Scaling J2EE™ Application Servers with the Multi-Tasking Virtual Machine

Mick Jordan, Laurent Daynès, Grzegorz Czajkowski
Marcin Jarzab, and Ciarán Bryce

Scaling J2EE™ Application Servers with the Multi-Tasking Virtual Machine

Mick Jordan, Laurent Daynès, Grzegorz Czajkowski,
Marcin Jarzab, and Ciarán Bryce

SMLI TR-2004-135 June 2004

Abstract:

The Java 2 Platform, Enterprise Edition (J2EE) is established as the standard platform for hosting enterprise applications written in the Java programming language. Similar to an operating system, a J2EE server can host multiple applications, but this is rarely seen in practice due to limitations on scalability, weak inter-application isolation and inadequate resource management facilities in the underlying Java platform. This leads to a proliferation of server instances, each typically hosting a single application, with a consequent dramatic increase in the total memory footprint and more complex system administration. The Multi-tasking Virtual Machine (MVM) solves this problem by providing an efficient and scalable implementation of the isolate API for multiple, isolated tasks, enabling the co-location of multiple server instances in a single MVM process. Isolates also enable the restructuring of a J2EE server implementation as a collection of isolated components, offering increased flexibility and reliability. The resulting system is a step towards a complete and scalable operating environment for enterprise applications.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email addresses:
mick.jordan@sun.com
laurent.daynès@sun.com
grzegorz.czajkowski@sun.com
mj@agh.edu.pl
ciaran.bryce@cui.unige.ch

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

Scaling J2EE™ Application Servers with the Multi-Tasking Virtual Machine

Mick Jordan, Laurent Daynès, Grzegorz Czajkowski
Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043, USA
firstname.lastname@sun.com

Marcin Jarzab
Dept. of Computer Science
AGH University of Science and Technology
Krakow, Poland
mj@agh.edu.pl

Ciaran Bryce
Object Systems Group
University of Geneva
Switzerland
Ciaran.Bryce@cui.unige.ch

1 INTRODUCTION

The Java™ 2 Platform, Enterprise Edition (J2EE) [Sun03b] is the standard server-side environment for developing enterprise applications in the Java programming language.

J2EE applications are encapsulated in well-defined *enterprise archive files* (ear files) and are deployed and executed on J2EE servers in a similar manner to conventional applications on existing operating systems. Applications are typically composed of several modules that handle specific aspects, notably web modules for interaction and presentation, Enterprise Java Beans™ (EJB) modules for business logic, and resource adapter modules for accessing legacy data systems. These modules are hosted in *containers* that interpose between the application modules and the available services. Containers are themselves instantiated in servers, for example, a Web or EJB server.

The J2EE architecture is specified in such a way that certain containers are defined to be logically separate – for example, the Web container and the EJB container – with communication through well defined interfaces. A particular implementation of the architecture can choose whether to support these containers in separate Java virtual machines (JVM™), possibly on different machines, or in a single JVM. In this way the architecture scales while providing a portable programming model. Large scale systems typically consist of a tier of web servers, a tier of EJB servers and a database tier. Each tier may also be clustered to provide high availability and/or increased throughput.

A single J2EE server supports the execution of multiple concurrent applications using the classloader mechanism of the Java platform [LB98] as the means of separation, and the threading mechanisms as the means of concurrency. The server also manages the virtualization of global, shared resources, such as database connections. A J2EE server thus provides an operating environment for J2EE applications in much the same way that an operating system provides an operating environment for traditional applications.

However, limitations in facilities provided by the underlying Java platform prevent the J2EE operating environment from providing the same quality of service as the process model of a traditional operating system. For example, there is no robust way to terminate a J2EE application as threads cannot be stopped safely. Similarly, the degree of isolation provided by the classloader model is weak compared to that of an operating system process. In addition, the Java platform provides no way to control important resources, such as CPU time, and therefore prevent denial of service attacks. Finally, scalability issues in the JVM itself, for example, garbage collection algorithms with long latencies, can be a limiting factor. Much effort continues to be directed at improving the scalability of a single JVM such that, in the foreseeable future, this issue may disappear. However, the platform limitations will remain unless the appropriate features are added.

