Interaction of Virtual Machine with the Operating System

Kiran Tati and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{ktati, voelker}@cs.ucsd.edu

Abstract

As applications executing in virtual machine runtime environments such as the Java Virtual Machine and the .NET Common Language Runtime become more prevalent on desktop operating systems, this trend raises the interesting question of the best model for supporting multiple applications in virtual machine environments. A natural model is to execute each application in its own virtual machine process. However, multiple applications could also be executed in a single virtual machine process by leveraging the protection and security features of the programming languages used by the applications. Executing within a single virtual machine process makes it easier to share code and data and communicate among the processes.

In this paper, we present an intermediate model for supporting multiple applications in virtual machine environments for desktop operating systems. We argue that applications should take advantage of the protection features and resource management provided by the operating system and execute each application in its own virtual machine process. However, to facilitate code reuse, reduce application initialization time, and facilitate interprocess communication among applications in different virtual machines, we propose extending virtual machine implementations with the use of a shared class cache and an efficient serialization implementation optimized for local machine interprocess communication. To demonstrate and evaluate our approach, we describe the design and implementation of a shared class cache and optimized class serialization in the Java virtual machine, and evaluate our implementation using a set of application and micro-benchmarks.

1 Introduction

The widespread adoption and use of the World Wide Web has spawned a new class of Internet applications on client-server systems, from email to e-commerce to down-loadable applets. To facilitate development and deployment, new programming languages such as Java and C# and runtime environments such as the Java Virtual Machine (JVM) and the .NET common language runtime (CLR) environment have been created to support the portability, security, and rapid development requirements of these applications. Both Java and .NET are now widely used to implement Web applications on servers, and are increasingly being used for desktop applications on clients as well. In particular, as Microsoft transitions more of its applications to the .NET CLR, executing applications in a managed runtime environment, or virtual machine, will become the norm rather than the exception on client-side desktop operating systems. As an extreme, if Microsoft adopts the .NET CLR for all of its applications, then nearly all applications executed on desktop computers will execute within a virtual machine.

Extensive use of virtual machines for executing applications on desktop operating systems raises the interesting question of the best model for supporting those applications in virtual machine environments. A natural model is to execute each application in

its own private virtual machine and operating system process. This model requires no additional functionality, and leverages the existing protection and resource management mechanisms of the operating system to isolate applications and manage resources across applications. In contrast, a different model is to execute many or all applications in a single virtual machine process [3, 4, 5, 6, 8, 9]. In this model, the protection and security provided by language features are used to isolate and protect applications within the same process address space. Once applications share a virtual machine process, they can also potentially share code (intermediate and/or native code) to save memory and class initialization time [4] and more efficiently communicate with each other, e.g., via regions supporting shared objects [5]. However, sharing a single virtual machine makes it awkward to incorporate native code [11], and leaves applications vulnerable to bugs in the virtual machine protection and security implementation [?].

In this paper, we present an intermediate approach for supporting multiple applications in virtual machine environments for desktop operating systems. We argue that applications should take advantage of the protection features and resource management provided by the operating system and execute each application in its own virtual machine process. However, to facilitate code reuse, reduce application initialization time, and facilitate interprocess communication among applications in different virtual machines, we propose extending virtual machine implementations with the use of a shared class cache and an efficient serialization implementation optimized for local machine interprocess communication. To demonstrate and evaluate our approach, we describe the design and implementation of a shared class cache and optimized class serialization in the Java virtual machine, and evaluate our implementation using a set of application and micro-benchmarks. From our evaluation, we find that common Java applications share up to 75% of their classes, and show that a shared class cache can reduce application initialization time by up to 72–90%. We also show that a class serialization implementation optimized for local communication can reduce communication costs by factors of 2.5–12.

The rest of this paper is organized as follows. Section 2 discusses the tradeoffs between executing multiple applications in single vs. multiple virtual machine environments. We then argue for an intermediate approach that combines the advantages of both models using class sharing and an efficient implementation of serialization for low-overhead interprocess communication. Section 3 then describes our class sharing mechanism in detail, and demonstrates its potential for improving class loading times. And Section 4 describes our serialization mechanism and evaluates its effectiveness at improving interprocess communication. Finally, Section ?? summarizes our model and results.

2 Single vs. Multiple VMs

There are two models for executing multiple Java applications. The first model is the *single VM model* where all applications execute in a single virtual machine [3, 4, 5, 6]. The second model is the *multiple VM model* where each application executes in a separate virtual machine [7, 8, 9].

The single VM model has better resource utilization than the multiple VM model [4], thereby reducing the per-application overhead [5]. The single VM model reduces application startup time, conserves memory, and more easily provides efficient interprocess communication [5]. The single VM model also provides multitasking without underlying operating system support, thus it could be used to execute applications on small devices such as PDAs and cell phones that do not have a multitasking operating system.

On the other hand, the multiple VM model provides strong protection for isolating applications and provides an efficient environment to execute native code [1]. The single VM model can be extended to a multiple process model to provide a safe environment for executing native code, albeit awkwardly and inefficiently [11]. As a result, the single VM model requires replication of some of the operating system functionality at user level [4, 5, 6] to provide strong

[1] Native code is code that is written in a language other than a type-safe language.

application isolation. This duplication of functionality unnecessarily adds extra cost to application execution time when compared to the multiple VM model.

We argue that the best approach for executing multiple applications in a virtual machine environment on a general purpose operating system is to run each application in a separate virtual machine process as in the multiple VM model, and extend the virtual machine to incorporate features that provide the advantages of the single VM model in a multiple process implementation. To motivate this intermediate approach, the following sections describe issues with the single VM model, their performance implications, and how the multiple VM model can efficiently address them.

2.1 Native Code

Native code is code written in a language other than a type-safe language, and is usually written in traditional unsafe languages such as C and C++. In the Java Virtual Machine, native code can access any data in the process in which it is executing, whereas Java byte code is limited in addressability by the strong type-safe properties of the language. Thus, executing the native code of an application can be harmful for other applications in the single VM model because the native code can access and modify any data of the other applications that are running in the same process. One solution to this problem is to execute the native code only if it is trusted code, otherwise terminate the application; this solution is used in the safe mode of the .NET environment. However, running untrusted code is still a problem for the single VM model.

