# Recoverable Class Loaders for a Fast Restart of Java Applications

Vladimir Nikolov
Institute of Distributed Systems
University of Ulm
Germany
vladimir.nikolov@uni-ulm.de

Rüdiger Kapitza
Dept. of Comp. Sciences
Informatik 4
University of Erlangen-Nürnberg
Germany
rrkapitz@cs.fau.de

## ABSTRACT

This paper proposes recoverable class loaders to enable a fast start-up and recovery of Java applications. In contrast to traditional snapshot approaches that create full system images, our approach creates partial snapshots that contain a static part of the execution state of a Java application. It is the state of the class loaders and their associated class objects, which are recovered and used for restart.

This is especially useful in the context of mobile devices and mobile services. In the first case it allows to shutdown applications for power-management reasons as their restart takes less time, so power management does not disturb users. In the second case services can be much faster rebooted to cure software faults such as memory leaks. Thus, users will notice a substantially reduced downtime.

We implemented recoverable class loaders inside the JamVM and the OSGi middleware Oscar. For both cases of use — Java application restart and service recovery — we provide experimental evaluations that show a substantially reduced start-up time from up to 74%.

## 1. INTRODUCTION

Mobile devices such as cellular phones and handhelds face a multitude of requirements to be useful in every day life. They should support resource-intensive applications such as video streaming, web browsing and provide mobile office support. This demands for a powerful CPU, plenty of RAM and persistent memory, a bright display and a fast network connection. These requirements are contradicted by the limited available energy due to the small weight and form-factor these devices should have.

The consequence is the need for a rigorous power management. Current, mobile device hardware and software provide already special power saving features (e.g. CPU frequency scaling) to slow down or even stop all services that are not essentially needed. In the ideal case the user should not experience a change if power-management is applied or not. Taking into consideration this fact, the present work targets at the fast restart of complex Java applications.

On mobile devices Java is often used to provide extended applications such as a Web-browser, office applications or navigation support. These applications should be shut down if not needed by the user to enable a more efficient use of available power (e.g. to switch off unused memory units). However, a user is not willing to wait for long time if he or she needs to access such an application. Thus, support for a fast application start-up is necessary.

Another use case of a fast restart of Java applications is the quick system recovery of mobile services after software faults. All software including complex services in the context of mobile environments suffer from faults. If software reaches a mature state these faults are more complex and often harder to detect and consequently to fix. Thus, software faults will occur even in well-tested environments. One instant way to overcome such faults when they happen is to restart the affected system. Many problems such as memory leaks or race conditions can be temporarily cured in this way. However, restarts take time and services are not available in the meantime. Consequently, extended support for a fast system restart is desirable to provide highly available services. This is especially the case in mobile environments where services are usually instantly needed (e.g. to request an alternative route from an navigation system to drive round a traffic jam).

This paper proposes *recoverable* class loaders as a means to support the fast start-up of Java-based applications on mobile devices and a fast recovery of Java-based services in mobile environments. This is achieved by a tight integration of virtual machine support for snapshooting the runtime state of a Java application, and by middleware support for recoverable class loaders running on top of the virtual machine. Complex Java-based software in context of embedded and mobile devices is often implemented using the OSGi [12] component middleware. OSGi enables the fine-grained update and exchange of components regarding functionality and code thereby making heavy use of Java class loaders. Originally, OSGi targeted the provision of services on top of gateway and wireless routers. Meanwhile, the scope of OSGi has widened and it is applied to structure and modularise numerous complex applications such as IDEs (e.g. the Eclipse project) on desktop computers or for infotainment software in mobile vehicles.


MOBILWARE 2008, February 13-15, Innsbruck, Austria

DOI 10.4108/ICST.MOBILWARE2008.2831

In a first step we investigated the start-up behaviour of OSGi based applications and encountered long times until such applications are up and operational. A common approach to solve such issues is the use of snapshots that capture the execution state of an application which is stored on disk and put back into memory on demand. However, this approach is not really suitable as it captures the whole application execution state including antecedent faults (e.g. memory leaks). Thus, full-fledged snapshots are not useful for a fast recovery to cure software faults. In our initial investigations we traced system start-up and detected the loading and initialisation of classes as one of the major bottlenecks. In fact it consumed up to ∼25% of the start-up time of a demo application consisting of 30 OSGi bundles, not including the time to load the Java system classes. A bundle is building the deployment unit of OSGi. As consequence we modified a virtual machine to capture the loaded and linked class objects and provided an integration of this feature into the OSGi framework to enable the recovering of class loaders. This reduces the start-up time in the best case by factor of ∼5. Unlike traditional snapshot techniques, not the whole application state is recovered and used for a restart but only a static part of the application execution state that is not subject to changes. Consequently, problems like memory leaks and inconsistent states that usually take effect after a long runtime are not an issue. Thus, the presented approach is not only suitable to speed-up the restart of an application but also to support the fast recovery from software faults.

The paper is structured as follows: First, we outline the basic concept of recoverable class loaders. Then we give a brief introduction of the JamVM that builds the foundation of our prototype implementation. In Section 4 we specify our prototype. Afterwards in Section 5, we present an experimental evaluation. First, we investigate a scenario where the start of an application should be improved. Second, we give an example of fast service recovery. Finally, we specify related approaches and draw conclusions.

