Cloneable JVM: A New Approach to Start Isolated Java Applications Faster

Kiyokuni Kawachiya Kazunori Ogata Daniel Silva *
Tamiya Onodera Hideaki Komatsu Toshio Nakatani

IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242-8502, Japan
<kawatiya@jp.ibm.com>

Abstract

Java has been successful particularly for writing applications in the server environment. However, *isolation* of multiple applications has not been efficiently achieved in Java. Many customers require that their applications are guarded by independent OS processes, but starting a Java application with a new process results in a long sequence of initializations being repeated each time. To date, there has been no way to quickly start a new Java application as an isolated OS process.

In this paper, we propose a new isolation approach called *Cloneable JVM* to eliminate this startup overhead in Java. The key idea is to create a new Java application by copying, or cloning, the already-initialized image of the primary JVM process. Since the clone is already initialized, it can begin actual operations immediately as a new isolated process. This cloning abstraction can support new scenarios for Java, such as user isolation and transaction isolation.

We implemented a prototype of the Cloneable JVM by modifying a production JVM on Linux, which provides a new API for cloning constructed on the Isolate API defined in JSR 121. Using this cloning API, several Java applications, including a large production J2EE application server, were modified to demonstrate the isolation scenarios. Evaluations using these prototypes showed that new ready-to-serve Java applications can start up as a new process in less than 5 seconds, which is 4 to 170 times faster than starting these applications from scratch.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—frameworks

General Terms Languages, Design, Performance, Experimentation

Keywords Java, startup overhead, cloning, isolation

* Daniel Silva joined this project as an intern from Northeastern University, and is currently a Ph.D. student at Harvard University.

VEE'07, June 13–15, 2007, San Diego, California, USA.

1. Introduction

Java [18] has become very popular, especially in server-side environments such as those used for Web services. One important feature required in the server environment is to run each application in *isolation*, in order to protect it against other applications' failures. The most preferred and proven approach for this isolation is to run each application on an independent process, which is strongly guarded by the underlying operating system. However, this approach has not been efficient in Java, since starting a Java application as a new process takes a long time. For example, IBM's WebSphere Application Server (WAS) [22] Version 5.1.1, which is middleware for hosting Java 2 Enterprise Edition (J2EE) [45] applications, takes about 14 seconds to start up on a 3.06 GHz dual Xeon PC, even if no J2EE application is installed. The situation is worse when J2EE applications are installed, and may take several minutes in actual configurations. This slow startup problem prevents Java from being used for various isolation scenarios.

Figure 1 illustrates why the startup of Java applications is so slow. The upper bar (JVM 1) shows the image of the tasks[1] performed when a Java application starts up. Before the application becomes ready-to-serve, the Java virtual machine (JVM) [32] completes a long startup sequence with actions such as loading and initializing classes, creating objects, constructing internal data structures, compiling bytecodes, and setting up the middleware. For example, about 6,000 classes need to be loaded and initialized to start WAS 5.1.1, even without any J2EE application. All Java applications pay this *startup overhead* tax every time they are started.

One existing approach to reduce the startup overhead is to share and reuse data structures among multiple JVMs [3, 4, 9, 11, 12, 20, 44]. If the loaded class structures and JIT-compiled native code blocks are reused through sharing, subsequent applications can be started faster. However, this sharing approach does not completely eliminate the startup overhead. Because each Java application is started from its entry point, there still remains a significant overhead, such as class initialization and object creation, as shown in the lower bar (JVM 2) of Figure 1. For example, in the case of WAS 5.1.1, class loading took about 25% and JIT compilation took 10% of the startup time. This implies that 65% of the startup overhead still remains even if the loaded classes and JIT-compiled code blocks are aggressively shared and reused among JVMs. In addition, to implement such data sharing, both the JVM and JIT com-

[1] Figures 1 and 2 show the breakdown of the startup overhead into several major tasks, but do *not* mean that these tasks are performed sequentially. In addition, each task contains processing for the middleware and the application as well as for the Java environment itself.

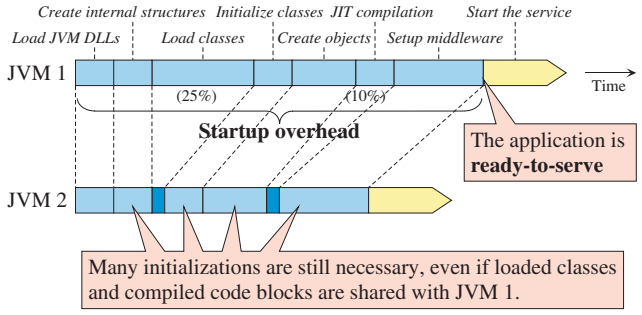

Figure 1. Startup overhead of Java application, most of which still remains even if some internal data structures are shared.

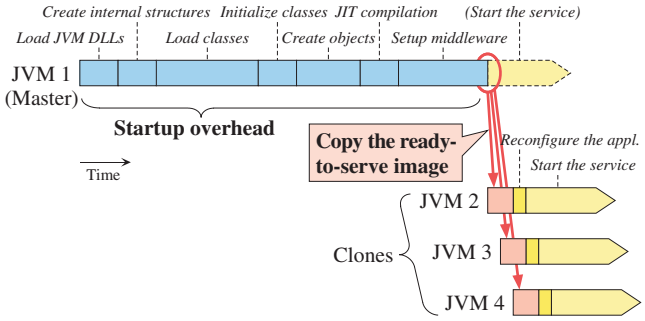

Figure 2. Create new Java execution environments by *cloning*.

piler must be extensively modified, possibly degrading execution performance of the application.

The Isolate API defined by the Java Community Process as JSR 121 [26] has the same limitation, since each Isolate must start an application from its entry point. A new execution abstraction should be introduced to reduce the startup overhead further.

In this paper, we propose a new isolation approach called *cloning* to remove almost all of the startup overhead by copying the initialized image of the primary JVM process into the new process after the target Java application is initialized. Figure 2 illustrates this cloning startup, where the JVMs 2 to 4 are started almost instantly by cloning the already-initialized JVM 1. Each cloned environment contains the initialized image of the application, and the application is resumed from the point of cloning instead of being started from its entry point. Therefore, it can begin actual operations after a few reconfiguration steps without redoing the initialization. Since each cloned environment is guarded by an independent OS process, new scenarios such as user isolation and transaction isolation become possible with this cloning abstraction.

We implemented the cloning function by modifying a production JVM as a *Cloneable JVM*, without modifying the underlying operating system. With this approach, we could implement the cloning function efficiently, since the internal structure of the JVM could be freely adjusted for cloning. Using a new cloning API provided by the Cloneable JVM, several Java applications were modified to demonstrate the isolation scenarios. Evaluation using these prototypes showed that new ready-to-serve Java applications can start up as a new process in less than 5 seconds, which is 4 to 170 times faster than starting these applications from scratch.

To the best of the authors' knowledge, this is the first successful demonstration of cloning the entire Java environment for isolation. The contributions of this paper are:

- Proposal of a new execution abstraction, cloning, where an initialized Java application is duplicated to start a new isolated environment almost instantly.
- The descriptions of several scenarios made possible by the cloning.
- Detailed design and implementation of the Cloneable JVM and the cloning API on a production-level JVM.
- Evaluation of the cloning using real clone-aware applications as well as micro-benchmarks.

The rest of the paper is organized as follows. Section 2 explains the key concepts of cloning by presenting several usage scenarios, a programming model, and design issues. Section 3 describes the necessary modifications to the JVM layer, and Section 4 shows actual Java applications that exploit cloning. Section 5 presents the effectiveness of cloning with the results of various measurements. Section 6 discusses related work, and Section 7 offers conclusions.

2. Key Concept

The cloning presented in this paper is an aggressive approach that tries to eliminate all startup overhead. The key idea is to create a new Java environment by copying the image of an already-initialized, ready-to-serve Java application, including its internal structures such as classes, objects, and JIT-compiled code blocks.