A solution for executing untrusted code in the single VM model is to use another process to run the untrusted code [11]. In contrast, the multiple VM model runs the untrusted code and application in the same process. In the multiple VM model, the untrusted code can at worst harm itself because it cannot access data in other applications as each application has its own address space. Hence, this solution uses the same mechanism, separate address spaces, as the multiple VM model to execute untrusted native code.

Interprocess communication is used to communicate between the application process and the process running native code in the single VM model. Interprocess communication is very costly because of the dramatic increase in the number of switches and the communication overhead. The overall effect of this overhead is an 8% - 880% increase in the execution time for the JVMSpec98 benchmark applications in comparison with the multiple VM model [11]. In summary, the single VM model ultimately uses the same mechanism, the separate process address space, as the multiple VM model to execute the native code safely, but with high overhead.

2.2 Unintended Resource Sharing

Even though the language type-safety prevents an application from accessing the data in other applications, a malicious or buggy application can easily degrade another application's progress by allocating all available memory or using all available file handles, network handles or, in general, consuming any operating system resource entirely in the single VM model. Because of these safety issues, the single VM model requires rigid resource management [5, 6, 12].

KaffeOS [5] and J-kernel [6, 12] provide such rigid resource control mechanisms at the user level. However, the solution essentially re-implements the process abstraction at user level [5]. Again, the multiple VM model does not have these safety issues because each application executes in its own address space and an application cannot access another application's data. The overall overhead of this user level process abstraction is up to a 7% increase to the application execution times for JVMSpec98 benchmark programs in comparison with the multiple VM model [5].

In a multi-user operating system, process ownership is used to control resource management. In the single VM model, process ownership cannot be used to manage operating system resources because all applications are executing in a single process. The multiple VM model does not have this problem. Any conceivable solution to this problem in the single VM model requires the notion of ownership at user level which is essentially re-implementing the operating system functionality at user level. Thus, any solution to this problem will likely add overhead to the application execution time.

2.3 User Level Threads

User level threads pose another problem for the single VM model. Some operating systems do not support threads in the kernel. Hence, the virtual machine has to implement threads at the user level. In this situation, the single VM model is very inefficient because of the mismatch between this model's notion of concurrency, threads in a single process, and the operating system's notion of concurrency, processes. In effect, the single VM model cannot express the concurrency well and thereby pays a significant penalty. For example, in this situation a single page fault stops all the applications from executing. In general, any blocking operation such as reading/writing file, accessing data from network, accessing data from any other blocking devices by an application stops all other applications. On the other hand, the multiple VM model notion of application matches the operating system notion of application. Thus, the multiple VM model efficiently exploits the concurrency among different applications to improve throughput [7].

2.4 Static variable Semantics

A virtual machine stores all the information about a class in a class structure. This information includes static variables that are defined in the class. In the single VM model, all applications share these class structures, thereby sharing static variables among applications. Although this sharing saves memory, it violates the semantics of static variables that should guarantee that each application has its own copy of static variables. Because of this sharing, some applications may produce incorrect results if the application correctness depends on correct initial values of static variables. Hence, the single VM model has to eliminate this sharing.

Solutions to this problem are to use a separate class loader for each application [3], not to share classes that have static variables [5], and providing a separate copy of static variables for each application by modifying the virtual machine [4].

The class loader is a mechanism that provides multiple name spaces for class types in a single virtual machine. Hence, a user-defined class loader [10] can be used to provide a separate copy of static variables because all classes loaded by this class loader are separate from other classes in the virtual machine. This mechanism is used in web browsers to execute applets. This solves the problem of providing separate copies of static variables. However, it eliminates advantages such as a reduction in application startup time and memory savings.

Another solution that changes the virtual machine is to provide a separate copy to each application. Unfortunately, this approach increases the application execution time 7% - 70% for JVMSpec98 benchmark applications [3, 4]. The multiple VM model need not worry about this issue because each application has its own copy of static variables. Again, the single VM model duplicates operating system functionality at the user level, thereby increasing the application execution time.

Overall, the single VM model either has to lose functionality or performance to execute untrusted code, and it is not suitable for environments where the underlying operating system does not support threads in the kernel. The single VM model is also not suitable for multi-user environment. Hence, the single VM model is not suitable to run multiple applications on a general purpose operating system even though it reduces the application startup time, shares resources at runtime, and provides efficient interprocess communication. A better approach is to use the multiple VM model and try to achieve the single VM model advantages in the multiple VM model. Note that these advantages are not inherent to the single VM model, and the following sections describe various techniques to achieve the single VM model advantages in the multiple VM model.

3 Class Sharing

The degree of class sharing among applications is the fraction of classes that are common among two or more simultaneously running applications. In this section we show that the degree of class sharing is high for a set of common Java applications, and that ignoring this class sharing property has significant impact on application startup times. The single VM model has the advantage of reducing application

Program	Description
JavaSound	Simple audio player
gfactory	2D Object plotter
iceMail	A mail client
SwingSets	Swing component browser
JVMSpec98	Applet running this benchmark

Table 1: Description of selected programs

Program	Number of Classes Loaded
JavaSound	1263
gfactory	1586
iceMail	1343
SwingSet2	1650
JVMSpec98	1241

Table 2: Number of classes loaded by programs

startup time because it shares classes among applications, whereas the multiple VM model does not have such an advantage. We then propose a new technique, a shared class cache, to reduce application startup times in the multiple VM model, and demonstrate its effectiveness.

We selected some commonly used Java programs to understand available class sharing. Table 1 presents the description of these programs. We modified the Sun HotSpot virtual machine [13] to collect the information about classes loaded by an application at startup time. This information includes the class name, its loader, and its size in memory. The modified virtual machine is used to run all applications listed in Table 1 without any options to identify the classes they load at startup time. We manually stopped applications once they showed the initial window because we are interested only in classes that are loaded at the application startup time. Table 2 shows the number of classes loaded by each application.