## 2. BASIC IDEA OF RECOVERABLE CLASS LOADERS

Figure 1 explains the basic priniciple of our recoverable class loader approach. The top part shows a simplified internal memory layout of a Java virtual machine (JVM). The Java-heap region stores objects and class blocks; it is the part of the memory that is subject to garbage collection, hence it is also called *GC-heap*. The object blocks contain the private data of Java object instances. Each object block is linked with a certain class block, which is the central definition unit of the Java class it instantiates. Class blocks in turn denote structures which store administrative data of Java classes; this is, for example, the class name, a class signature, access flags, class state and also references to various method-area blocks. The method-area data usually do not reside on the GC-heap; each JVM manages a common method-area region where e.g. the constant pools, the methods-, fields- and interface-blocks, and also bytecodes of the defined and linked Java classes are settled. In order to enable fast application reboots, we investigated a facility to preserve a variety of already initialised static information on the Java-heap across system shutdown and consequently to reuse it on demand during recent JVM restarts. As already mentioned, we do not aim to recover the complete application state; that would be the total quantity of objects, class blocks, method-area and runtime data residing in JMV's memory. In contrast, only a desired set of the already existing class loader objects and respectively their loaded system and application classes shall be reused. Thus, we allow reconfigurations during fast application reboots which, however, are still able to remedy software faults and inconsistent states.

Generally, in Java a class loader is the instance of providing the necessary interfaces to the class loading routines of the underlying Java virtual machine to any user application running on the top, thus allowing dynamic extension of the application's code and semantics. In this regard class loaders are central units for organising already loaded Java classes, their code and meta data. They also provide code isolation between application components, for example in a J2EE [11] server environment and similarly in the OSGi middleware layer. From a programmatic point of view, class loaders are `java.lang.ClassLoader` object instances residing on the garbage collected JVM heap besides various additional objects and class blocks.

One straightforward solution to recover existing class loader objects and the entire pool of their loaded classes information within an initialised JVM is to create a persistent image of the whole GC-heap area, as indicated in the bottom part of Figure 1. We denote this procedure as heap-snapshooting. Consequently, all already initialised class blocks and objects (including the Java class loaders) are persistently saved across JVM restarts. During recent reboots, the pre-initialised GC-heap snapshot can be mapped directly into the JVM process memory. Hence, class loader objects and class definitions required by the JVM and a Java application can be reused instantly during each bootstrap process. This procedure is also depicted in Figure 1 and further referred to as snapshot-injection. At this point we assume that all memory-mapped information on the heap, which is not further needed, is discarded during the first garbage collection after a system restart.

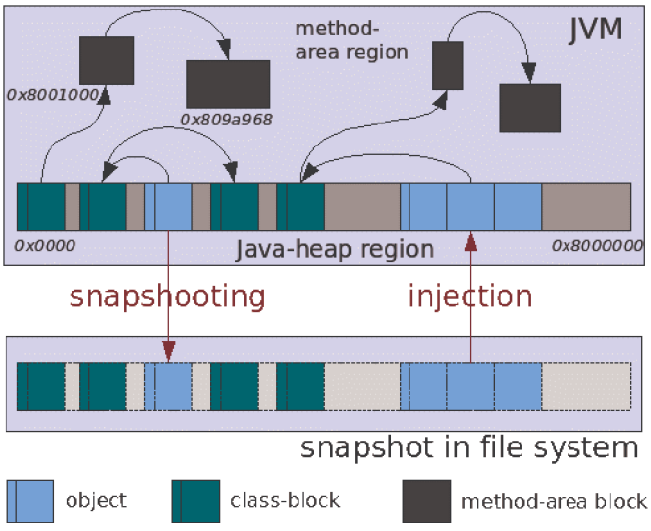


**Figure 1: A simple JVM memory layout**

A typical use case for this technique could be for exam-

ple a checkpoint during program execution, at which the snapshooting of the actual JVM-internal heap state is initiated by a privileged user or even by a Java application itself. On later JVM invocations the automatically generated persistent heap image can be reused by specifying a snapshot-injection option. In this paper we essentially want to demonstrate, how after short re-initialisation of an embedded heap snapshot all class loading procedures within a rebooting JVM and Java application can be completely skipped. Of course, this is feasible until program execution reaches again the checkpoint state where the heap snapshot was previously taken. Instead of recently parsing system and application `class`-files and thereby generating appropriate internal class blocks on each start-up, the already initialised data which is provided by the embedded heap image shall be reused. In this manner, the instruction path, which a JVM has to work off during its bootstrap process, is shortened extensively. In consequence, the start-up procedure of the JVM and also of the hosted Java application is accelerated enormously. Thus, the usability and availability of a Java system can be improved, as well as the latency until user-interaction is feasible during a system reboot is potentially decreased.

The Java Virtual Machine Specification [8] is rather general and offers little insight about how the memory management as well as the structuring of the loaded classes information and meta data could be realised in fact. Into how many regions the heap of a JVM is divided, and in which manner a garbage collection mechanism operates on them is implementation-dependent. In addition, the way how class loaders are organising and referencing already loaded classes is not standardised. For that reason we had to focus on one certain JVM implementation in our case the JamVM [9] .