The idea resembles the `fork` system call of Unix-style operating systems, which creates a new process by duplicating the memory image of the current process. However, that is usually a transient state until a new program is loaded by using `exec`. In addition, a Java execution environment cannot be cloned simply by `fork` since it does not duplicate several OS resources such as threads, mutexes, and file management structures, as will be discussed in Section 3.2.

Before going into the implementation details, in this section we show how the cloning concept can be used with actual Java applications.

2.1 Cloning Scenarios

By using cloning, a new ready-to-serve Java environment can be started immediately as a new OS process separated from the original environment. Therefore, various new usage scenarios become possible, some of which are shown in Figure 3.

The first scenario is *user isolation*. Cloning can provide a Java application environment for each user isolated from those of other users. Figure 3(a) shows an example flow of this scenario. The application is started in advance of the real operations up to the point where initialization has been completed. When a user requests the application to perform the operation, a clone is created from that image and dedicated to the user. The cloned Java environments are executed as independent processes, so they do not interfere with each other. A user's application can be properly isolated from failure in other users' applications.

The second scenario is *transaction isolation*. In transaction processing, there is a requirement to process each transaction on a clean and reliable environment [7]. With cloning, such isolation is realized by processing each transaction in a cloned transaction processing environment. Figure 3(b) shows the flow of this scenario, where each transaction is executed on dedicated middleware, without being affected by other transactions. It is also possible to process multiple transactions in parallel if they are independent or appropriately synchronized. This scenario is also useful to provide a scripting environment by Java. By cloning, each script processing can be started immediately in a dedicated clean environment.

The third scenario is *failure recovery*, as shown in Figure 3(c). In this scenario, after the application is initialized, a clone is created to perform the actual operations. The master monitors the status of the cloned environment, instead of performing the operations. When it detects abnormal termination of the clone, it creates a new clone, which can take over the operation immediately.

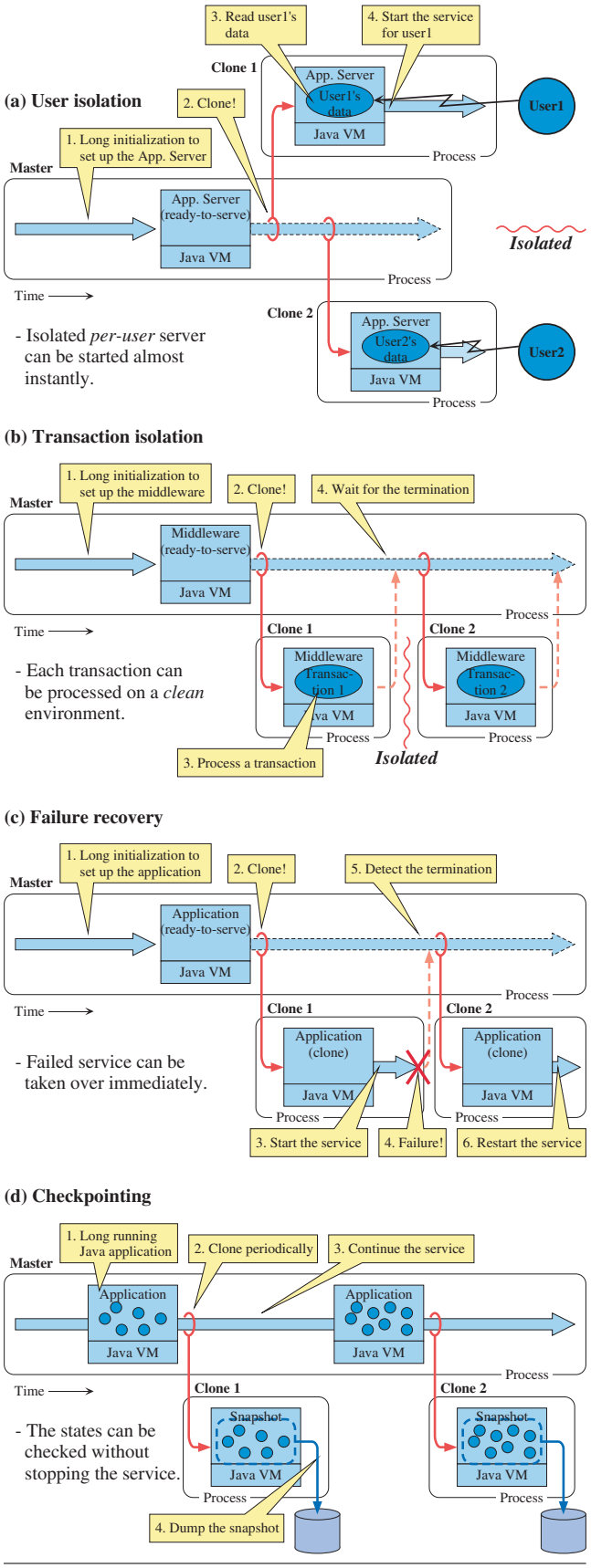

Figure 3. Several scenarios using cloning.

`class Isolate` — is modified to provide the cloning functions.

`static Isolate cloneMe()`
— clones the current Java environment as an Isolate, and returns different values to the master and clone.
`void start()`
— starts the created Isolate (clone).
`int waitFor()`
— waits for the termination of the specified Isolate (clone), and returns its exit code.

`abstract class CloneAction` — is a convenient class to manage CloneAction callbacks.

`static CloneActionList \`
`addCloneAction(CloneAction a, Object o, int when)`
— adds an action invoked while the environment is cloned. The argument `when` is one of `BEFORE_CLONE_IN_MASTER`, `AFTER_CLONE_IN_MASTER`, or `AFTER_CLONE_IN_CLONE`.
`abstract boolean perform(Object o, int when)`
— defines a method invoked before or after the cloning.

Table 1. The provided cloning API, which is an extension of the Isolate API.

The fourth scenario is *checkpointing*, as shown in Figure 3(d). This scenario clones a long-running Java application periodically and uses the clones for dumping snapshots or for checking the internal consistency. A similar idea was reported for C as Libckpt [36]. In this scenario, the dumping or checking can be done completely in the cloned environment without stopping the original application. In addition, the cloned snapshot can also be used as a *hot-standby* image to quickly take over the application if the master crashes.

2.2 Programming Model

As shown in the above scenarios, in the cloning execution model, the *master environment* is started in advance. The cloning is initiated when the master environment invokes a method for cloning at a *cloning point*, after the necessary initialization is finished.

Table 1 summarizes our API for cloning. Since the primary purpose of cloning is to create isolated Java environments, we defined the cloning API by extending the Isolate API designed in JSR 121 [26]. The `cloneMe` method added to the `Isolate` class is the key function for the cloning, and creates a new process by copying the current Java environment.

Figure 4 shows an example of a simple *clone-aware* Java program using this API, implementing the *transaction-isolation* scenario explained above. The vertical bars indicate the portion added for cloning. After the master finished the long initialization, it reads a request at line 9, and creates a clone of the initialized environment by calling the `cloneMe` method at line 12. The master and cloned environments are essentially the same, but different values are returned by the `cloneMe` method. The value `null` is returned to the clone, while an object of the `Isolate` class is returned to the master in order to allow the master to control the clone. Therefore, by checking the returned value, a program can distinguish between the environments in which the program is running and change the processing after the cloning. In this example, the master starts the clone at line 14, and the clone processes the request at line 17. Since the initialization has already been done, the requested operation is started immediately in the clone dedicated to it. The master can wait for the termination of the clone by invoking the `waitFor` method, shown as a comment at line 15. However, in this example, it just moves to the next iteration for reading the next request. There is no problem even if the variable `req` is modified while the previous request is being processed, since it was already copied to another process.

```
  1 import javax.isolate.*;
  2
  3 class CloneAwareServer {
  4         :
  5  public static void main(String[] args) {
  6   initializeServer();      //Long initialization
  7
  8   while (true) {           //Master loop
  9    req = readARequest();   // read a request
 10
| 11    // Create a clone for processing the request
| 12    Isolate clone = Isolate.cloneMe();//cloning point
| 13    if (clone != null) {    //Master
| 14       clone.start();       // start the cloned JVM
| 15     //clone.waitFor();     // then wait, if necessary
| 16    } else {                //Clone
 17       processARequest(req);// process the request
| 18       System.exit(0);      // then exit
| 19    }
 20   } // while (true)
 21  }
 22 }
```

Figure 4. Simple clone-aware Java program, based on the *transaction-isolation* scenario.