The degree of class sharing for applications x, y is measured as the number of classes y loads that are already loaded by x. The degree of class sharing measures the number of application classes that are already loaded by previous applications. For a given number of applications, average class sharing is measured as the average of class sharing for all permutations of applications.

Table 3 presents average class sharing for the applications listed in Table 1. As expected, the average class sharing increases with the number of already loaded applications. The main reason for this is that all the programs use standard libraries. For these Java programs, on average 77% of an application's

Number of already loaded programs	Already loaded classes	Already loaded classes in percentage
1	882	62.3
2	1035	73.1
3	1075	75.9
4	1094	77.3

Table 3: Available sharing

classes are already loaded if all of the four applications are executed before this application. The class sharing in this environment is significantly higher than the available class sharing in the web browser environment [27].

3.1 The Experimental Environment

Table 4 presents the machine configuration and the software used for the following experiments. All the timings presented in this paper are average of ten runs and an extra five runs are used to warmup the cache.

3.2 Class Loading Time and its Impact on Application Startup Time

We define application startup time as the time difference between application start time and the time

OS	Windows 2000 5.00.2195 Service Pack 1
CPU	Intel Pentium 4, 1.6GHz
RAM	512MB
JVM	Sun Server HotSpot Virtual Machine 2.0
JDK	Sun JDK 1.3.01

Table 4: Machine Specification

it is ready to take the input from user. This startup time includes the process creation time, virtual machine initialization time, class loading and initialization times, and the main thread execution time to draw the main window completely. The startup time represents the user wait-time before the user can interact with the application. All the selected programs have GUIs and these applications do not accept any input until they draw the initial window.

The Sun HotSpot virtual machine is implemented as a shared library and a startup program is used to load this library. This startup program also initializes the JVM, loads the application main class, and then passes control to the application main method. This initializing thread acts as the main thread for the application. Usually GUI components create other threads to get user input. We modified the startup program to exit after the main thread finishes its execution to measure the application startup time.

In order to measure the class loading times, we implemented a Java program that takes class names that are loaded by the application as input and measures the time to load these classes. Using this program, we measured the class loading time for applications listed in Table 1. Figure 1 shows the results of these class loading times and application startup times.

Each bar in the figure represents the overall application startup time and the lower part of the bar indicates the class loading time. Each application is associated with two bars in the figure. The first bar is the application startup time when class files are *not* in the file cache, and the second bar is the application startup time when class files are in the file cache. The application startup times are very high for these applications even on a fast machine, and these times are high even when class files are in the file cache. Furthermore, the class loading time is a major component of application startup time when class files are not in the file cache as it constitutes 31% - 65% of application startup time.

We know that a significant number of application classes are already loaded by previous applications from the class sharing experiments. Thus, one could exploit this available class sharing to reduce application startup time. In fact, the single VM model very

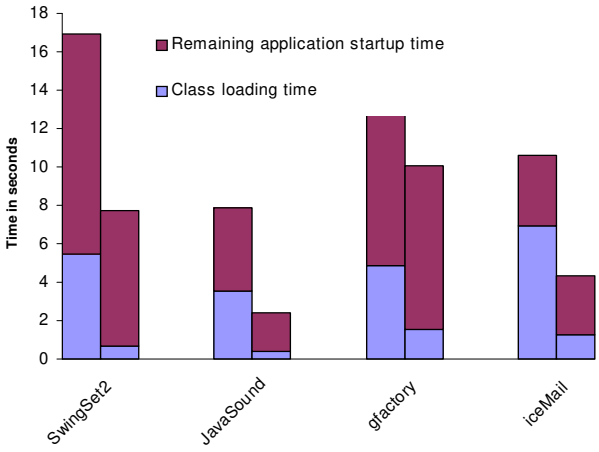

Figure 1: Class loading time impact on application startup time

efficiently eliminates the class loading time because it shares classes among all applications and thereby improves the application startup times. However, the single VM model increases application total execution time because of static variable issue discussed in Section 2.4. File caching is the only mechanism that can take advantage of class sharing among applications in the multiple VM model. The standard code sharing facilities offered by the operating system, shared libraries, cannot take advantage of the class sharing because the class file format is not in the executable format that is recognized by the operating system to share code across processes. Issues in sharing dynamic code generated by JITs are discussed in the related section.

The virtual machine initialization costs can be eliminated by forking an existing Java application process [7]. The virtual machine initialization time is one of the components in the application startup time. In the Java virtual machine, this encompasses process creation time, the interpreter generation, initializing the virtual machine data structures, and loading approximately 250 standard classes used by the virtual machine. Here onwards this optimization is referred to as the *IBM optimization*. This optimization eliminates approximately 1.5 seconds from

the application startup time when class files are not in the file cache, and 100 milliseconds when class files are in the file cache. This optimization saves 10% - 20% of the application startup time when class files are not in cache, and 1% - 5% when all class files are in file cache for the above mentioned applications. This optimization certainly reduces the application startup time. However, there are still a significant number of application-specific classes (80% - 83% classes for applications mentioned in Table 1) that need to be loaded at application startup time and this optimization cannot eliminate this class loading time overhead.

From Figure 1, we see that the file cache reduces 68% - 90% of the class loading time in comparison with the case when files are not in the file cache because of costly I/O operations. This shows that some form of caching is effective at reducing the application startup time. However, the file cache may not retain these classes because they can be flushed away by other file data [25]. Class files are very valuable to reduce the application startup times and they should get higher priority than ordinary files in the file cache. Operating systems put executable files in the virtual memory cache, which is separate from the file cache. Hence, executable files get higher priority than the ordinary files in the traditional operating system. However, class files do not get such a privilege in the multiple VM model because they are not part of the virtual memory cache. The following subsection describes a user level cache mechanism, shared class caching, that improves application startup time.