## 3. JAMVM

The JamVM is a relatively new, open-source JVM that is extremely small (150∼200 Kbyte), but implements the complete Java Virtual Machine Specification Version 2. Thus, it supports user-defined class loading, object finalisation and class unloading. As standard Java Class Library the JamVM comes with the prevalent and likewise open-source GNU Classpath, and additionally offers dynamic loading and execution of native code through its own low-overhead JNI (Java Native Interface) implementation. The JamVM operates exclusively in interpreter mode (direct/indirect) and has no native- or JIT-compiler, which substantially facilitates our extensions, as well as the measurements of the actual performance gain obtained thereby. By default the JamVM manages internally one global Java-heap area, where all objects and loaded classes structures are organised and scanned by a mark-sweep garbage collector. A similar memory management outline as that implemented by the JamVM is also briefly depicted by the top part of Figure 1. If free memory dissipation heavily accrues because of fragmentation, the heap area is compacted as well by another mark-compact collector, which relies on the efficient Jonkers compaction algorithm. Since JamVM implements a kernel-based threading model on the top of the POSIX threads OS interface, it is already ported for miscellaneous POSIX-aware platforms and tested with various processors like PowerPC/G3/G4/64, AMD/64, MIPS, ARM and iPAQ, with Linux and MAC OS X. Thus, the JamVM addresses desktop and mobile devices. Summarising, the extreme compactness and full JVM V2 compatibility were the decisive points for the choice of this JVM for our implementations and measurements.

## 4. PROVIDING RECOVERABLE CLASS LOADERS AND JAVA CLASSES

In order to automatically provide a persistent image of the garbage collected heap allocated within the JVM process, we utilised the common *memory-mapped* [6] services in Linux. Those services are supported nowadays by the most POSIX compliant operating systems. Therewith, we succeeded to redirect entirely all memory operations within a pre-reserved memory-mapped heap area of a modified JamVM to the local file system. Thus, an outright persistent image of the Java-heap can be easily generated.

For this purpose the virtual memory manager of the operating system is requested by means of the `mmap()` system call, to mirror all write operations 1:1 in a background file. When objects and class structures are settled and initialised on the heap, they are automatically replicated within the same address-offsets into the background file. In the normal case, this replication is not adherent to any performance overhead during runtime, since we implemented our JamVM prototype to operate on a private copy of the memory-mapped file in its process memory. Thus, all write operations are cached until synchronisation is forced. Theoretically, the synchronisation between process memory and the background file can be initiated automatically by the operating system, e.g. when the total amount of virtual memory in the system runs low. In fact, the affected virtual memory pages due to modifications are written back to the file system, when the Java-heap area is unmapped or a synchronisation is explicitelly forced via the `msync()` system call. The latter is used for example on snapshot generation, where afterwards the up-to-date background file can be simply copied under a certain snapshot name in the file system.

Similarly, during a restart of our modified JamVM version an already existing snapshot file can be easily hooked into the process as a pre-initialised heap area likewise using `mmap()`. On read accesses within the mapped area, the memory manager is in turn responsible for swapping the affected partitions page by page from the background file into the process. We meassured the swapping overhead introduced on read operations during runtime. Since an accessed page is read only once from disk the effective costs for these opreations are negligible.

JamVM does not use object handles to reference objects on the garbage-collected heap. Instead, it uses direct pointers to the object's physical address in memory, a strategy which improves the general runtime performance. Because of that, we are forced to guarantee, that objects can be found after restart on the expected and respectively referred positions in process memory, even in an embedded heap snapshot. A simple possibility to fulfil this requirement is offered likewise by the memory-mapped services. It is additionally possible via `mmap()` to map a snapshot file again at a certain fixed address in the process memory.

In addition it should be mentioned, that since each JVM process operates on a private read-only copy, theoretically a persistent snapshot can be shared concurrently between several JamVM instances. However, this scenario is not targeted by our approach because multiple snapshot replication would very likely exhaust rapidly the system's virtual mem-

ory and lower drastically the overall system performance.

## 4.1 Memory-Management Extensions

Class blocks and object instances are usually placed on the garbage-collected heap area, so that their validity can be automatically managed by the JVM and especially the garbage collector. As already explained, they can be reused across system restarts by injecting a pre-initialised image of the Java-heap (a snapshot) into the private memory of a newly starting JVM-instance. Though, on system shutdown all method-area data referenced by the class blocks are irrecoverably lost. The reason for it is, that only the information on the memory-mapped GC-heap is captured just at that moment within a heap snapshot.

In the JamVM the entire memory for method-area blocks is allocated via usual `malloc()` on the machine's process-heap, beyond the garbage-collected heap area. In Figure 1 we have indicated accordingly a method-area region in the JVM's process memory, wherein method-area blocks are placed arbitrarily. Furthermore, during the process of initialisation and class-linking the JamVM generates a multitude of further machine-level metadata e.g. methods, interface and exception tables used for faster method invocations, as well as various internal runtime data like administrative hashtables, entries for loaded DLLs, Thread entries, and UTF-8 strings. All those constructs' memory is similarly reserved within the JVM's address-space via `malloc()` and likewise lost in the normal case on system shutdown.