By the time the master environment reaches the cloning point, the JVM has initialized many of the states based on the command-line parameters and property files. Likewise, the middleware and the application have also initialized their states based on various configuration files. We assume that the clone basically inherits and uses these states as they exist. However, for some scenarios, it may be necessary to *reconfigure* some states, such as network connections, in the cloned environment. Each clone-aware application is responsible for such reconfiguration, but the cloning API includes a function to assist it. The application can extend the `CloneAction` abstract class and register its instance with an object that should be reconfigured through the `addCloneAction` method. The `perform` method of the class will then be called back before or after the cloning in the master or cloned environment, as specified by the `when` parameter. What should be reconfigured depends on each application and scenario. If the application already supports some kind of reconfiguration, for example through the OSGi framework [34] or Java Management Extensions [24], we can utilize those functions for our purposes.

Since our cloning API is based on the concept of Isolates, we can also use the standard inter-Isolate communication mechanisms defined in JSR 121 [26], such as `Link` and `IsolateStatus`, to control the cloned environment.

2.3 Design Issues

As shown above, various new isolation scenarios become possible using the cloning abstraction, since it can remove almost all of the startup overhead. Next, we discuss several design issues of the cloning functions.

The first, fundamental issue is how to isolate the cloned Java environment. We chose to create a new OS process for each Java environment, which is the most preferred and proven approach for isolation. It might be possible to run multiple Java environments in a single OS process [20]. This approach is suitable for reducing memory consumption, but controlling the resource consumption for each environment is very difficult, which may also degrade the execution performance. In addition, failure in one environment such as an `OutOfMemoryError` may affect other environments.

The second issue is how to control the cloning. As explained in the previous section, we chose an approach to modify each application to be clone-aware, by exposing the necessary cloning API. Another approach might be to control the cloning *outside of* the target application, which is very attractive since it could minimize modifications of the application. However, it is usually difficult to determine the cloning point, where the application is initialized and ready for cloning, from outside of the application. Moreover, each application needs to be modified anyway, because only the application knows what must be reconfigured after the cloning.

The third issue is where to implement the cloning function. We chose the JVM layer as a primary module to implement the functions, because (i) all activities of Java applications pass through the JVM layer, and (ii) the internal structure of the JVM can be easily adjusted to be suitable for cloning. It might be possible to modify the operating system rather than the JVM, but the execution performance might suffer because such OS-level functions would need to be more generic to support arbitrary processes other than just a JVM. In addition, distributing a modified OS is more difficult and potentially less reliable than distributing a modified JVM, even though kernel extension techniques can be used. Therefore, in our implementation, the underlying operating system was not modified at all.

Another possible approach is to use the emerging hypervisor technologies [15, 17]. This approach makes it possible to clone arbitrary applications, but introduces performance degradation and additional overhead of copying the full image of the guest OS. In addition, the concept of cloning is *not* exactly same as the dump/restore or migration supported by hypervisors. In cloning, multiple execution environments run in parallel as shown in Figure 3, which introduces the necessity of reconfiguration. This means that, even in hypervisor-based cloning, applications need to be modified since only they know what should be reconfigured.

3. The Cloneable JVM

The `cloneMe` method described in Section 2.2 is the key function for the cloning provided by the modified JVM, called the *Cloneable JVM*[2]. In this section, we describe the details of the JVM-level modifications for implementing the cloning, which consists of (1) copying the memory image and (2) regenerating OS resources.

3.1 Copying the Memory Image

The first cloning step in the JVM layer is to create a new process by copying the memory image of the master environment. Each memory region must be copied to the same address of the new process to avoid pointer relocation. In addition to JVM data structures such as classes, this step also copies the initialized Java heap image, threads' execution stacks, and DLL code and data, so no initialization will be needed in the cloned environment.

On Unix-style operating systems, the `fork` system call can be used for this step. In recent systems, the memory copy for `fork` is performed virtually by using copy-on-write [38], where the memory pages are shared as read-only between the master and the newly created processes until modified. Therefore, the process can be created significantly faster because the contents of memory pages need not actually be copied. The actual data copying will occur incrementally on a memory page basis when one of the processes tries to write into the read-only page.

Even on an operating system that does not support `fork`, the memory copy step can be implemented using a memory allocation interface while explicitly specifying the logical addresses, such as

[2] The word `Cloneable` is already used in Java to indicate a class whose object can be duplicated by `clone` method. We adopt this word in the meaning that the JVM itself can be duplicated. Actually it is a misspelling of "clonable", but we use the same misspelling to maintain consistency in Java.

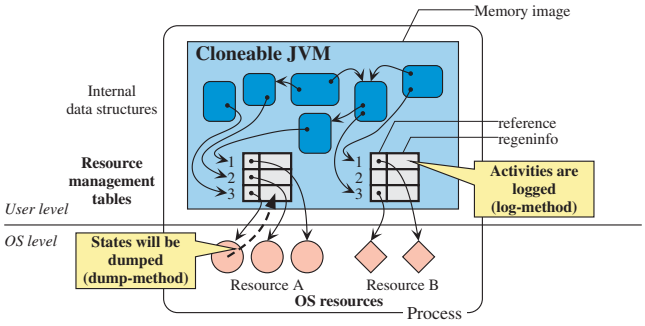

Figure 5. Resource management tables to control all OS resources.

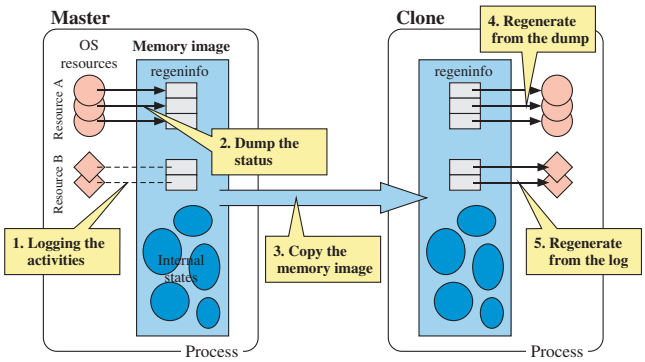

Figure 6. Cloning steps in the JVM layer.

`VirtualAlloc` in Windows. This is because our target is not an arbitrary program but just a Java execution environment, and we can modify the JVM code to record all of the allocated addresses. While a cloned environment is created, the new process allocates the memory regions explicitly at the recorded addresses and copies the contents from the master environment.

3.2 Regenerating OS Resources

With the memory copy step explained above, most of the internal states of the Java execution environment are copied, such as the Java heap and class structures. However, this is not enough for the new process to run as a new Java environment. It is necessary to bring the copied memory image to life by reproducing states inside the OS kernel. Examples of such states are the register contexts of the threads, the internal states of any mutexes, and the internal file management structures. These are usually first class resources provided by the operating system, so we call them *OS resources* here.

Regeneration of these OS resources in the new environment is the second cloning step in the JVM layer, where two issues must be solved. The first issue is that the states of the OS resources exist inside the OS kernel and may not be represented in the memory image of a process. The memory image usually contains just *handles* (or descriptors) to control the OS resources. The second issue is that even if OS resources are regenerated in the cloned environment, the values of their handles may be different from those in the master environment.

