new method. While heuristics can help in simple cases, the general case cannot be solved, e.g., when a sorting method changes its algorithm from selection sort to quick sort. We want to provide a mechanism for specifying method forward points that are hints which instructions of the old method match instructions of the new method. If such an instruction is executed in the old method, the interpreter could change its current activation to continue running in the new method.

## 11. Conclusions

We presented a technique for dynamic code evolution in Java and described its implementation for the Java HotSpot™ VM. We allow arbitrary modifications to Java classes, including changes to the class hierarchy. The update can be performed at any point during program execution, and our VM works with standard Java development environments. Old and new code may co-exist in the VM, and therefore our approach allows redefining methods that are currently active. We discussed the problems introduced by binary incompatible changes. We present solutions to binary incompatible changes for deleted methods and static fields as well as for complex changes to the class hierarchy. This solution can be used to prevent class redefinition changes that can cause runtime exceptions in continued execution.

We showed that a production-quality VM can provide code evolution without losing performance during normal program execution. Our algorithm needs no additional memory indirections and works with the interpreter as well as with both just-in-time compilers of the Java HotSpot™ VM. The code evolution step is combined with a modified garbage collection run and has similar performance characteristics. The focus of our VM modification lies on improving debugging productivity, but it is also safe for updating long-running applications as long as the application developer is careful about handling the semantics of updated applications.

Our approach is integrated in the latest development version of a high-performance VM. Only few modifications were necessary, and they were limited to specific components although the VM was not designed with code evolution in mind. The implementation is part of a larger effort to extend the Java HotSpot™ VM with first-class support for languages other than Java (especially dynamic languages), called the *Da Vinci Machine Project* or *Multi-Language Virtual Machine Project* (MLVM) [37]. The source code and binaries of our modified virtual machine are available from http://ssw.jku.at/dcevm.

## Acknowledgements

We would like to thank current and former members of the Java HotSpot™ VM team at Oracle, especially Thomas Rodriguez, John Rose, David Cox, and Kenneth Russell, for their persistent support, for contributing many ideas and for helpful comments on all parts of the Java HotSpot™ VM.

## References

[1] J. Andersson, T. Ritzau, Dynamic code update in JDrums, in: Workshop on Software Engineering for Wearable and Pervasive Computing, 2000.

[2] M. Atkinson, M. Jordan, L. Daynès, S. Spence, Design issues for persistent Java: a type-safe, object-oriented, orthogonally persistent system, in: Proceedings of the ECOOP Workshop on Object-Oriented Databases, 1996.
[3] R. Bialek, E. Jul, A framework for evolutionary, dynamically updatable, component-based systems, in: Proceedings of the International Conference on Distributed Computing Systems Workshops, IEEE Computer Society, 2004, pp. 326–331.
[4] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, Formalizing dynamic software updating, in: Proceedings of the International Workshop on Unanticipated Software Evolution, 2003.
[5] S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 2006, pp. 169–190.
[6] E. Casais, Managing class evolution in object-oriented systems, Object-Oriented Software Composition (1995) 201–244.
[7] S. Chiba, Y. Sato, M. Tatsubori, Using HotSwap for implementing dynamic AOP systems, in: Proceedings of the Workshop on Advancing the State-of-the-Art in Run-time Inspection, 2003.
[8] S.M. Clamen, Type Evolution and Instance Adaptation, Technical Report CMU-CS-92–133, Carnegie Mellon University, 1992.
[9] R.P. Cook, I. Lee, Dymos: a dynamic modification system, in: Proceedings of the Symposium on High-Level Debugging, ACM Press, 1983, pp. 201–202.
[10] M. Dmitriev, Safe Class and Data Evolution in Large and Long-Lived Java Applications, Ph.D. thesis, University of Glasgow, 2001.
[11] M. Dmitriev, Towards flexible and safe technology for runtime evolution of Java language applications, in: Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, 2001.
[12] M. Dmitriev, M. Atkinson, Evolutionary data conversion in the PJama persistent language, in: Proceedings of the Workshop on Object-Oriented Technology, Springer-Verlag, 1999, pp. 25–36.
[13] D. Duggan, Type-based hot swapping of running modules, in: Proceedings of the International Conference on Functional Programming, ACM Press, 2001, pp. 62–73.
[14] P. Ebraert, T. D'Hondt, Y. Vandewoude, Y. Berbers, User-centric dynamic evolution, in: Proceedings of the International ERCIM Workshop on Software Evolution, 2006.
[15] P. Ebraert, Y. Vandewoude, T. D'Hondt, Y. Berbers, Pitfalls in unanticipated dynamic software evolution, in: Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2005, pp. 41–49.
[16] R.S. Fabry, How to design a system in which modules can be changed on the fly, in: Proceedings of the International Conference on Software Engineering, IEEE Computer Society, 1976, pp. 470–476.
[17] O. Frieder, M.E. Segal, On dynamically updating a computer program: From concept to prototype, Journal of Systems and Software 14 (1991) 111–128.
[18] R.P. Gabriel, J.L. White, D.G. Bobrow, CLOS: integrating object-oriented and functional programming, Communications of the ACM 34 (1991) 29–38.
[19] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java™ Language Specification, Addison-Wesley, 3rd edition, 2005.
[20] A.R. Gregersen, D. Simon, B.N. Jørgensen, Towards a dynamic-update-enabled JVM, in: Proceedings of the Workshop on AOP and Meta-Data for Software Evolution, ACM Press, 2009, pp. 1–7.
[21] D. Gupta, P. Jalote, Increasing system availability through on-line software version change, in: Proceedings of the International Conference on Fault-Tolerant Computing, IEEE Computer Society, 1993, pp. 30–35.
[22] D. Gupta, P. Jalote, On-line software version change using state transfer between processes, Software - Practice and Experience 23 (1993) 949–964.
[23] D. Gupta, P. Jalote, G. Barua, A formal framework for on-line software version change, IEEE Transactions on Software Engineering 22 (1996) 120–131.
[24] J. Gustavsson, A classification of unanticipated runtime software changes in Java, in: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, 2003.
[25] J. Gustavsson, T. Staijen, U. Assmann, Runtime evolution as an aspect, in: Proceedings of the International Workshop on Unanticipated Software Evolution, Springer-Verlag, 2004.
[26] M. Hicks, S. Nettles, Dynamic software updating, ACM Transactions on Programming Languages and Systems 27 (2005) 1049–1096.
[27] G. Hjlmtysson, R. Gray, Dynamic C++ classes – a lightweight mechanism to update code in a running program, in: Proceedings of the USENIX Technical Conference, 1998, pp. 65–76.
[28] U. Hölzle, C. Chambers, D. Ungar, Debugging optimized code with dynamic deoptimization, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1992, pp. 32–43.
[29] R. Jones, R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, 1996.
[30] A. Ketfi, N. Belkhatir, P.Y. Cunin, Adapting applications on the fly, in: Proceedings of the IEEE International Conference on Automated Software Engineering, IEEE Computer Society, 2002, p. 313.
[31] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, D. Cox, Design of the Java HotSpot™ client compiler for Java 6, ACM Transactions on Architecture and Code Optimization 5 (2008).
[32] J. Kramer, J. Magee, The evolving philosophers problem: Dynamic change management, IEEE Transactions on Software Engineering 16 (1990) 1293–1306.
[33] J. Kramer, J. Magee, Analysing dynamic change in software architectures: a case study, in: Proceedings of the International Conference on Configurable Distributed Systems, IEEE Computer Society, 1998, pp. 91–100.