The practical consequence of these limitations is that typical J2EE servers host a single application, even on large SMP machines, so that the process-based mechanisms of the underlying operating system can be used to satisfy the requirements of isolation and resource management. Not only are the multi-application facilities of the J2EE server going largely unused, but the resulting increase in the number of server instances is extremely wasteful of memory, as the memory footprint of a

J2EE server is substantial. System administration is also made more complex, as it is inherently harder to manage a system comprising multiple processes. The solution in many J2EE systems is to include a special administrative server, which adds additional footprint and complexity.

One can argue that if a particular J2EE application must support so many clients that it requires a multi-tiered and clustered J2EE installation, then there is evidently no need to run multiple applications in a single server. However, not all applications have such requirements. Furthermore, it is expensive and potentially wasteful to dedicate a large hardware installation to a single application. It would be more flexible and efficient to deploy multiple applications across the cluster on each server and be able to dynamically adjust the resources that are assigned to each application. Currently, this can only be achieved by providing a J2EE server instance for each application on each machine in the cluster.

Our previous research led to the development of a programming model, and an associated API extension to the Java platform, that supports fully isolated computations. The API development, carried out as a Java Specification Request under the Java Community Process, is described by JSR 121 [JCP01]. The isolation API is capable of being realized by a set of co-operating JVMs or by a single, "multi-tasking" JVM. The API thereby provides a portable programming model while ensuring fault isolation between components. Two implementations of this model have been built to date. The first is the reference implementation of JSR 121, that uses multiple operating system processes to represent the isolated computations. The second is the the Multi-tasking Virtual Machine (or MVM) [CD01]. Work is also underway to extend the model to clusters of separate machines.

MVM has demonstrated that co-locating computations in a multi-tasking virtual machine – combined with aggressive, transparent, sharing of runtime data structures – can significantly decrease startup time and memory footprint [CD01], [CDN02]. That work was carried out in the context of the Java 2 Platform, Standard Edition (J2SE™) platform, which targets desktop applications.

This paper addresses the applicability of the isolate programming model and the MVM to the J2EE platform. It shows how the memory footprint of large multi-application systems can be dramatically reduced and how sound inter-application isolation can be achieved. In contrast to previously reported measurements on MVM, which used micro-benchmarks, we apply MVM to the J2EE 1.3.1 Reference Implementation (J2EERI) [Sun03c], reporting comparative performance measurements against a standard JVM. We also explore ways in which isolates could be used to structure the internal implementation of a J2EE server. We touch briefly on the issue of inter-application resource management but a full discussion of this topic is beyond the scope of this paper.

The rest of the paper is structured as follows. Section 2 provides an overview of the MVM architecture and the isolate programming model. Section 3 contains an overview of the J2EE platform, outlines the J2EERI architecture and discusses the use of classloaders to achieve isolation. Section 4 describes the straightforward application of MVM to support multiple J2EERI server instances and provides comparative performance measurements against a standard JVM. Section 5 discusses how isolates might be used explicitly in the implementation of a J2EE server. Section 6 describes two particular experimental implementations and compares performance against the results of Section 4. Section 7 discusses related work, Section 8 discusses ideas for future work and we conclude in Section 9.

2 BACKGROUND ON MVM

MVM is a general-purpose virtual machine for executing multiple applications that are written in the Java programming language. It is based on the Java HotSpot™ virtual machine (HSVM) [Sun00a] and its client compiler, version 1.3.1 for the Solaris™ Operating Environment [MM01].

Applications executing in MVM are referred to as *isolates* [JCP01]. MVM-aware applications can use the provided API to control the life-cycle (e.g., creation and asynchronous termination) of other isolates. The main (first) isolate does not have to be an application manager – it can be any application written in the Java programming lan-

guage. A simple example of the API is the creation of an isolate, which will execute MyClass with a single argument "abc":

```
new Isolate("MyClass", new String[] {"abc"}).start(...);
```

The key design principle of MVM was to examine each component of the JVM and determine whether sharing it among isolates could lead to any interference among them. Such components are either replicated transparently on a per-isolate basis or made *isolate re-entrant*, that is, usable by many isolates without causing any inter-isolate interference. They include static fields, class initialization state, and instances of class java.lang.Class. Several components of HSVM also needed modification to become isolate re-entrant.