Application startup Component	**Single VM Model Optimization**	**Multiple VM model Optimization**
Virtual Machine initialization cost	Yes	IBM Optimization
Class loading Times	Yes	Class caching
Class initialization Time	No	No

Table 5: Application startup time components and optimizations that eliminate or reduce these costs in both the single VM and multiple VM models

3.3 Class Cache

One solution to minimize the application startup time is to create a user level cache to store application classes using the shared memory mechanism supported by the operating system. The virtual machine maps the shared region into its address space as part of its initialization. This class caching puts the class code into the virtual memory cache. As a result, these pages get the same priority as traditional code pages.

In JVMs, classes are stored in the heap as a java object. The JVM constructs this class object when it needs to load a new class. The class loading involves reading the class file, parsing the class file, verifying the class, and initializing static variables. Ideally, one would like to store these class objects in the shared region so that all other applications can share these classes and get the same benefits as the single VM model provides for class sharing. However, this approach has the same problems, such as static variable issues, as in the single VM model.

In our approach, the JVM stores class summary objects in the class cache that is similar to the JVM class object. The class loading process is modified to look in the class cache first for the class whenever it needs to load a new class. If the required class is in the class cache, it constructs the class object from the information stored in the class cache. If the class cache does not have the required class, the JVM uses the regular method to load the necessary class. If the JVM loads the class successfully, it also constructs the summary object and stores the summary object in the class cache. The JVM also skips the verification phase whenever it loads a class from the class cache because the class is already verified when it enters the class cache. The class summary object has all the information as in the class file and extra information that is computed as part of class loading. This extra information includes the *itable* that is used to speedup the interface method lookups, the *vtable*

that is used to speedup the class method lookups, and the *oop maps* that are used at garbage collection time to identify object pointers in objects of this class.

Our approach has two advantages over just relying upon the file system cache. First, it saves space in the shared region because the class structure is smaller than the class file. Second, it speeds up the class parsing time.

Our approach saved most of the class loading time as in the single VM model except the class initialization time that initializes static variables. However, this class initialization time cannot be eliminated even in the single VM model. The main reason for this is that the class initialization can execute any arbitrary user code. Therefore, initial values of static variables depend on the entire virtual machine state. Hence, initial values of static variables are not cachable. Our approach uses more memory than the single VM approach to store the class summary objects, but in the desktop environment main memory is abundant. Hence, trading memory for performance is justifiable. Table 5 summarizes the various components of application startup time and optimizations that eliminate or reduce these component costs in both single and multiple VM models.

The class cache has a finite amount of memory and simple LRU is used as a replacement policy. Cache coherence is implemented using the file modification event mechanism supported by many operating systems, including Windows 2000 and FreeBSD. An extra process is used to maintain the class cache. This extra process is used to implement the IBM Optimization to fork from a clean process running a bare virtual machine. Here onwards this process is referred to as the skeleton process. Any virtual machine process can load the class into the class cache and it also informs the skeleton process about the loaded class. The skeleton process registers with the operating system to get the notification events whenever any of the cached files are modified. The skeleton process removes the modified class and all of its subclasses from the cache whenever it gets the modification event for a class. The skeleton process has to remove all subclasses of the modified class because the *itable* and *vtable* of subclasses depend on the modified class. If the operating system does not support the file modification event then we could store the file modification time in the summary object and this file modification time can be used to validate staleness of the class cache summary object.

3.3.1 Implementation status

We implemented this shared memory based class caching in the Sun HotSpot server virtual machine 2.0 on the Windows 2000 operating system. Most of the above design is implemented except for the modification event mechanism. In the current implementation only *itables* and *vtables* sizes are saved.

3.4 Results

The programs in Table 1 are used to show the efficacy of class caching on micro-benchmarks that load only application classes and on application startup times. Figure 2 shows the class loading time for programs in Table 1. Each bar represents the application class loading time. The first, second, third, and fourth bars show the application class loading time when class files are not in the file cache, when class files are in the file cache, when class files are in the file cache but not in the class cache, and when class files are loaded from the class cache, respectively. The class loading time when class files are in the class cache and not in the file cache are shown again in Figure 2 for completeness. We did not show the class loading costs for the single VM model because it eliminates all of these costs. The class cache improves class loading times 72% - 90% compared with the case when class files are not in the file cache and it improves 4% - 15% compared with the case when class files are in file cache for the applications in Table 1. As expected, the class cache improves the class loading costs dramatically compared to the case when files are not in the cache. Furthermore, the class cache improves class loading time even compared to the file cache.

Figure 3 shows the results of the application startup time experiments. Each bar in this figure shows the application startup time after applying the IBM optimization. The first, second, third, and fourth bars show the application startup time when class files are not in the file cache, when class files are