[34] T. Lindholm, F. Yellin, The Java™ Virtual Machine Specification, Addison-Wesley, 2nd edition, 1999.
[35] S. Malabarba, R. Pandey, J. Gragg, E. Barr, J.F. Barnes, Runtime support for type-safe dynamic Java classes, in: Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 2000, pp. 337–361.
[36] I. Neamtiu, M. Hicks, G. Stoyle, M. Oriol, Practical dynamic software updating for C, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 2006.
[37] Oracle Corporation, Da Vinci Machine Project, Oracle Corporation, 2011. http://openjdk.java.net/projects/mlvm/.
[38] Oracle Corporation, Java Debug Wire Protocol (JDWP), Oracle Corporation, 2011. http://download.oracle.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html.
[39] Oracle Corporation, OpenJDK, Oracle Corporation, 2011. http://openjdk.java.net/.
[40] Oracle Corporation, Top 25 RFEs (Requests for Enhancements), Oracle Corporation, 2011. http://bugs.sun.com/top25_rfes.do.
[41] A. Orso, A. Rao, M.J. Harrold, A technique for dynamic updating of Java software, in: Proceedings of the IEEE International Conference on Software Maintenance, 2002, pp. 649–658.
[42] M. Paleczny, C. Vick, C. Click, The Java HotSpot™ server compiler, in: Proceedings of the Java Virtual Machine Research and Technology Symposium, USENIX, 2001, pp. 1–12.
[43] M. Pukall, C. Kästner, G. Saake, Towards unanticipated runtime adaptation of Java applications, in: Proceedings of the Asia-Pacific Software Engineering Conference, IEEE Computer Society, 2008, pp. 85–92.
[44] M. Pukall, M. Kuhlemann, Characteristics of runtime program evolution, in: Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution, ACM Press, 2007.
[45] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 2002, pp. 205–230.
[46] M.E. Segal, O. Frieder, On-the-fly program modification: Systems for dynamic updating, IEEE Software 10 (1993) 53–65.
[47] G.L. Steele, Jr., Common LISP: the Language, Digital Press, second edition, 1990.
[48] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, I. Neamtiu, Mutatis mutandis: Safe and predictable dynamic software updating, ACM Transactions on Programming Languages and Systems 29 (2007).
[49] S. Subramanian, M. Hicks, K.S. McKinley, Dynamic software updates: a VM-centric approach, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 2009, pp. 1–12.
[50] D. Ungar, Generation scavenging: A non-disruptive high performance storage reclamation algorithm, in: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM Press, 1984, pp. 157–167.
[51] Y. Vandewoude, P. Ebraert, Y. Berbers, T. D'Hondt, An alternative to quiescence: Tranquility, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society, 2006, pp. 73–82.
[52] A. Villazón, W. Binder, D. Ansaloni, P. Moret, HotWave: creating adaptive tools with dynamic aspect-oriented programming in java, in: Proceedings of the International Conference on Generative Programming and Component Engineering, ACM Press, 2009, pp. 95–98.
[53] T. Würthinger, W. Binder, D. Ansaloni, P. Moret, H. Mössenböck, Applications of enhanced dynamic code evolution for Java in GUI development and dynamic aspect-oriented programming, in: Proceedings of the International Conference on Generative Programming and Component Engineering, ACM Press, 2010, pp. 123–126.
[54] T. Würthinger, W. Binder, D. Ansaloni, P. Moret, H. Mössenböck, Improving aspect-oriented programming with dynamic code evolution in an enhanced Java virtual machine, in: Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution, ACM Press, 2010, pp. 5:1–5:5.
[55] T. Würthinger, C. Wimmer, L. Stadler, Dynamic code evolution for Java, in: Proceedings of the International Conference on the Principles and Practice of Programming in Java, ACM Press, 2010, pp. 10–19.