An arbitrary number of isolates in MVM can share the code (bytecode and compiled) and much of the related metadata, of both core and application classes. Runtime modifications make the replication of non-shareable components transparent. In effect, each application "believes" it executes in its own private JVM, as there is no interference due to mutable runtime data structures visible directly or indirectly by the application code. Similarly, certain runtime (JRE™) classes, such as System and Runtime, had to be modified to make operations such as System.exit() apply only to the calling isolate.

The heaps of isolates are logically disjoint. The separation of isolates' data sets in MVM implies that isolates cannot directly share objects, and the only way for isolates to communicate is to use copying communication mechanisms, either standard ones, such as sockets, or custom protocols [PCD+02]. Another option is to use *links*, which are a low-level isolate-to-isolate communication mechanism introduced in the isolate API [JCP01].

In MVM, most of the class representation and the class loading, linking, and run-time compilation effort is shared. In particular, only when a class is loaded into MVM for the first time, are the actual file fetching, parsing, verification, building of a main-memory run-time representation of the class, and several other steps performed. These do not need to be repeated when another isolate uses the same class. This provides a significant reduction in the startup time of programs, such as J2EE servers, that comprise a large number of classes.

2.1 Multi-user Capabilities

MVM uses the process facilities of the underlying operating system to encapsulate the ideas of protection and access control [CDT03]. A single instance of MVM exists as one process and contains multiple isolates. Isolates may be started within MVM by different users through a separate login program called *Jlogin*, written in C. Jlogin corresponds to a notion of a user session, and is used to start a single isolate – the user simply types in the name of the main class and its arguments, similarly to running the standard "java" command.

After session initialization, Jlogin serves as a daemon process that services all requests generated by its associated isolate that require the user identity to be correctly set, e.g., accessing the file system, environment information or spawning subprocesses. The Jlogin process has the effective user id and associated privileges of the actual user, regardless of the process attributes of MVM (Figure 1).

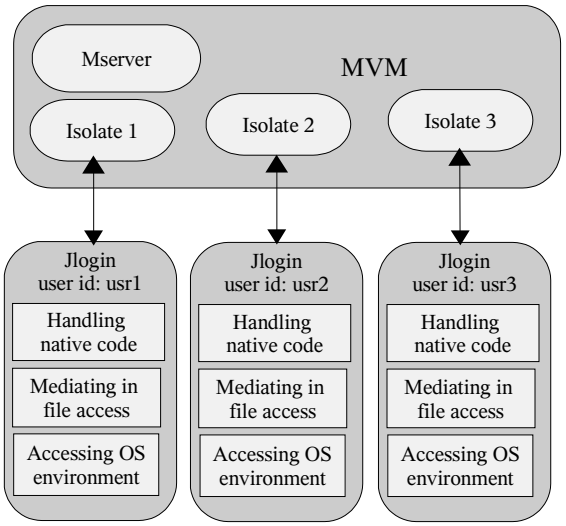

Figure 1: Three users execute applications in MVM; each of them has an instance of the Jlogin process.

The handling of the standard input/output/error streams is similar to that found in the familiar op-

erating system process model. Each isolate has its own instance[1] of a Jlogin process that, in addition to encapsulating the user-id information, also hosts user-supplied/untrusted native code libraries, so that a failure of native code associated with one isolate does not affect the others. This is especially important for the J2EE environment as the J2EE Connector Architecture is frequently used to access legacy code loaded as a native library.

The first isolate of MVM is a simple application called Mserver that listens on a socket for connections from Jlogin processes. Each new Jlogin connects to Mserver and the two exchange information such as relevant environment variables and user settings. Jlogin then sends a request to Mserver to create an isolate to run the application the user specified when starting that Jlogin instance. The isolate connects to its Jlogin's standard input, output, and error streams. Multiple Jlogin processes from different users can connect to the Mserver to launch their applications within the same instance of MVM.

Communication between isolates may take place using any of the standard means provided by the underlying operating system, for example, files or sockets. To communicate via the link mechanism of JSR121, a newly created isolate must be provided with at least one link in its start method. It can then use one of these links to receive messages which may contain additional links on which to communicate.