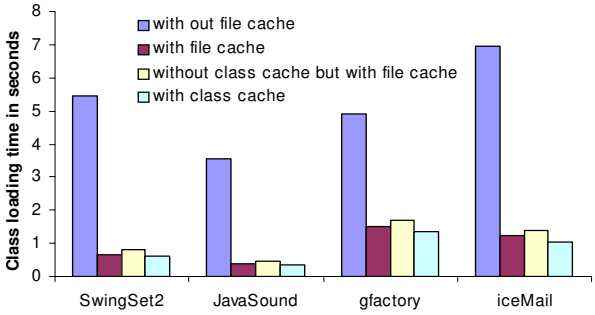

Figure 2: Application class loading times

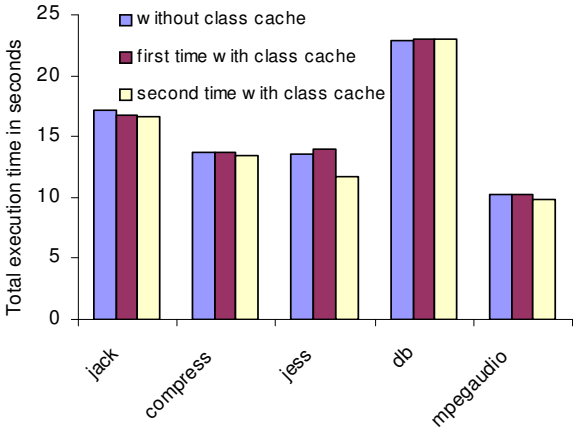

Figure 4: Execution times for JVMSpec98 benchmark programs

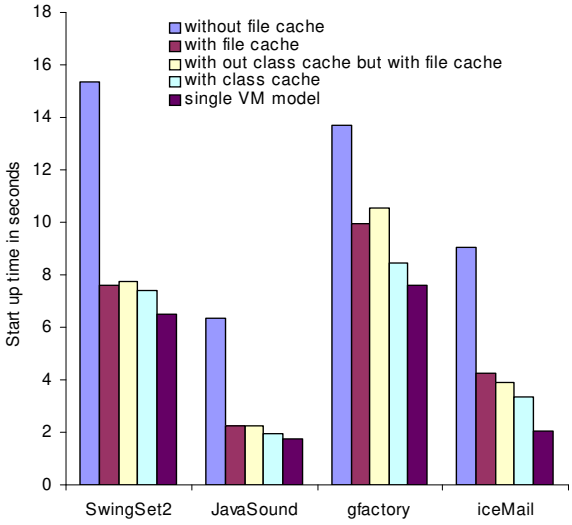

Figure 3: Application startup times

in the file cache, when class files are in the file cache but not in the class cache, and when class files are in the class cache, respectively. The fifth bar shows the application startup time in the single VM model. Class files are loaded into the virtual machine and then the application main thread is executed to measure its startup time in the single VM model. The application startup time in the single VM model is a conservative estimate only. This is a conservative estimate because the single VM model eliminates the class verification costs and we could not eliminate these verification costs in our approach.

The overhead of class copying into the class cache is 0.3% - 6% compared with the case when class files are in the file cache and, as the figure shows, this overhead very small. This cost is a one time cost and it is amortized over all applications that load this class.

The class cache reduces application startup time 38% - 69% compared with the case when class files are not in the file cache. More importantly, this class cache improves application startup time 2% - 21% compared with the case when class files are in file cache. We are comparing our approach with the file cache because the file cache time gives the best application startup time that could be achieved using existing mechanisms. As previously mentioned, this

time is the most optimistic and may not always be achieved. Our approach always performs better in comparison with the file cache, the most optimistic case. However, class caching is 9% - 38% more costly than the single VM model. As the figure shows, the application startup times using class caching in the multiple VM model is within the order of application startup time in the single VM model.

We used the JVMSpec98 benchmark programs with the class cache to evaluate the performance implications of the class cache on long running programs. Figure 4 shows the total execution time for five of the spec98 benchmark programs. We ran each spec98 benchmark program (1) without the class cache (the first bar for each program shows this execution time), (2) with an empty class cache (the second bar), and (3) with all the required classes in the lass cache (third bar). The JVMSpec98 benchmark programs load very few classes (around 300–400). As expected, the impact of class caching is very little (less than 1% improvement over the base case) and overhead of loading class summary into the class cache is also very little (less than 1%).

4 Inter Process Communication

The single VM model claims to have the most efficient interprocess communication among application because it allows applications to pass object references among themselves [5]. However, in practice this is not always possible because of data coherence issues. Most of the applications on the desktop are developed independently. Hence, passing an object reference to another application may lead to a state where both applications trying to access the object at the same time may corrupt object data, thereby hampering the correctness of both applications.

One solution is to make all methods of a class synchronized if there is a possibility of passing an object of the class to another application. This solution is not feasible because one has to implement synchronization on all methods of a serializable class, which decreases the application performance unnecessarily and is not possible for older applications that did not have these synchronized methods implemented already.

Thus, even in the single VM model, independently developed applications require some form of copying to pass objects [4]. The easiest way is to use the Java serialization. However, the single VM model can implement a more efficient serialization mechanism.

There are two mechanisms for copying objects in the Java environment and they are Java serialization and cloning. Java serialization serializes an object and all objects that can be accessed from the object into a stream, and the same mechanism can be used to reconstruct an object from the stream. Cloning is another mechanism to copy objects within one JVM. Cloning simply copies the object but does not copy the all objects that are accessible from the given object. For simple array types that do not refer to any other objects, cloning and serialization are the same in terms of functionality.

In this section, we compare the Java standard serialization with cloning for copying simple array objects to show that serialization is very inefficient for local communication. The simple array type objects are the easiest objects to serialize and serialization needs to do at least the same amount of work for serializing any other object types. Hence, serialization efficiency results for the simple array type are also applicable to other object types. Comparing the serialization cost with the cloning cost for simple array types estimates the amount of overhead of serialization. We emphasize again that, even in the single VM model, cloning is not useful for copying objects that have references to other objects because it does not perform a deep copy and it does not provide any data protection. On the other hand, serialization is designed to copy objects securely across applications. Hence, serialization has to be used to copy objects between applications in both single and multiple VM models.

Table 6 shows the time to copy a simple integer array using various copying mechanisms. All the times in the table are microseconds. The integer array size is varied from 1 to 256KB. The second column shows copying costs for cloning and the third column shows the copying costs for standard serialization with a file as a stream buffer. Because using a file introduces

size	cloning	Serialization using a File	Serialization using byte array
1	0.010	0.282	0.122
2	0.010	0.280	0.114
4	0.010	0.300	0.122
8	0.008	0.292	0.117
16	0.010	0.329	0.127
32	0.010	0.308	0.149
64	0.009	0.336	0.164
128	0.010	0.395	0.224
256	0.013	0.336	0.137
512	0.018	0.354	0.157
1024	0.018	0.438	0.212
2048	0.028	1.137	0.297
4096	0.053	2.405	0.489
8192	0.083	2.855	1.472
16384	0.653	5.553	1.605
32768	0.899	13.73	6.330
65536	1.237	24.47	6.612
131072	7.058	50.11	15.33
262144	14.57	90.57	24.15

Table 6: Serialization costs for simple Int Arrays (times are in microseconds)

overhead, we also implemented a simple byte-array-based stream buffer to eliminate operating system interference. The fourth column shows copying costs for standard serialization with the byte-array-based stream buffer. Cloning is one operation to create a new copy of an object in the same process whereas serialization requires two operations, namely *readObject, writeObject*, to do the same. All timings for serialization in Table 6 are the combined cost of *readObject* and *writeObject* operations.