To implement the OS resource regeneration while solving these problems, we modified the JVM code to centrally manage all OS resources through *resource management tables*, as shown in Figure 5. In Java, OS resources are not directly accessed from an application but are accessed through the JVM. Therefore, the cloning mechanism can control all of the OS resources used in the Java execution environment through these tables. If a Java application uses its own non-Java native code through the Java Native Interface [31] and the code directly uses some OS resource, the application may not be correctly cloned since that resource is not managed by the Cloneable JVM. This problem does not occur for "100% Pure Java" applications.

An entry of the resource management table corresponds to an OS resource and consists of two fields, `reference` and `regeninfo`. The `reference` field contains a handle for the OS resource. The JVM code is modified to access the OS resources *indirectly* through the tables. Only the table entry contains the actual handle, while other data structures in the JVM are modified to point to the table entry. The other field, `regeninfo`, is used to store information necessary for regenerating the OS resource. In the new environment, the Cloneable JVM regenerates the OS resources using the information in the fields, which are copied by the memory copy step, and stores the new handles into the corresponding `reference` fields. Since handles are used only in the `reference` fields, it does not cause any problem even if their values are different from those in the master environment.

There are two methods to store information into the `regeninfo` field. First, we use the *dump-method* for OS resources whose internal states can be retrieved from a user-mode program, which is a JVM in our situation. For such resources, their states are retrieved and dumped into the `regeninfo` field just before copying the memory image. Unfortunately, for some OS resources, we cannot retrieve their internal states. For such resources, we use the *log-method*, where operations on the resources in the JVM are hooked and logged into the `regeninfo` field. The logging overhead is small compared to that of system calls to control the OS resource.

3.3 Implementation Details on Linux

By using these memory copy and OS resource regeneration techniques as explained above, we can create a new Java execution environment as a clone of the master Java environment. Figure 6 summarizes the general cloning steps in the JVM layer, where Resource A is regenerated by the dump-method and Resource B is regenerated by the log-method.

We developed the Cloneable JVM by modifying the IBM J9 Java virtual machine [19] for Linux. The memory copy is performed by `fork`, and the threads and file management structures are regenerated in the new environment. In Linux, the `fork` system call copies other OS resources used in the JVM such as the mutexes, so it was not necessary to explicitly regenerate them in the prototype.

Figure 7 shows the actual flow of cloning in the prototype. When the application requests cloning by calling the `Isolate.cloneMe` method, the Cloneable JVM first sets a global *cloning flag* which indicates that cloning is underway. To lock up the memory image that should be copied, all running threads other than the thread that requested the cloning, called the *initiator thread* here, are suspended (M1). During the suspension process, each thread executes the `setjmp` function to dump its register context into the corresponding `regeninfo` field, and waits for a resume message from the initiator thread (M1a).

The J9 JVM already has a mechanism to cooperatively suspend threads at a *GC-safe point* for a stop-the-world type of garbage collection, and we used this mechanism for the suspension. However, a thread blocking at a system call such as `accept` is not suspended by this mechanism since such a thread is marked as GC-safe before issuing the system call and need not be suspended for GC. For the cloning, such threads must also be suspended to dump their contexts. Therefore, in the current implementation, we specify a timeout for the blocking system calls in order to periodically check the cloning flag. Currently, the timeout is set to 1 second, so the cloning

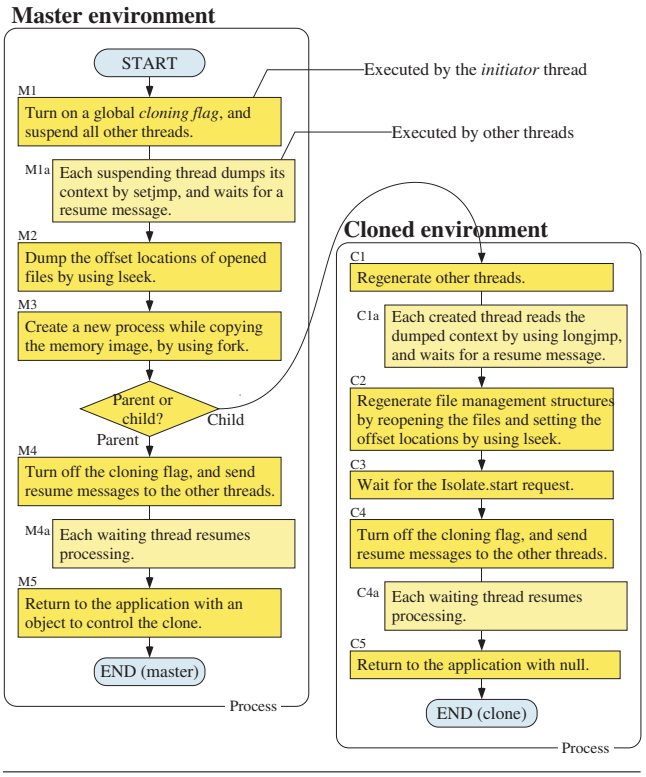

Figure 7. Flow of cloning in the Cloneable JVM on Linux.

time may be extended to several seconds in the worst case. For usage scenarios that require significantly fast cloning (of less than 1 second), an alternative implementation may need to be pursued, for example, to force the blocking thread's suspension by a signal or interrupt. In addition, the JVM creates several *helper threads*, which are not bound to Java threads, to help with GC or JIT compilation. These must also be suspended, so we modified the JVM code for each helper thread to check the cloning flag for suspension.

When all threads are suspended and their contexts have been dumped, the initiator thread prepares information for regenerating the file management structures in the new environment. The Cloneable JVM regenerates this OS resource by reopening each file and setting its seek pointer. Here, the seek pointer value of each file is retrieved by the `lseek` system call and stored into the corresponding `regeninfo` field using the dump-method (M2). For the file name and the access flag, we modified the JVM code to record the information into the `regeninfo` field using the log-method when each file is opened.

After these dump processing steps are finished, the initiator thread creates a new process by using the `fork` system call (M3). After the process creation, the master's initiator thread turns off the cloning flag and sends a resume message to each thread (M4) to restart the execution of all of the threads (M4a). An object of class `Isolate` to control the cloned environment is created and returned to the application (M5).

In the new process created by the `fork`, only the initiator thread is running, so it first regenerates the other threads (C1). The threads are created one-by-one and execute the `longjmp` function to read the register context dumped in the `regeninfo` field (C1a). This `longjmp` forces the thread to return to the point where the `setjmp` was executed. There each thread waits for a resume message from the initiator thread. Note that the stack area of the regenerated thread is also switched into the copied memory area by executing the `longjmp` because the stack pointer register is also replaced.

File descriptors, which are the handles used to access files, have been automatically copied to the new process by the `fork`. However, each file management structure in the OS kernel represented by a file descriptor is now shared between the master and new environments. To avoid interference, the initiator thread in the new environment regenerates the file management structures (C2). This regeneration is done for regular files opened as read-only, by reopening each file and setting its seek pointer using `lseek` with the information dumped in the `regeninfo` field. File descriptors which represent writable files or connected sockets are closed in the new environment. Clone-aware applications are responsible for reconfiguring these resources appropriately in the cloned environment.

After all of the threads and file management structures are regenerated, the initiator thread in the cloned environment waits for a start request from the master environment (C3), which will be sent when the master invokes the `start` method for the returned `Isolate` object. On receiving that request, the initiator thread turns off the cloning flag and sends a resume message to each thread (C4). In the new environment, the `cloneMe` method returns `null` to the application that requested the cloning (C5).

These are all of the cloning steps that we have implemented in the Cloneable JVM. A new Java process which has fully-initialized states has been created, and starts running independently. This implementation added about 5,000 lines[3] to the JVM and class libraries, which is very small compared to their total code size.