However, in order to be able to reuse a snapshot after a system restart, the entire method-area data blocks, all UTF8-strings and several internal system-hashtables must be forthcoming again at their former positions in the JVM address space, as they are still referenced and in use by the recovered class blocks, objects and the runtime system. Even all DLL entries must be stored persistently across shutdown and recovered afterwards, as we will explain later. Fortunatelly, the JVM specification indicates no guidelines where and in which way method-area and runtime data structures are to be internally organised by a JVM. In order to overcome this issue we identified two different approaches.

### 4.1.1 All-In-GC-Heap Approach

The first approach arranges all required method-area and runtime data likewise on the GC-heap of the JamVM besides class definitions and object instances, as depicted in Figure 2. Thus, all necessary data structures for the reuse of class loader objects and their class definitions after a system restart are entirely contained in one and the same heap-snapshot. In this manner, we further refer to that approach as *All-In-GC-Heap (AIGH)*. In order to attain a particular treatment of accessory non-object data on the GC-heap and also to protect it against the garbage collector and further heap-scans, we comprised appropriate markings in all the concerned memory chunks. With the help of special "non-object" flags in the chunk-headers, we are incessantly enabled to differ between objects, class blocks and the pure non-object memory-allocations on the heap, and to perform different operations on the latter. In that manner we also adapted and extended accordingly the allocation- and garbage-collector-routines of the JamVM. However, measurements and benchmarks of our JamVM-AIGH prototype pinpointed, that the additional complexity which we introduced for the garbage collector enormously extends the durations of the collections and decreases the overall runtime performance of the JVM. Similarly the total number of the GC runs has doubled, since now nearly twice as much data resides on the garbage collected heap of the machine. Despite the faster start-up achieved by the snapshooting and the reuse of class definitions, our JamVM-AIGH prototype is inconceivable for application under real circumstances.

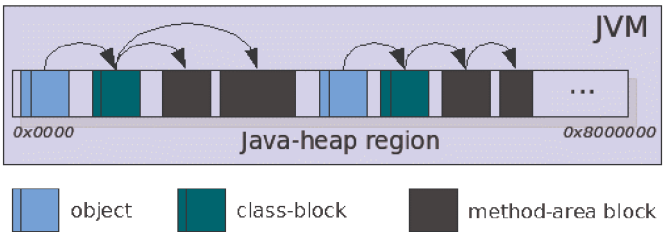


**Figure 2: JamVM memory layout with AIGH**

### 4.1.2 Method-Area-Heap Approach

Within the scope of our second solution we extended the JamVM with an additional managed memory-mapped heap, the *Method-Area-Heap (MAH)*, as depicted in Figure 3. We equipped that area with own allocation-routines, as well as with an own memory management. Method-area data and required runtime-structures like UTF-8 strings, hashtables and DLL entries can be positioned now via `mallocMAH()` on a separate cross-restart cache, which also can be mapped directly into memory and likewise reused after a restart of the JVM. In this way the garbage-collected heap is not overloaded and the number of the collections does not increase. Furthermore, no own garbage-collection mechanism is necessary for the MAH at all, since the structures settled therein can be released directly by our `freeMAH()` function. In order to countervail the emerging fragmentation of the MAH, our `freeMAH()` function was implemented in a way, that adjacent free memory chunks are merged together into bigger free blocks. Our measurements prove that the overall runtime performance of our JamVM-MAH version is not considerably lower than the runtime performance of the unmodified JamVM. The sole marginal performance-leaks we were able to determine pertain to allocations of runtime and method-area data, since those are reserved and released now not by means of `malloc()` and `free()`, but with the less efficient versions of the MAH memory-management functions. This performance loss is however negligible in relation to the start-up acceleration won by the applied JVM snapshooting.

## 4.2 Recoverable Class Loaders Extension

Class loader objects have to organise in some way a reference to each already loaded and defined Java class. In order to allow quick class-queries during the class loading and instantiation process, the JamVM, as well as many other JVMs, makes use of hashtables. Each class loader encapsulates in its instance-data a reference to a certain hashtable object, and that hashtable in turn manages references to the definition structures of all Java types, which were already loaded and defined by the class loader. Since in the JamVM class-blocks and object instances (including class loaders) reside by default on the GC-heap, we only bypassed the default hashtables generation on the process heap via `malloc()` and repositioned them in a similar way on a snapshotable heap