As expected, serialization using the file performed the worst of all three mechanisms. However, cloning performed an order of magnitude better than serialization. Cloning performed 6 - 41 times better than the standard serialization with the file as buffer, and 1.6 - 22 times better than the standard serialization with byte array as a buffer. This shows the need for efficient copying mechanisms in both single and multiple VM models.

Much of previous research approached this problem in the remote method invocation (RMI) context because serialization is a major overhead in RMI. A nice summary of the previous work in this area is presented in [15]. The major overhead in serialization is copying costs to copy the data among different buffers and the usage of costly class introspection primitives to get object type information and data. For example, the standard serialization copies the data three to four times among different buffers to serialize a simple array type depending on the stream buffer. Not surprisingly, most of improved implementations [15, 16] used their own buffering mechanisms to reduce copying costs.

The class introspection cost can be avoided by using customized marshaling code to marshal the object data [16]. However, this requires support from the compiler to generate the marshaling code for each serializable class. Hence, this optimization is not feasible in practice because existing applications do not have this code built-in and re-compilation is not always possible because their source code may not available for many applications.

The Java virtual machine (JVM) first copies the simple type array into the C-heap area and then it returns the address of this copy to the native code to access simple array type elements. This extra copy is eliminated if the JVM simply returns the start address of the simple type array rather than the copied array address [15]. Hence, this approach reduces copying costs very effectively for basic type arrays. It thereby reduces the serialization costs dramatically for them.

However, this approach interacts very poorly with garbage collection that moves objects. It has to delay the garbage collection until it finishes copying the array to another buffer because garbage collection may move the Java object that invalidates the address returned by the JVM for an array. At present, most JVMs use stop-the-world garbage collection and this occurs only at certain points. Hence, one can delay the garbage collection in the current JVMs. However, concurrent garbage collection poses a serious threat to the correctness of this approach because garbage collection can happen at any time. Note that .NET

uses concurrent garbage collection and soon JVMs are also going to implement concurrent garbage collection to minimize the garbage collection cost [26]. Hence, this approach is not applicable in the virtual machines that use concurrent garbage collection even though it minimizes the copying costs for simple array types.

The type information returned by class introspection methods can be cached to reduce class introspection costs. However, the Sun JDK1.3 also uses these optimizations in its serialization. Thus, we just simply compare our approach with the Sun JDK1.3 serialization implementation to evaluate the merits of our approach.

4.1 Serialization

We argue that the best way to minimize copying and class introspection costs is to implement serialization in the virtual machine where all the needed information to serialize an object is accessible cheaply. Class introspection costs are minimized because the virtual machine maintains information about class fields in a very compact way and we can use this information directly if the serialization is implemented in the virtual machine. Buffering costs are also minimized because objects can be accessed directly in the virtual machine without going through the native interface that requires an extra copy. The virtual machine also provides handles, an indirect reference to objects, for objects that are preserved during garbage collection. Hence, all the issues of incorrectness due to garbage collection are eliminated by using these handles. Note that this is not possible if the serialization is implemented in native code. Salient features of our serialization design are described in the following paragraphs.

Shared memory is the most efficient interprocess communication method for passing data between applications on a single machine because this eliminates operating system interference once the shared memory region is established. Hence, we use shared memory to transfer the data between applications. Our implementation uses shared memory as a simple queue with lock primitives to synchronize the reader and the writer processes. The write pointer points to the free location in the buffer and the read pointer points to the next available data position in the buffer.

Serialization usually uses a small buffer to serialize small objects in order to minimize the frequent usage of relatively high-cost interprocess communication primitives. Shared memory is also costly for passing small data frequently because of synchronization. We avoided these buffering and synchronization costs by using shared memory as a temporary buffer. The shared memory is accessed by two processes only and in each of these processes only one thread is able to access the shared memory because of Java synchronized methods. A thread reserves a small portion and uses that as a temporary buffer whenever it needs to copy small data. The important point is that this reservation does not move the read/write pointer so that the other process does not access the temporary buffer until this thread releases the buffer. Since the thread has exclusive access to the temporary buffer, it does not require any synchronization while it accesses the buffer. As a result, it minimizes the cost of synchronization. It also minimizes copying costs because only one copy is needed. The thread releases the temporary buffer once it fills the temporary buffer or it completes the serialization. This release also moves the read/write pointer so that the other process can access the data as a big chunk of data.

Serialization first determines the serializable fields of an object and then starts serializing those fields. The JVM primitive to access field attributes is costly and required every time the object is serialized. We minimized these costs by caching the identity of serializable fields of a class. We create a small array that has information about serializable fields of a class when we first serialize an object of the class. This array is stored in the class structure to use in the subsequent serialization of an object of this class.

Though we focused on efficient serialization for the multiple VM model, our approach also reduce serialization costs in the single VM model. Our serialization is amenable to further improvements in the single VM model, and we describe one of them here. One copy is required in the multiple VM model to transfer data from one process to another. The only way to eliminate this copy is to somehow share an application's garbage collected heap space so that one

process can create Java objects in another process and write directly into those new objects. However, this sharing of garbage collected heap space leads to security problems that are similar to the native code problems in the single VM model. However, in the single VM model, all applications share the garbage collected heaps and the native code has access to create and write into the other application's garbage collected heap. Thus, our serialization can easily be modified to remove this one copy required in the multiple VM model to improve the serialization for the single VM model.

4.2 Implementation status

We implemented the serialization optimization and the required classes to use it. The shared memory based stream buffers are also implemented. At present, the shared memory stream is using the mutex primitives provided by the JVM; ideally, we would want to use the operating system provided synchronization primitives directly. In our experiments, the reader and writer threads are running in a single JVM. Thus, these synchronization primitives are enough for the correctness. We still need to implement the support for the serializable and externalizable interfaces that are used to customize the serialization process.

JVM allocation primitives always return zero-filled memory. This initialization could be eliminated in reading the simple array types because we know that this entire array is going to be initialized by serialization with correct values immediately. We did not implement this optimization in the current implementation because it required significant changes to the allocation primitives even though this optimization clearly improves the read serialization cost.