In the implementation of the Cloneable JVM for Linux shown here, the memory copy is performed virtually by using copy-on-write. Therefore, pages that are not modified after cloning by either of the two environments remain shared as read-only, which can reduce the overhead of actual memory copy and the system-wide memory consumption. The J9 JVM manages the class data structure by dividing it into read-only portions and writable portions, and the read-only portions can remain shared after the cloning.

If objects are moved by garbage collection, it causes additional page separations by the copy-on-write. Since the copying-type GC would cause immediate page separations, a traditional single heap mark-and-sweep GC [27] was chosen for the Cloneable JVM. Although not performed automatically in the cloning step, it is also possible for each application to explicitly execute GC before the cloning, to tidy up the Java heap area.

4. Clone-Aware Java Applications

By using the cloning API provided by the Cloneable JVM, a Java application can create its clone almost instantly. However, since it differs for each application and scenario how the cloning is used and when the clone should be created, the application should be slightly modified to be *clone-aware*, as explained in Section 2.2.

At present, we have already developed clone-aware prototypes of several real Java applications. This section describes the usage of cloning and the modifications in each application.

4.1 Clone-Aware HTTP Server

For the first experiment in cloning a real application, we chose the Jigsaw HTTP server [49], an HTTP server written in Java, developed and distributed by the World Wide Web Consortium (W3C). We modified Jigsaw Version 2.2.4 and implemented a *failure-recovery* scenario using cloning.

In the clone-aware Jigsaw, a clone is created at the point where the initialization of creating objects and worker threads has finished and just before starting the actual HTTP service. After cloning, the

[3] This number does not include the changes we made separately to add the Isolate API to the JVM, which was about 4,000 lines.

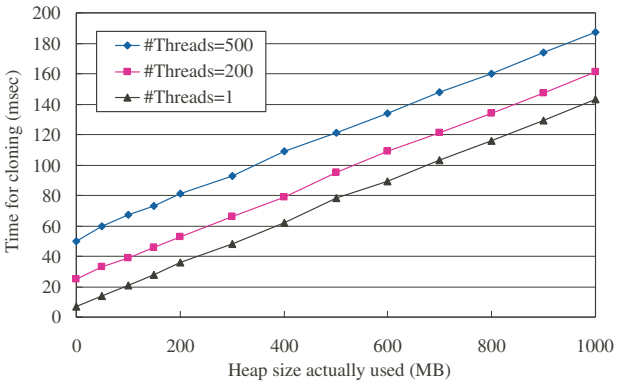

Figure 8. The time required to clone with various heap sizes actually used.

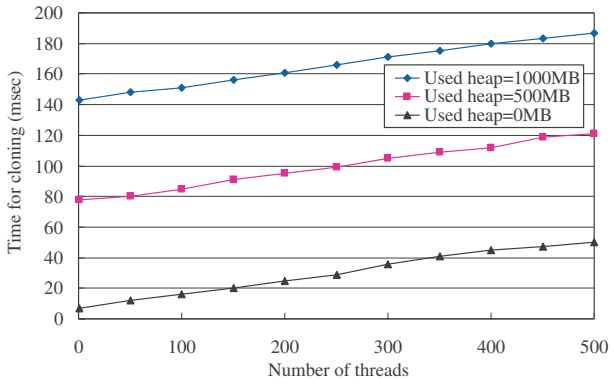

Figure 9. The time required to clone with various numbers of threads.

clone continues to perform the HTTP service, while the master waits for the termination of the clone. If the clone terminates abnormally, for example due to some erroneous servlet, the master detects it and creates another clone to recover the HTTP service immediately. This is possible because the master is isolated from the clone and not affected by its abnormal termination.

Less than 100 lines were changed for the clone-aware modification, which is very small compared to the approximately 160,000 lines of the original Jigsaw.

4.2 Clone-Aware XML Parser Generator

The next experiment was done with an XML parser generator, which is a Java program that reads an XML Schema [50] and generates the corresponding XML parser for validating XML data. The version we used has about 90,000 lines of Java code and requires a considerable amount of time to initialize the internal states before compiling the specified XML Schema files.

In the clone-aware XML parser generator, after the initialization is finished, a clone is created to compile each XML Schema file. For each input file, the compilation is started immediately by a cloned generator that has no need to be re-initialized. Fewer than 200 lines were modified to make this application clone-aware.

Here, cloning provides a powerful mechanism to allow all of the clones to run in parallel even if the original Java program is not reentrant, since each clone is an independent Java environment that shares no global data. This is considered to be a variation of the *transaction-isolation* scenario shown in Section 2.

4.3 Clone-Aware Application Server

Although these first two clone-aware applications clearly illustrate the feasibility of cloning, their startup times were relatively trivial even without cloning. To demonstrate a noticeable reduction in startup time, a larger scale Java program should be cloned. Therefore, we modified IBM's WebSphere Application Server (WAS) [22] Version 5.1.1 to be clone-aware, based on the *user-isolation* scenario explained in Section 2.1. As explained in Section 1, WAS 5.1.1 takes several minutes to start up in actual configurations, so it had not been realistic to start a new server for each user who wants to be isolated.

In our prototype of the clone-aware WAS, after all of the WAS components and installed EJB applications are initialized, a special thread is created to receive a *cloning request*. When the cloning request is sent from a user, the thread executes the `cloneMe` method and a new WAS environment is created by cloning. The cloned WAS first reconfigures all of the network ports to have non-conflicting port numbers, then starts operating as an application server dedicated to the user. It is also possible to create multiple WAS clones by sending the cloning request repeatedly.

The clone-aware WAS was created by adding about 900 lines. With these changes, we succeeded in cloning the WAS with *unmodified* EJB applications, which will be described and evaluated in Section 5.3.

5. Evaluation

Using the Cloneable JVM and the clone-aware applications described in the previous sections, we measured the performance of cloning from various viewpoints. All of the measurements were done on a 3.06 GHz dual Xeon PC with 4 GB of memory, running the Red Hat Enterprise Linux 3 AS operating system. The Java heap size was set to 1,024 MB.

5.1 Micro-Benchmarks

First, the performance of cloning at the JVM level was evaluated. The time taken to create a clone by calling `cloneMe` was measured for various JVM internal states. It turned out that the cloning time is mainly affected by the size of allocated Java heap area and by the number of Java threads.

Figure 8 shows the time required to clone when the total size of objects created by the test program was changed from 0 MB to 1,000 MB. The cases of 1 thread, 200 threads, and 500 threads are shown. The graphs are almost linear, with the cloning time increasing about 13.6 ms for every 100 MB of heap space that was used. For example, it took about 143 ms for a 1,000 MB heap with 1 thread. This is because the memory copy cost is increased for the larger heap sizes. In our prototype on Linux, the memory copy is performed using copy-on-write and the contents of the heap are not actually copied at the time of cloning. Only the page management structures need to be prepared to share the contents as read-only. Therefore, the cloning cost is quite small even for the larger heap sizes.

The fast memory copy by copy-on-write may not be available for other operating systems. For such environments, the memory contents must actually be copied during the first step of cloning. To estimate the cost of such cloning with copying, the speed of memory copy was measured. It took about 111 ms to copy 100 MB of page-aligned data using the `memcpy` library function in the same Linux environment. Because this value would add up, the cloning cost is estimated to be 125 ms for every 100 MB of heap, which is about 10 times slower than when copy-on-write is used. Based on this, it is expected that cloning a very large application without using copy-on-write should take about 1 to 2 seconds. Whether

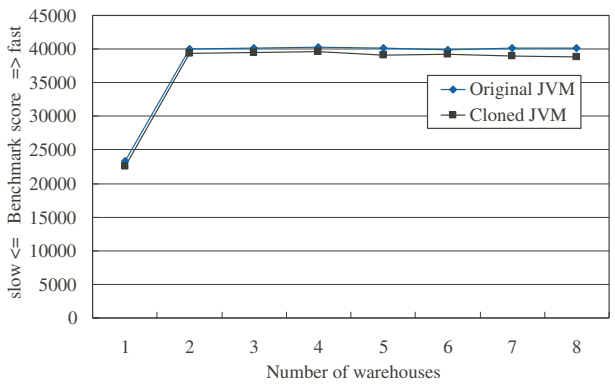

Figure 10. pBOB scores in original and cloned environments.