The default Mserver application starts isolates with a null set of links, i.e., the expectation is that each isolate started by Mserver is completely independent of the others. To provide flexibility in the initial setup, it is possible to supply Mserver with an alternate factory class that can customize the initial environment in which the isolates operate. In Section 5 we describe how this capability is exploited in the J2EERI environment to monitor a collection of server components.

3 J2EE OVERVIEW

We first provide an overview of the architecture defined by the J2EE specification and then outline the implementation architecture of the J2EERI.

3.1 J2EE Specification

The J2EE 1.3.1 specification defines four mandatory application component types:

- Application Clients
- Applets
- Servlets, Java Server Pages
- Enterprise Java Beans

These components have access to some or all of the following required services:

- HTTP, HTTPS, RMI-IIOP communication
- Java™ Transaction API (JTA)
- Java Database Connectivity (JDBC™)
- Java IDL(CORBA Interface Definition Language)
- Java™ Message Service (JMS)
- Java Naming and Directory Interface™ (JNDI)
- JavaMail™, JavaBeans Activation Framework™ (JAF)
- Java API for XML Parsing (JAXP)
- J2EE™ Connector Architecture
- Java™ Authentication and Authorization Service (JAAS)

The J2EE platform also requires a database accessible through JDBC.

The four application component types are required to be hosted in a container that interposes between the application and the federation of available services. The abstract relationships between the containers and services is shown in Figure 2. Note that while suggestive of physical structuring, the architecture does not imply any particular realization. In principle all elements could be instantiated in a single JVM, although they would typically span JVMs and machines in a large installation. Although not shown in the diagram, the containers themselves and some of the services, e.g., JMS, are typically embedded in servers – for example, a Web server, an EJB server and a JMS server. We use the term *J2EE server* to describe the aggregate of these logically

[1] Lazily created for isolates created by program code.

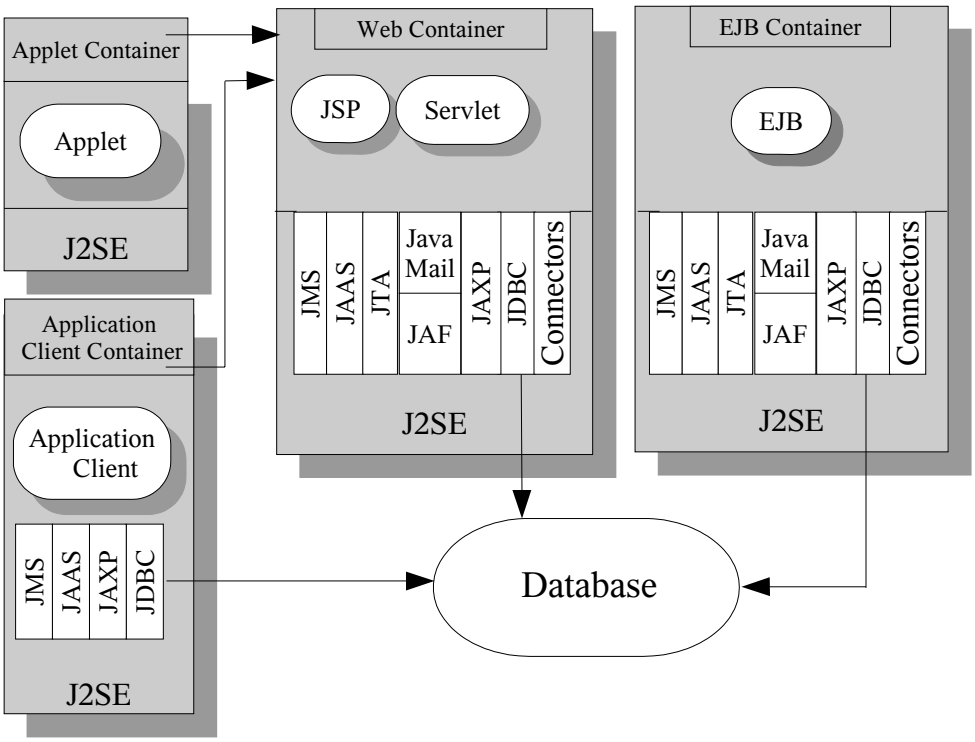

Figure 2: The J2EE Architecture.

embedded servers even if, in practice, there is no corresponding concrete entity. In the remainder of the paper, an unadorned use of the term "server" should be taken to mean J2EE server. We use the term "sub-server" to indicate a generic embedded server and the qualified form, e.g., Web server, when appropriate.