4.3 Results

In this section, we compare our serialization optimization to the standard JVM serialization with file and shared memory as stream buffers. The file buffer is representative of the case where the operating system is involved in the communication and shared memory is representative of the case where operating system is not involved in communication. By comparing results for these two buffering mechanisms

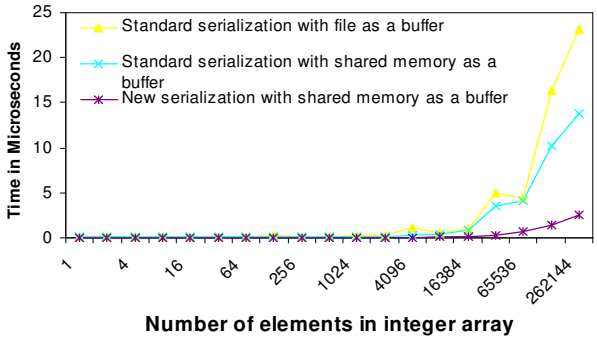

Figure 5: Serialization cost to read simple Integer array

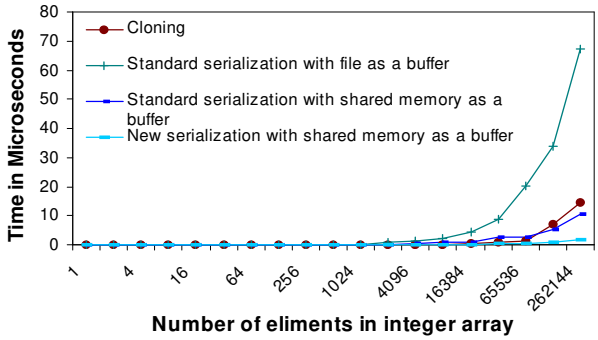

Figure 6: Serialization cost to write simple Integer array

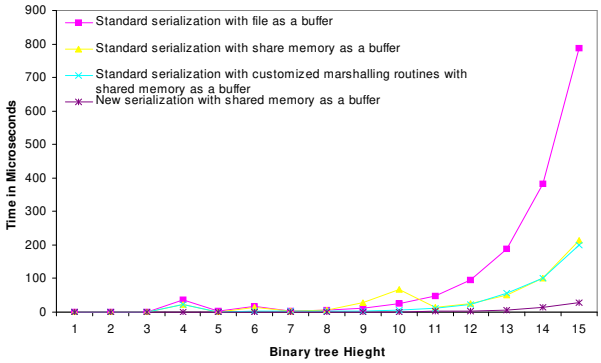

Figure 7: Serialization cost to read binary Tree

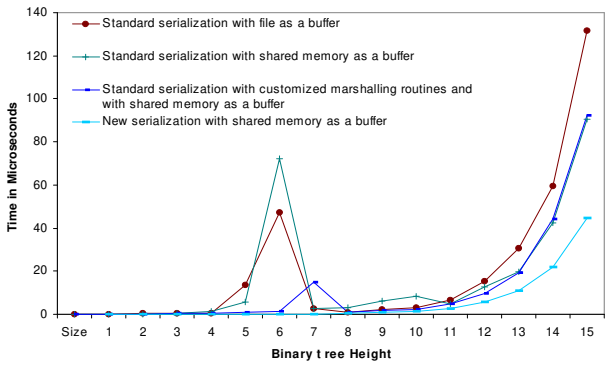

Figure 8: Serialization cost to write binary Tree

we show that communication that requires operating system involvement performs very poorly. This validates our choice of buffering mechanism, shared memory. We chose simple a type array and binary tree to evaluate various serialization mechanisms. These two types represent simple and complex object types. Since these two types are two ends of the spectrum, we claim that results for other type of objects are also similar to the results presented in this section.

Figure 5 shows serialization costs to read a simple integer array from the stream. As expected, standard serialization with shared memory buffer performs an order of magnitude better than the standard serialization with file buffer. The main reason for this is that file buffer requires one extra copy and system call overhead. The new serialization performed extremely well, and it reduces the serialization costs by 6 - 21 times in comparison with standard serialization with a file buffer as a stream buffer. It also reduces the serialization costs by 2.5 - 12.5 times compared with the standard serialization using shared memory as a stream buffer.

Figure 6 shows the serialization costs for writing a simple integer array into the stream. These results are very similar to the serialization costs for reading objects. We also compared our serialization with cloning because it is the best copying mechanism that is available in the single VM model for basic type arrays. We assumed the read cost for cloning is zero because cloning is one operation and we characterize the entire cloning cost as a writing cost. One interesting point is that our combined read and write serialization costs performed better than cloning for larger integer arrays. In cloning, the JVM marks cards that are used in garbage collection once it copied the array irrespective of the array type. This operation added additional overhead to cloning, and our serialization eliminates this cost for basic type arrays because we know that basic type arrays do not have any object references. As expected, our serialization performed much better than the standard serialization.

Figure 7 and Figure 8 show the read and write serialization costs for a complete binary tree with integers as data in each node. Our optimized serialization performed an order of magnitude better than the standard serialization. In these figures, we also compared our approach with the customized marshaling code using a shared memory based buffering serialization. As explained earlier, customized marshaling routines reduces class introspection costs. However, the customization only marginally improves the cost over the standard serialization.

Our new serialization improves serialization costs an order of magnitude over standard serialization for both types of objects because of better buffering and more efficient primitives to access object data and type information. Since these two object types represent two ends of the spectrum of different kinds of objects, we expect that our serialization will also perform similarly for other types of objects.

5 Conclusions

In this paper, we presented an intermediate model for supporting multiple applications in virtual machine environments for desktop operating systems. We argue that applications should take advantage of the protection features and resource management provided by the operating system and execute each application in its own virtual machine process. However, to facilitate code reuse, reduce application initialization time, and facilitate interprocess communication among applications in different virtual machines, we proposed extending virtual machine imple-

mentations with the use of a shared class cache and an efficient serialization implementation optimized for local machine interprocess communication.