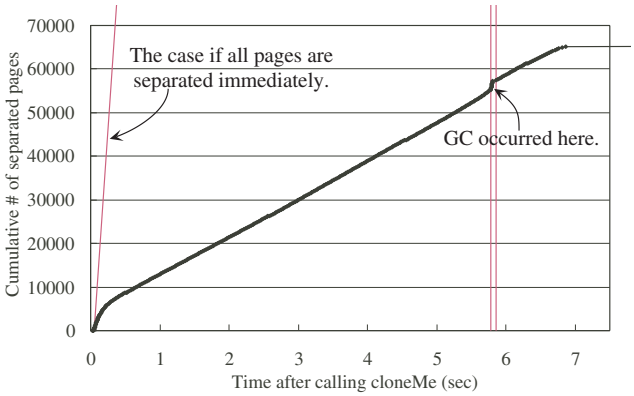

Figure 11. Copy-on-write activities in the cloned pBOB.

this is acceptable or not depends on the specific scenario using the cloning, but it is still much faster than starting the application from scratch.

Another factor that affects the cloning time is the number of threads. Figure 9 shows the time required to clone when the number of Java threads was increased from 1 to 500. The cases where the total size of the created objects is 0 MB, 500 MB, and 1,000 MB are shown. The cloning time increased about 0.9 ms for every 10 threads, reaching 50 ms with 500 threads even if no objects are allocated by the test program. This is due to the costs for suspending, regenerating, and resuming the increasing numbers of threads. Although the impact of the number of threads is not so large compared to the allocated heap size, it should be taken into account for applications that create many worker threads.

5.2 Execution Performance

Next, the execution performance of the Cloneable JVM was measured. For the measurement, we used a benchmark program called pBOB (Portable Business Object Benchmark) [6], which consists of about 30,000 lines of Java. The pBOB is a multithreaded business transaction benchmark used to measure the performance of Java execution environments. Scalability can also be measured by changing the number of emulated *warehouses*. In the normal configuration, after the benchmark environment is initialized, 30 seconds of ramp-up execution is performed, and then the actual benchmark is executed for 2 minutes. The ramp-up is done to stabilize the execution environment, and the major methods for the benchmark are JIT-compiled during this phase.

We modified the benchmark to create a clone just before the actual measurement is started after the initialization and the ramp-up execution are completed. Since a log file is opened during the initialization phase, we added reconfiguration code for the cloned environment to open a new log file and to copy the contents of the original log file. Although the master and clone can run simultaneously, that makes the benchmark score meaningless. Therefore in the clone-aware pBOB, the clone first executes the benchmark, while the master waits for the termination of the clone. The total changes needed to make it clone-aware were about 200 lines.

Using the pBOB benchmark, the scores on the original unmodified JVM and those on the cloned JVM were measured. The benchmark was performed separately for each of 1 to 8 warehouses. Figure 10 shows the results. The score on the cloned JVM was 1 to 3% worse than the original JVM. The reason is believed to be the overhead for making the JVM cloneable, such as using the resource management tables. Another possible reason is the overhead for separating pages by copy-on-write during the benchmark. Anyway, the performance degradation is very small compared to the slowdowns from other approaches such as modifying the JIT compiler to generate sharable code or executing the JVM and OS on a hypervisor.

Figure 11 shows the page separation activity caused by the copy-on-write during the pBOB benchmark on the cloned JVM. This measurement was done for the case of 1 warehouse with the heap size specified as 256 MB because of the limitations of the evaluation system. The x-axis is the elapsed time from the `cloneMe` call, and the y-axis shows the cumulative number of separated pages.

The cloning was completed at 47 ms, and the benchmark measurement started. The pBOB benchmark continuously creates objects during its execution. Therefore, most of the 256 MB heap area, which is 65,536 pages, were written to and therefore actually copied in the first 7 seconds of the measurement period. However, this is much slower than copying the whole heap at once. Because copying the 256 MB of memory takes only 280 ms, it can be said that the page separation occurred *gradually* during the benchmark execution. We can say that the copy-on-write is functioning effectively for the initial goal of providing a new ready-to-serve Java environment more quickly.

Another interesting point in Figure 11 is in the period from 5.78 to 5.85 seconds, where garbage collection was performed. This was the first GC after the clone was created, so the number of separated pages increased because the GC work area was modified. However, excessive page separation did not occur, because the mark-and-sweep GC does not move the live objects.

For the pBOB measurements, almost all of the heap area was separated eventually because pBOB creates many objects after cloning. However, we believe that most pages will remain shared for an application in which most objects are created during the initialization phase and most of them are read-only, or in scenarios where the operations of the cloned environment are finished quickly, as expected in the transaction-isolation scenario shown in Section 2.

5.3 Startup Time Reduction

The main objective of cloning is to create isolated ready-to-serve Java environments instantly. Therefore, the startup times were measured for the clone-aware applications described in Section 4 and the clone-aware pBOB used in Section 5.2. Here, the startup is defined as the period until the application becomes ready-to-serve.

Figure 12 summarizes the reduction of startup time by cloning. For each application, the left bar shows the original startup time when started from scratch, and the right bar shows the time necessary to create a clone with the application-level reconfiguration. The original startup times of pBOB[4] and WAS were affected by

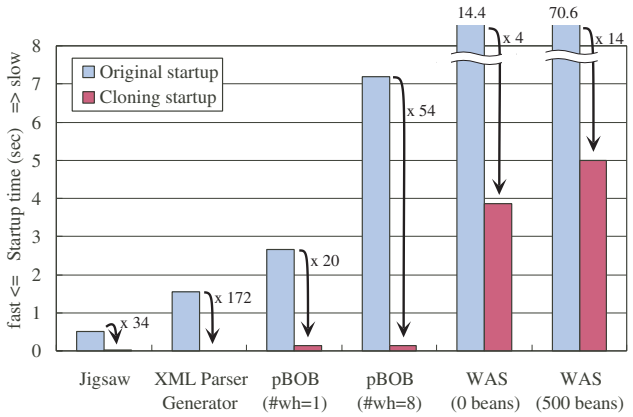

Figure 12. Improvement in the startup time by cloning.

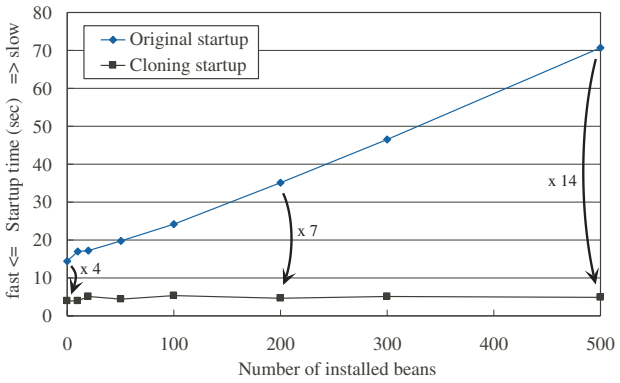

Figure 13. Startup times of WAS with a stress-test application.

their configurations, so two cases are shown for each of these two applications. For WAS, more detailed results will be shown later.

The number beside each arrow in the graph shows the ratio of startup improvement. By using cloning, a new ready-to-serve environment could be created from 4 to 170 times faster than the original startup. As shown in Section 5.1, the JVM-level cloning time mainly depends on the allocated heap size and the number of threads. The XML parser generator could be cloned significantly faster, in 9 ms, because its allocated heap size is not very large and no extra threads exist at the cloning point. Jigsaw needed more time for cloning, about 15 ms, because about 60 worker threads are running at the cloning point. The reason for pBOB's longer cloning time is the large heap size. In addition, pBOB took about 9 ms for reconfiguration, in which the log file was separated.