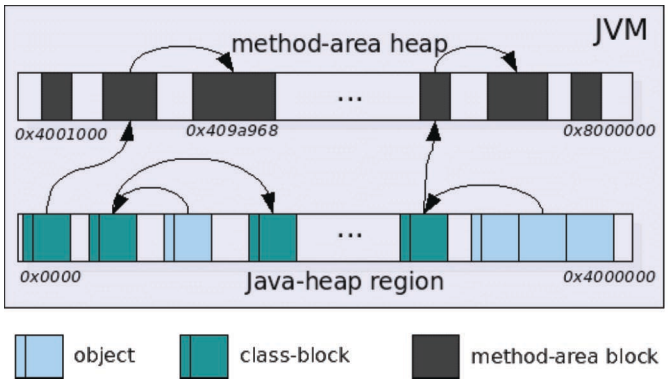


**Figure 3: JamVM memory layout with MAH**

area — that is either the GC-heap (with JamVM-AIGH), or the extended MA-heap (with JamVM-MAH). From that point on all class loader objects and their loaded classes information are entirely reconstructable from a persistent snapshot. In order to facilitate the class loader extension for user applications, we realised the necessity for a technique to uniquely identify and restore a certain class loader instance within the heap snapshot during an application restart. An application needs to be able to register newly instantiated class loader objects as "recoverable" in the JVM and thus to query and reuse them during its initialisation process on later restarts. For that purpose we extended the JamVM with an additional *Recoverable Class Loader Module (RC-module)*, which facilitates appropriate *registration*, *query* and *deregistration* functions for class loaders under certain *unique IDs*. We then bypassed the interfaces of that RC-module through the JamVM's internal JNI-tables over the GNU-Classpath up to the user-application level. We implemented an extended version of the `java.lang.ClassLoader` system class, which now provides new public (native) interfaces `registerClassLoader()` and `restoreClassLoader()` and offers therewith any arbitrary Java program the opportunity to access it's own stock of previously loaded classes, even after a reboot of the JVM. Class loader objects which in turn were not registered as "recoverable" are garbage collected by default together with all their loaded classes information and hashtables on the heaps.

## 4.3 Reinitialisation

After restarting the JVM with an embedded heap snapshot accomplishing some preparations and reinitialisations is mandatory, before the data contained therein can be reused. As previously mentioned the JamVM implements its own low-overhead JNI-interface. First of all, it should be considered that already invoked native methods of the Java classes residing on the reconstructed heap most likely refer to ranges in process memory, where after a JVM restart with a very large probability no valid native function-code is available. The reason for this is the dynamic linkage of native methods.

Usually the implementation code of native methods is contained in pre-compiled native libraries (DLLs) which are bound dynamically into the JVM's address-space. On load the therein contained native functions are resolved as native symbols in the process memory. The actual address-offsets of those symbols differ each time a library is bound into the JVM process. When the native method of a Java class is invoked, and it is not implemented directly in the JVM, the pertinent native symbol of the implementing function has to be resolved in any of the merged libraries; otherwise an `UnresolvedLinkError` exception is thrown by the JVM. On success the JamVM stores a direct reference to the located native function in the appropriate method block, in order to avoid further unnecessary recent resolutions. In that manner a new invocation of a native method, which has been already resolved during the last execution of the JVM, will cause offhand an unpredictable behaviour of the machine or at least a `segmentation fault`. Because of that certain precautions have to be taken while restarting the JVM with an embedded heap snapshot.

Initially, all previously loaded DLLs must be bound again into the JVM's address-space. For determining which DLLs exactly are to be reloaded, we store and respectively reconstruct the internal administrative structures of the DLL subsystem of the JamVM (the *DLL-Entries*) likewise from the embedded snapshot. Since the addresses of the native symbols will very improbably match again, all already defined Java classes contained in the heap snapshot must be scanned for native methods and any already existing native binding must be cancelled. Thus, on a subsequent invocation of a native method its native binding must be resolved again dynamically. In some cases where class level locks are hold during snapshot generation, deadlock problems are possible when that snapshot is reused later. For that reason, besides the existing JNI bindings we also have to cleanup the lock field of every encountered class object during snapshot initialisation.

Furthermore the static constructors (`<clinit>`) of certain Java classes are to be invoked over again. Therein many classes initialise static variables and parameters, or call additional native initialisation routines. We specified a set of classes experimentally, which are to be reinitialised after a restart. In that manner, during the reboot procedure we only have to traverse over the hashtables of all class loaders already registered in our RC-module and to invoke manually the static constructors of all detected classes that are matching in the predefined set. However, a sophisticated implementation could adapt this quantity dynamically at the actual needs of the executed Java program.

## 4.4 Start-up Scenarios

During the start-up of the JamVM we differentiate in principle between two runtime modes: an initial start in *"normal-mode"* and a restart with an embedded heap snapshot in the so-called *"heap-injection-mode"*.

In general, the initial start-up in normal-mode is intended for generating valid snapshots of the JamVM heap areas and likewise class loader objects can be registered as "recoverable" in the RC-Module. In normal-mode the JamVM starts up and initialises all its subsystems as usual. At the meantime all memory operations on the heap areas are mirrored in the memory-mapped background files, as already explained. The snapshots of the heap areas can be provided either by the Java program at runtime or automatically on shutdown of the machine. Thereby all operations on the heaps are frozen (stop-the-world) and after synchronisation with the memory-mapped background files copies of the heap areas are stored under specified names in the file system. We implemented this technology similarly for both JamVM prototypes (AIGH and MAH).