To explore the feasibility of our model, we designed and implemented a shared class cache and optimized class serialization in the Java virtual machine. To demonstrate its effectiveness, we evaluated our implementation using a set of common Java applications and micro-benchmarks. From our evaluation, we find that common Java applications share up to 75% of their classes, and show that a shared class cache can reduce application initialization time by up to 72–90%. We also show that a class serialization implementation optimized for local communication can reduce communication costs by factors of 2.5–12.

We note that our class sharing cache stores class data structure and byte code only. The code generated by JIT compilers is not shared in our current implementation, and the overhead of JIT compilation could be minimized by caching partial results of previous compilations [14]. Another solution to reduce this overhead is to share the compiled code as described in [2]. The code generated by the JIT compiler is very specific to the current execution of the program and classes loaded till that time. Hence, the JIT generated code may not be optimum code for another application. A more detailed study is needed to understand effectiveness of re-usability of JIT generated code.

Finally, we note that Java RMI could be modified to take advantage of our serialization to improve its performance. A similar study showed that 2 - 8 times improvement for small benchmarks [1].

References

[1] K. Palacz, G. Czajkowski, L. Daynes, and J. Vitek, "Incommunicado: Efficient Communication for Isolates", In Proceedings of Object-Oriented Programming, Systems, Languages and Applications, Seattle, WA. (Nov. 2002).

[2] G. Czajkowski, L. Daynes, and N. Nystrom, "Code sharing Among Virtual machines", In ECOOP, Malaga, Spain, (Jun. 2002).

[3] G. Czajkowski, "Application Isolation in the Java Virtual Machine", In Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Minneapolis, MN, (Oct. 2000).

[4] G. Czajkowski, L. Daynes, "Multi-tasking without compromise: a Virtual Machine Approach", In Proceedings of Object-Oriented Programming, Systems, Languages and Applications, Tampa, FL. (Oct. 2001).

[5] G.Back, W.C. Hsieh, and J. Lepreau. "Processes in KaffeOS: Isolation, resource management, and sharing in Java", In Fourth Symposium on Operating Systems Design and Implementation, (Oct. 2000).

[6] G. Czajkowski, C-C. Chang, C. Hawblitzel, D. Hu, T. von Eicken "Resource Management for Extensible Internet Servers", In Eighth ACM SIGOPS European Workshop Support for Composing Distributed Applications Sintra, Portugal, (Sep. 1998).

[7] 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", In IBM Systems Journal, Volume 39, Number 1, 2000.

[8] Dahlia Malkhi, Michael Reiter, Avi Rubin, "Secure Execution of Java Applets using a Remote Playground", In IEEE Symposium on Security and privacy, Oakland, CA, (May 1998).

[9] Trent Jaeger, Jochen Liedtke, and Nayeem Islam, "Operating System Protection for Fine-Grained Programs", In Seventh USENIX Security Symposium, (Jan. 1998).

[10] S. Liang and G. Bracha, "Dynamic class loading in the Java Virtual Machine", In Proceedings of Object-Oriented Programming, Systems Languages and Applications, (Oct. 1998).

[11] G. Czajkowski, L. Daynes, M. Wolczko, "Automated and Portable Native Code Isolation",

In Proceedings of the 12th IEEE International Symposium on Software Reliability Engineering, Hong-Kong, (Nov. 2001).

[12] J. Lepreau, P. Tullmann. "Nested Java processes: OS structure for mobile code", In Eighth ACM SIGOPS European Workshop Support for Composing Distributed Applications Sintra, Portugal, (Sep. 1998).

[13] Michael Paleczny, Christopher Vick, and Cliff Click, "The Java HotSpot Server Compiler", In Java Virtual Machine Research and Technology Symposium, Monterey, California, (Apr. 2001).

[14] Chandra Krintz and Brad Calder, "Using Annotations to Reduce Dynamic Optimization Time", In ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI), (June 2001).

[15] Fabian Breg, Constantine D. Polychronopoulos, "Java Virtual Machine Support for Object Serialization", In Java Grande/ISCOPE 2001 Special Issue of Concurrency and Computation:Practice and Experience.

[16] Michael Philippsen, Bernhard Haumacher. "More Efficient Object Serialization", In Workshop on Java for Parallel and Distributed Computing, (1999).

[17] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau, "Annotating the Java Bytecodes in Support of Optimization", In Journal Concurrency:Practice and Experience, Vol. 9(11), (Nov. 1997).

[18] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge, "A Framework for Optimizing Java Using Attributes." In Sable Technical Report No. 2000-2, (2000)

[19] O. Babaoglu and W. Joy, "Converting a swap-based system to do paging in an architecture lacking page-referenced bits." In Proceedings of the Eight Symposium on Operating Systems Principles, Pacific Grove, CA, (Dec. 1981).

[20] Sun JDK1.4 new IO APIs, `http://java.sun.com/j2se/1.4/docs/guide/nio/inde` (2002).

[21] Tim Lindholm and Frank Yellin, *The Java^{TM} Machine Specification*, second edition, Addison-Wesley Publishers (1999).

[22] Microsoft .NET Environment, `http://www.microsoft.com/net/`

[23] J. C. Brustoloni and P. Steenkiste, "Effects of buffering semantics on I/O performance", In Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation, Seattle WA, (Oct. 1996).

[24] V. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A unified I/O buffering and caching system", In Proc. of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans, LA, (Feb. 1999).

[25] A. Wang, H. Kuenning, P. Reiher, J. Popek. "Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System", In Proc. of the 2002 USENIX Annual Technical Conference, Monterey, (June 2002).

[26] T. Domani and et al., "Implementing an on-the-fly garbage collector for Java", in Proc. of the ACM SIGPLAN Symposium on Memory Management, (2000).

[27] Marie T. Conte, Andrew R. Trick, John C. Gyllenhaal, and Wen-mei W. Hwu, "A Study of Code Reuse and Sharing Characteristics of Java Applications", In Workshop on Workload Characteristics, Micro-31, (Nov. 1998).