For WAS, the cloning took a longer time because some threads are blocking within the `accept` system call at the cloning point. In the current Cloneable JVM implementation, a timeout mechanism is used to suspend such blocking threads, which made the cloning slower. Although this part has room for improvement, the startup time of the cloned WAS was already 4 to 14 times faster than that of the original WAS.

Next, we performed a more detailed evaluation of the clone-aware WAS by installing an EJB application for a stress test on it. This application uses many session beans, whose number can be specified at deployment time. When many beans are installed, the startup of WAS becomes slower in order to create, initialize, and start them. While changing the number of installed beans, we measured the original startup time of WAS and the time for cloning this initialized WAS environment. For these measurements, the EJB application was *not* modified at all. Only the JVM and WAS were modified for cloning as already described in Sections 3.3 and 4.3.

The results are shown in Figure 13. The original startup time of WAS became slower in proportion to the number of installed beans. It took about 70 seconds when 500 beans were installed. In contrast, the clone of the initialized WAS could be created in almost constant time, just 4 to 5 seconds, without being noticeably affected by the number of installed beans. This result includes the time for reconfiguring the network ports in the clones, so the cloned WAS environment can start operations immediately at that point.

To summarize the results in this section, the cloning approach can create an isolated ready-to-serve environment 4 to 170 times faster than starting the application from scratch, without degrading the execution performance. This is very promising for the new isolation scenarios based on cloning, as discussed in Section 2.1.

[4] For pBOB, the original startup time means the time for finishing the initialization, and does not include the 30 seconds ramp-up time.

6. Related Work

This section will introduce related work from several viewpoints, and compare that work with the cloning approach.

6.1 Application Isolation

Java is designed to execute a single application although it supports parallel processing primitives such as threads. If multiple applications are forced to run in a single Java execution environment, unexpected results may occur, because static variables and the resolution status of classes are globally shared. In addition, since there is no way to protect data, an application may be affected by errors in other applications. Therefore, some mechanism to provide an *isolated* Java environment for each application is desired.

The Java Community Process defined an API to create and manage logically isolated Java environments, named *Isolates*, in JSR 121 [26]. Although it is not defined how to implement the Isolate API, its execution model states that an application is started from its entry point on a new Isolate, even if it is the same application currently running. Therefore, the startup overhead cannot be eliminated completely. However, since it also defines various useful inter-Isolate communication mechanisms such as `Link` and `IsolateStatus`, we defined the cloning API by adding some functions to the Isolate API as described in Section 2.2.

Smits [41] is proposing an approach to keep user states for its middleware outside the JVM using a technique called Shared Closures, then dispatching them to VM Containers that can be started very quickly. The approach is very interesting, but it seems that the mechanism is currently dedicated to and tightly coupled with SAP's Advanced Business Application Programming Server.

6.2 Data Sharing Among JVMs

As briefly mentioned in Section 1, if sharable parts of the JVM's internal data structures are separated and reused, a new Java environment can be created more quickly. Czajkowski et al. proposed various methods for such data sharing [10, 11, 12, 20]. KaffeOS [4] and Janos [46] proposed an architecture to run multiple *processes* on a Java environment. Among production Java execution environments, Apple's Java Shared Archive [3], Sun's Class Data Sharing [44], and IBM's Shared Classes [9] make it possible to share class data structures among multiple Java processes.

With these data sharing mechanisms, application startup can be accelerated to some extent because the shared data structures need not be constructed again after the first creation of a JVM. However, in all of these implementations, each Java application is started from its entry point in the new environment. Therefore, the startup overhead cannot be removed completely, as shown in Figure 1,

unlike in cloning. However, it is worth incorporating these sharing techniques in the Cloneable JVM to further reduce the memory consumption.

IBM's Persistent Reusable JVM [21] for z/OS can also create multiple Java environments in a single address space while sharing the class data structures and JIT-compiled code blocks [14]. It also supports a unique mechanism for reusing the Java environment and middleware by *resetting* the middleware after an application finishes [7]. This idea resembles the cloning approach in that the initialized environment is reused. However, the environment can only be used serially, and the middleware must be extensively modified to be resettable.

Adding orthogonal persistence to Java [25, 28] allows some of the cloning scenarios such as checkpointing. In addition to its narrower applicability than the Cloneable JVM, the approach requires a programmer to decide which part of the Java heap should be made persistent, which may not necessarily be trivial.

6.3 Startup Acceleration

Both the data sharing shown above and our cloning are approaches to eliminate the startup overhead by reusing already-initialized (or constructed) data in subsequent Java environments. On the other hand, there is another type of approach, to accelerate or reduce the time-consuming steps in the startup by modifying the JVM.

Multi-level JIT compilation [19, 43] is a typical example of this approach, in which every method is first executed by an interpreter or by being compiled quickly with a low optimization level. Since many methods executed during the startup are used only for initialization and not executed repeatedly, startup time is accelerated by reducing the overhead of the JIT compilation for those methods.

The ahead-of-time (AOT) compilation [40] uses the JIT compiler as a static compiler. Though the generated code may not be as highly-optimized, it would be "good-enough" and fully compliant to the Java specification. The J9 JVM also supports the AOT. Originally, it is used in small devices to reduce footprint by eliminating the JIT compiler. More recently, it is used in WebSphere Real Time [23] to eliminate non-deterministic behaviors due to the JIT.

Class loading and verification are also time-consuming steps during the startup. By performing these steps in advance and converting the class files to some internal format, startup overhead can be reduced [30].

However, even if these techniques are used, the startups of large Java applications are still very slow compared to native applications. This is because there remains a lot of overhead such as class initialization and object creation, as explained in Section 1.

6.4 Freezing and Migration

There is a technique to dump, or freeze, an initialized application image in advance and start the application faster by loading the image. For example, the GNU Emacs editor [16] dumps the Lisp heap after the initialization of Emacs Lisp, and uses the image for ordinary startup. In Smalltalk [29], it is possible to create a snapshot of the environment, which can be used for future restarts. The hibernation of Windows and the snapshot function of VMware [47] are considered to be approaches towards applying the dump technique to the whole system image. In the world of Java, the Jikes RVM [1] uses the dump mechanism to create a *bootstrap image*, which contains the minimum set to start the virtual machine. However, it is limited to the bootstrap image, and cannot dump an arbitrary JVM image with its application.

As a related project, *application migration* is emerging as a new research area [8, 33, 35], where an application can be migrated to another environment by using virtualization techniques [5, 37, 39, 42]. Compared to these approaches, our cloning approach targets creating *multiple* isolated environments rather than suspending and resuming a single application instance. Both the master and clones can coexist and run in the system. For this purpose, we explicitly exposed the cloning API to the applications, to assist in the reconfiguration necessary to run multiple application instances.

The Potemkin virtual honeyfarm system in UCSD [48] utilizes a hypervisor to quickly start multiple *honeypot* environments by *flash cloning*, where copy-on-write is also used to reduce the cost of cloning. It might be possible to use a hypervisor for cloning Java applications, but we chose to implement the function in the JVM layer, as discussed in Section 2.3. Through this approach, we could minimize the execution overhead, as measured in Section 5.2. Recently, JVMs become runnable directly on hypervisors [2, 13]. Comparing them with our Cloneable JVM is one of the future work.

7. Conclusion

This paper described the proposal, implementation, and evaluation of a *cloning* execution model, which is an idea to start a new Java application faster by copying an initialized running Java environment. Since the cloned environment runs as a separate process, it is possible to create an *isolated* Java application almost instantly.

We developed the Cloneable JVM by modifying the IBM J9 JVM for Linux to add functions for memory copy and OS resource regeneration. In this version, JVM-level cloning can be performed in less than 200 ms.