In order that snapshot files can be properly merged again as appropriate heap areas into the JamVM, certain information such as start address, current logical and physical dimension, or start-positions of free-lists etc. is necessary for each individual area. Additionally, the positions of certain runtime structures and data used by the individual JamVM subsystems must be indicated in the snapshot, so that these can be re-initialised correctly (from the snapshot). This information is collected automatically on snapshot generation and exported in a separate "injection-values" structure in the file system. During the restart in injection-mode the JamVM queries automatically for the injection-values file and initialises the appropriate snapshot files with the help of the values indicated there as heap areas in its process memory. In inject mode, subsystems like the Class, UTF-8 and DLL modules of the JamVM are not initialised from scratch, but with the already initialised runtime data and the administrative hashtables provided by the snapshot. This is also the case for the RC-module containing references to the class loader objects marked as "recoverable". After all necessary re-initialisations as described in Section 4.3 are done, the class loader objects needed for the further reboot procedure of the JVM and the Java application can be queried within the snapshot with the help of the RC-module and they can be directly reused together with all their loaded classes.

Currently, the presented prototypes do not support class updates while they are operating in injection mode. Consequently, if newer versions of the classes provided by a snapshot shall be applied, the complete snapshot must be regenerated. However, in case of OSGi, updates of private bundle classes are managed automatically by the OSGi Framework. Consequently, a bundle is provided with an accessory class loader, when its code has been altered. Thus, snapshot regeneration is essential only if middleware or system classes are out-of-date.

As a last point, RC-aware Java programs running on the top also need to be informed about in which mode the JamVM underneath currently operates. Thus, the Java program should adapt its behavior accordingly. In this manner a system like the Oscar OSGi for example should be able to decide whether certain bundle class loader are to be instantiated normally and probably then registered in the JamVM (in normal-mode), or whether in case of injection-mode those can be reconstructed on the basis of their bundle ID directly from the snapshot. For this purpose our modified JamVM implementation defines automatically a system variable "vm.heap.inject=yes" in injection-mode, which can be queried by a Java program. Thus, class loaders of all installed bundles can be reconstructed from the snapshot.

# 5. EXPERIMENTAL EVALUATION

To verify the claim of an improved start-up performance using recoverable class loaders two scenarios were evaluated. In a second experiment we proved the usability of recoverable class loaders for a fast recovery from software faults. Finally, we measured if the necessary modifications of the JamVM have an impact on the overall runtime performance. Throughout the experimental evaluation we used three different devices: a standard PC, a small embedded system and a typical handheld device (see Table 1 for hardware details). For all experiments we compared the standard JamVM 1.4.4 with our extended version supporting the MAH approach. The general performance drawbacks of the AIGH approach, which were already explained in Section 4.1.1, foreclose an effective application of the corresponding JamVM-AIGH prototype. Hence, in our measurements we focused on the JamVM-MAH implementation, since we consider this one as an improved successor.

## 5.1 Restart Example

For the first evaluation we conducted two separate experiments. First, we evaluated the start-up time of a very simple Java application, basically consisting of a `main()` function. Thus, only the system class loader was recovered and used for a fast restart. We measured the start-up time for the unmodified JamVM, the initial start-up time for the JamVM-MAH, taking a snapshot, and the start-up time for the modified version, using a snapshot gained from a previous run. The results are shown in Figure 4. Each value is an average value of 100 runs. The initial start-up of the modified JamVM takes slightly longer than the original JamVM when a snapshot is taken. If a snapshot is used and the class loader can be recovered, starts take significant less time. In fact the start-up could be decrease by 74% for the handheld device. The reason for the longer initial start-up time of the modified JamVM can be found in the slightly more complex `mallocMAH()` operation that is necessary to place elements on the separate heaps.

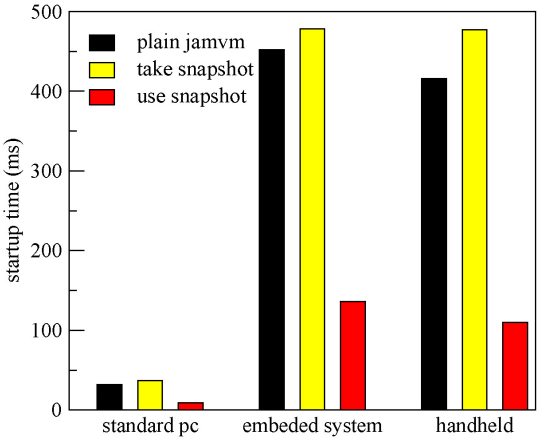


**Figure 4: Start-up time for a simple Java application**

In a second scenario, we measured the start-up of a complex OSGi configuration of 13 bundles taken from the Oscar Bundle Repository (OBR). A bundle resembles a component that is loaded via an own dedicated class loader. As a default all these class loaders are recovered. Again we measured the time for a start-up using the unmodified JamVM, the initial start of the JamVM-MAH, and a restart of the JamVM-MAH, using a previously taken snapshot. The results are depicted in Figure 5. In all case using a snapshot could significantly decrease the start-up time. Regarding both Figures 4 and 5 there is a major difference between the embedded system and the handheld device. While in Figure 4 both devices deliver equal start up times, in Figure 5 the handheld needs twice as long time to boot. This is due to the fact that the handheld operates with much fewer RAM than the embedded system. Thus, since the second scenario is more complex, the handheld has to swap virtual pages within its flash memory more frequently. This turned out to be a major bottleneck for the handheld device.

| Device | CPU | Memory | Disk |
|---|---|---|---|
| Standard PC | Pentium 4 2,40 GHz | 512 MB | 7200 RPM |
| Embedded System | Strong ARM 233 MHz | 256MB | 5400 RPM |
| Handheld (HP Jornada) | Strong ARM 200 MHz | 32 MB | Flash memory 2 GB (SanDisk Ultra II) |

**Table 1: Hardware used for experimental evaluation**

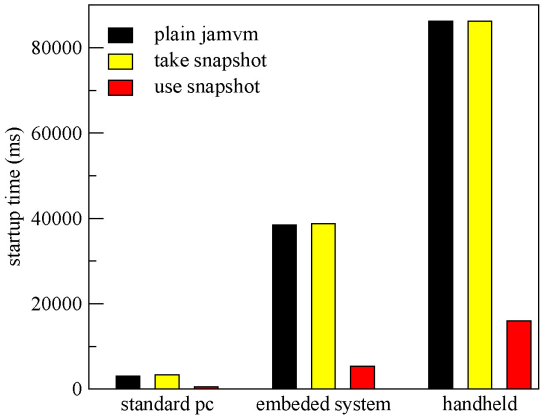


**Figure 5: Start-up time for a complex OSGi application**

## 5.2 Recovery Example

For the recovery example we chose a small OSGi configuration, covering the default bundles of Oscar, and two additional ones that provide a HTTP server and a small set of Web pages. Furthermore, we implemented a client program that connected the HTTP server and recorded the time to establish a successful connection. For the test we rebooted the service periodically and measured the time to successfully reconnect the service after a reboot. Figure 6 shows the average time of 100 reboots. Again using recoverable class loaders substantially reduces the start-up and consequently the time to recover the service.

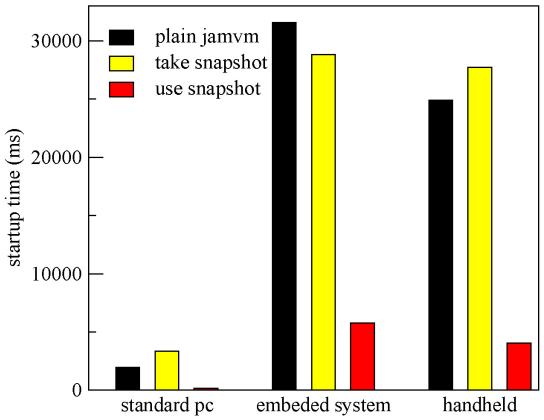


**Figure 6: Recovery time for a OSGi HTTP service**

## 5.3 Performance Evaluation

To evaluate if our prototype implementation, supporting recoverable class loaders, and the necessary modification of the JamVM have an impact on the runtime performance, we measured the original JamVM and our modified version with the SPEC JVM98 [3] benchmarks. The results are shown in Figure 7. All benchmarks provide similar values for the original and the modified JamVM. Thus, our approach has almost no impact on the runtime performance of the virtual machine.

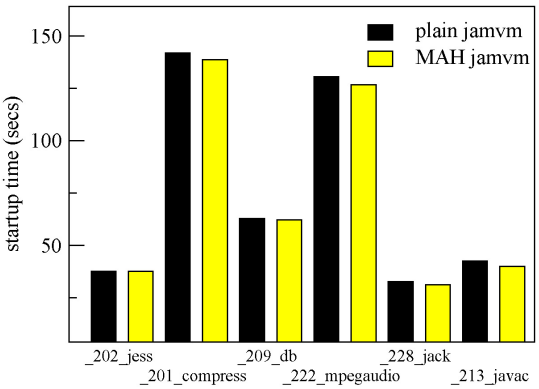


**Figure 7: Performance evaluation using the SPEC JVM98 benchmark**

# 6. RELATED WORK

Sharing data among multiple JVM instances has been exploited by various research projects. Czajkowski et al. explored the sharing of byte code and compiled code [5]. Along these lines Class Data Sharing (CDS) [10], as supported by the current J2SE 5.0, shares the virtual machine internal representation of system classes among multiple virtual machine instances. This saves start-up time and decreases memory consumption as the effected system classes are no longer replicated in each JVM instance. However, CDS is limited to system classes whereas our approach explicitly targets the fine-grained selection of application code that should be used for a fast recovery. The IBM SDK 5 and the upcoming version 6 provide more sophisticated class sharing concepts, named class caches, that even enable the sharing of application classes among multiple JVM instances [4]. Thus, this approach can be directly compared to the the proposed recoverable class loaders, but it is restricted to static parts of the JVM internal representation of classes, whereas the presented approach restores all data related to a class loader, which should lead to a better start-up performance.

Microreboots [2] are a technique for fast recovery from software faults by rebooting the transitive closure of dependent software modules. The key idea is to have components that can be recovered independently of the execution infrastructure and other components. This enables a fine-grained and fast cure for certain software faults using reboots. How-

ever, this approach is limited to software faults of components and misses support for faults at the middleware and infrastructure level. This problem is attacked by the presented recoverable class loaders that enable a fast reboot of the whole virtual machine.