The Cloneable JVM provides a new API for cloning, which is constructed on the Isolate API. Using this cloning API, we have modified several real Java applications to be *clone-aware*, according to various *cloning scenarios* such as user or transaction isolation and failure recovery.

Measurements using the Cloneable JVM and clone-aware applications showed that the time required to create a new application environment by cloning was 4 to 170 times faster than the time required to start the application from scratch. With cloning, the new environment became ready-to-serve in less than 5 seconds for all of the tested cases. In addition, the execution on the Cloneable JVM was only 1 to 3% slower than on the original JVM.

The primary contributions of this paper are: the proposal of a cloning abstraction in Java along with an API, the scenarios using cloning, and the actual implementation and evaluation of the Cloneable JVM. We believe this is the first successful demonstration of cloning the entire Java environment for isolation enablement.

Acknowledgments

We thank Matt Hogstrom, Martin Trotter, John Duimovich, Trent Gray-Donald, Bob Blainey, and Kevin Stoodley for their help and advice on this project. We thank Tom Musta and Joseph Latone, who gave us permission to use their programs. We also thank the members of the Systems group in IBM Tokyo Research Laboratory, who gave us valuable suggestions.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalapeño in Java. *Proc. 14th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, 314–324, Denver, October 1999.

[2] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. *Proc. 3rd ACM Conference on Virtual Execution Environments (VEE '07)*, San Diego, June 2007.

[3] Apple Computer. Mac OS X Java Shared Archive. *Java Development Guide for Mac OS X*, May 2006. `http://developer.apple.com/documentation/Java/Conceptual/Java14Development/Java14Development.pdf`

[4] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI '00)*, 333–346, San Diego, October 2000.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 164–177, Bolton Landing, October 2003.

[6] S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, S. Munroe, R. Arora, and R. Dimpsey. Java Server Benchmarks. *IBM Systems Journal*, 39(1), 57–81, February 2000.

[7] S. Borman, S. Paice, M. Webster, M. Trotter, R. McGuire, A. Stevens, B. Hutchison, and R. Berry. A Serially Reusable Java Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing. Technical Report TR 29.3406, IBM.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. *Proc. 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, 273–286, Boston, May 2005.

[9] B. Corrie. Java Technology, IBM Style: Class Sharing, IBM, May 2006. `http://www.ibm.com/developerworks/java/library/j-ibmjava4/`

[10] G. Czajkowski. Application Isolation in the Java Virtual Machine. *Proc. 15th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, 354–366, Minneapolis, October 2000.

[11] G. Czajkowski and L. Daynès. Multitasking without Compromise: a Virtual Machine Evolution. *Proc. 16th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, 125–138, Tampa, October 2001.

[12] G. Czajkowski, L. Daynès, and N. Nystrom. Code Sharing among Virtual Machines. *Proc. 16th European Conference on Object-Oriented Programming (ECOOP '02)*, 155–177, Málaga, June 2002.

[13] J. Dahlstedt. "Bare Metal" – Speeding Up Java Technology in a Virtualized Environment. Presentation in JavaOne '06, TS-3792, San Francisco, May 2006. `http://developers.sun.com/learning/javaoneonline/2006/coolstuff/TS-3792.pdf`

[14] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. St. John. Building a Java Virtual Machine for Server Applications: The JVM on OS/390. *IBM Systems Journal*, 39(1), 194–210, February 2000.

[15] R. Figueiredo, P. A. Dinda, and J. Fortes (ed). Resource Virtualization Renaissance. *IEEE Computer*, 38(5), 28–69, May 2005.

[16] GNU Project. GNU Emacs. `http://www.gnu.org/software/emacs/`

[17] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6), 34–45, June 1974.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*, Addison Wesley, 2005.

[19] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java Just-In-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. *Proc. 3rd USENIX Virtual Machine Research and Technology Symposium (VM '04)*, 151–162, San Jose, May 2004.

[20] J. J. Heiss. The Multi-Tasking Virtual Machine: Building a Highly Scalable JVM, Sun Developer Network, March 2005. `http://java.sun.com/developer/technicalArticles/Programming/mvm/`

[21] IBM Corporation. *Persistent Reusable Java Virtual Machine User's Guide*, SC34-6201-01, 2001. `http://www.ibm.com/servers/eserver/zseries/software/java/pdf/prjvm14.pdf`

[22] IBM Corporation. WebSphere Application Server: Product Overview. `http://www.ibm.com/software/webservers/appserv/was/`

[23] IBM Corporation. WebSphere Real Time. `http://www.ibm.com/software/webservers/realtime/`

[24] Java Community Process. JSR 3: Java Management Extensions (JMX) Specification. `http://jcp.org/en/jsr/detail?id=3`

[25] Java Community Process. JSR 20: Orthogonal Persistence for the Java Platform. `http://jcp.org/en/jsr/detail?id=121`

[26] Java Community Process. JSR 121: Application Isolation API Specification. `http://jcp.org/en/jsr/detail?id=20`

[27] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, 1996.

[28] M. Jordan and M. Atkinson. Orthogonal Persistence for Java – A Midterm Report. *Proc. 3rd International Workshop on Persistence and Java (PJW3)*, Tiburon, September 1998.

[29] G. Krasner (ed). *Smalltalk-80: Bits of History, Words of Advice*, Addison Wesley, 1983.

[30] C. Laffra, S. Foley, and J. McAffer. Packaging Eclipse RCP Applications. *EclipseCON 2005*, Burlingame, February 2005. `http://www.eclipsecon.org/2005/`

[31] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*, Addison Wesley, 1999.

[32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*, Addison Wesley, 1999.

[33] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. *Proc. 2005 USENIX Annual Technical Conference (USENIX '05)*, 391–394, Anaheim, April 2005.

[34] OSGi Alliance. About the OSGi Service Platform: Technical Whitepaper, November 2005. `http://www.osgi.org/documents/collateral/TechnicalWhitePaper2005osgi-sp-overview.pdf`

[35] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, 361–376, Boston, December 2002.

[36] J. S. Plank, M. Beck, and G. Kingsley. Libckpt: Transparent Checkpointing under Unix. *Proc. USENIX Winter 1995 Technical Conference*, 220–232, New Orleans, January 1995.

[37] Qumranet Inc. KVM: Kernel-based Virtual Machine for Linux. `http://kvm.qumranet.com/kvmwiki`

[38] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *Proc. 2nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, 31–39, Palo Alto, October 1987.

[39] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, 38(5), 39–47, May 2005.

[40] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: A Quasi-Static Compiler for Java. *Proc. 15th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, 66–82, Minneapolis, October 2000.

[41] T. Smits. Unbreakable Java: The Java Server that Never Goes Down, SAP AG, 2004. `https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/library/webas/Unbreakable%20Java.pdf`

[42] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. *Proc. EuroSys 2007*, 275–288, Lisbon, March 2007.

[43] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. *Proc. 16th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, 180–194, Tampa, October 2001.

[44] Sun Microsystems. Class Data Sharing, 2004. `http://java.sun.com/j2se/1.5.0/docs/guide/vm/class-data-sharing.html`

[45] Sun Microsystems. Java 2 Platform, Enterprise Edition (J2EE) Overview. `http://java.sun.com/j2ee/overview.html`

[46] P. Tullmann, M. Hibler, and J. Lepreau. Janos: A Java-Oriented OS for Active Network Nodes. *IEEE Journal on Selected Areas in Communications*, 19(3), 501–510, March 2001.

[47] VMware Inc. Using the Snapshot. *VMware Workstation 4 User's Manual*. `http://www.vmware.com/pdf/ws40_manual.pdf`

[48] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, 148–162, Brighton, October 2005.

[49] World Wide Web Consortium. Jigsaw – W3C's Server. `http://www.w3.org/Jigsaw/`

[50] World Wide Web Consortium. XML Schema. `http://www.w3.org/XML/Schema`