The MERPATI [13] system enables persistent serialisation of the current execution state of a running JVM, including all actually active Java threads therein. In this manner a migration of an initialised JVM instance and the currently executed Java program on top is intended. At a certain preemption point all running Java threads are suspended. However, the entire Java stacks of the threads, affected objects and necessary type information are then serialised within a persistent snapshot. Thus, after migration, the persistent execution state is reconstructed in a new host JVM, using the snapshot. Although MERPATI stores the entire JVM state persistently, it is not intended for a speed-up of the JVM bootstrap process, but for application migration. The applied snapshooting techniques premise a completely initialised JVM instance even on host side. The reuse of the entire objects pool and threads neither allows software reconfiguration, nor remedy actual software faults.

Additionally IBM Hursley propose a design approach for a *Persistent Reusable JVM (PRJVM)*. In [1] they assume, that serial execution of transactions in the same JVM instance, e.g. method invocations of an EJB component, lack on isolation. However, an initialisation of a completely new JVM instance from scratch for each transaction is not acceptable because of performance reasons. The PRJVM features a sophisticated memory partitioning, with a system, middleware and a transient heap. Each heap area is provided with separate garbage collection and memory management, adjusted to its own GC and allocation rates. After each transaction execution the JVM is *reseted*, whereas the middleware and the transient heap are cleaned-up and reinitialised. On heavy-weight errors, or static information alteration, the PRJVM is considered as *dirty* and has to be destroyed and rebooted. The PRJVM approach also does not improve the startup performance of a total reboot; therefore those are delayed as long as possible. Only lightweight faults are cured by reinitialisation of the affected system part and heaps.

Finally, we want to mention the *Clonable JVM* [7], developed by the IBM Tokyo Research Laboratory. The presented concept targets at transaction isolation, failure recovery and checkpointing of Java applications. For that purpose, a running JVM instance is *cloned* by means of the `fork()` UNIX system call. Thereby, a full-fledged copy of the JVM memory is created and, if required, also persistently stored. After short initialisation, the generated clone can be continued immediately, or can be invoked later on demand. Thus, this approach captures the whole execution state of a Java application, including all OS ressources e.g. running threads, mutexes and even file management structures. However, as already explained, full-fledged snapshots are not useful for a fast recovery to remedy software faults, as targeted by our recoverable class loaders concept.

# 7. CONCLUSION

The presented approach for recoverable class loaders enables a fast restart and recovery of Java applications. Both these claims have been verified by our prototype application and by the experimental evaluation. Particularly, Java systems that utilise a sophisticated class loading infrastructure, e.g. OSGi, can benefit from recoverable class loaders and enhance their start up performance enormously. However, even for the case of traditional non-component based Java applications our modified JVM exhibits a much better boot-performance and a constant overall-performance. Moreover our prototype system executes also standard non-RC-aware Java programs and is fully operational under real circumstances.

In spite of these results there is a minor issue — the handling of static class constructors. So far, we have been handling this issue manually in our implementation, but providing automated support seems also possible in the near future.

# 8. REFERENCES


[1] S. Borman, S. Paice, M. Webster, M. Trotter, R. McGuire, A. Stevens, B. Hutchison, and R. Berry. A serially reusable java(tm) virtual machine implementation for high volume, highly reliable, transaction processing. Technical Report TR 29.3406, IBM Hursley, Hursley Park, UK.

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *OSDI'04*, pages 31–44, Berkeley, CA, USA, 2004. USENIX Association.

[3] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks (http://www.spec.org/osg/jvm98/).

[4] Ben Corrie. Java technology, ibm style: Class sharing (http://www.ibm.com/developerworks/java/library/j-ibmjava4/), 2006.

[5] G. Czajkowski, L. Daynès, and N. Nystrom. Code sharing among virtual machines. In *ECOOP '02*, pages 155–177, London, UK, 2002. Springer-Verlag.

[6] The IEEE and The Open Group. mmap() The Open Group Base Specifications Issue 6, IEEE Std 1003.1, (http://opengroup.org/onlinepubs/009695399 /functions/mmap.html), 2004.

[7] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu, and T. Nakatani. Cloneable jvm: a new approach to start isolated java applications faster. In *VEE '07*, pages 1–11, New York, NY, USA, 2007. ACM Press.

[8] T. Lindholm and F. Yellin. *The Java (TM) Virtual Machine Specification (Second Edition)*. Sun Microsystems, second edition, April 1999.

[9] Robert Lougher. JamVM (http://jamvm.sourceforge.net/), 2007.

[10] Sun Microsystems. JDK 5.0 Documentation (http://java.sun.com/j2se/1.5.0/docs/).

[11] Sun Microsystems. Official Java EE website (http://java.sun.com/javaee/), 2007.

[12] The OSGi Alliance. OSGi service platform: Core specification, release 4. Technical report, 2005.

[13] Takashi Suezawa. Persistent execution state of a java virtual machine. In *JAVA '00: Proc. of the ACM 2000 Conf. on Java Grande*, pages 160–167, New York, NY, USA, 2000. ACM Press.