3.2 J2EERI Architecture

The J2EERI implementation is written entirely in the Java programming language. It includes a version of the Tomcat [ASFb] Web server. The JMS service depends on the Cloudscape relational database [IBMC], which is included with the J2EERI, and is also written entirely in the Java programming language. At runtime, all servers execute in the same JVM, including the Cloudscape instance that supports JMS. Typically an additional instance of Cloudscape, that executes in its own JVM, provides the external database component for JDBC access from deployed applications, although any database with a JDBC driver can be used.

The internal structure of the J2EERI is relatively complex, in particular the Object request broker (ORB) that underpins the Java IDL service, the RMI-IIOP communication service, and the naming service accessed through JNDI. However, the Web, EJB and JMS sub-servers are relatively well encapsulated. These sub-servers all make use of the ORB for naming and inter-server communication.

3.3 Classloaders and Isolation

The J2EERI and most J2EE servers make extensive use of the classloader framework provided by the Java platform [LB98]. Since classloaders and isolates share some of the same characteristics, it is instructive to analyze their use in J2EE servers.

Classloaders provide several basic capabilities:

- The capability to load multiple instances of the "same" class in the same JVM. By "same" we mean "has the same fully-qualified class name," regardless of whether the class content is identical or not.

- The capability to alter the basic JVM search mechanism for the classfile, for example, to load classes from remote machines or specific directories.
- Runtime support for "isolating" objects by tagging the effective type of an object by both its class and its classloader.
- The capability to interpose on the loading of the class, for example to perform dynamic bytecode transformations.

J2EE servers, including the J2EERI, make extensive use of the first three capabilities. More recently, JBOSS [JBOSS] has exploited the transformation capability to customize the server's capabilities at runtime.

Evidently the first three capabilities allow a J2EE server to act more like a traditional operating system in the sense that applications created after the server was started may be loaded (deployed) and executed. Further, applications may be unloaded, a feature that is not available for classes loaded by the standard JVM mechanism.[2] A modicum of security is also achieved by the JVM preventing the passing of otherwise equivalent objects between classloaders, by causing a class cast exception to be thrown.

Isolates provide similar capabilities to classloaders but the isolation and termination guarantees are much stronger. Classloader "termination" is essentially dependent on the garbage collector finding the classloader to be unreachable. Further, the isolation provided by classloaders is well known to be incomplete and error-prone as objects can leak and be captured. Isolates, precisely since they cannot share objects, deliberately or accidentally, can be terminated and unloaded cleanly. However, as discussed above, the implications of this strict isolation mean that the grain cannot be as fine as that available with classloaders.

There is no doubt that the use of classloaders adds complexity and opaqueness to a system. For example, debugging a class cast exception that is caused by a classloader mismatch can be very challenging.

The classloader structure of a typical J2EE server is surprisingly complex, much of it due to the need to isolate some classes but share others. For example, while servlet classes and EJB classes can occupy separate classloaders, there is a problem with (non-system) utility classes used by both. One potentially expensive solution would be to load (replicate) the utility classes in each classloader. Generalizing from the basic need for a non-replicated solution to the core classes, classloaders can, since JDK 1.2, be organized in a hierarchy allowing shared classes to be loaded in parent classloaders. It is this mechanism that leads to the complex classloader structures in J2EE servers. For example, to handle the utility class problem, the J2EERI loads such classes into the EJB server classloader and makes this the parent classloader for the Web server components.

Note the very important distinction between the use of a classloader hierarchy to share application data (state) and the use to merely share code. In the latter case, isolates arguably offer a fundamentally better solution by providing genuine encapsulation while retaining all the benefits of code and meta-data sharing. In the data-sharing case isolates cannot currently provide an equivalent solution, as there is no notion of partial isolation even between parent and child isolates. It is an area for future work to determine whether it would be possible to provide a transparent and safe data-sharing capability[3] in MVM that could replace this use of classloaders.

MVM currently cannot share class meta-data and compiled code for classes loaded in user-defined classloaders. Therefore, the footprint of the J2EERI server with deployed applications that contain duplicated classes is larger than it should be. However, we have recently developed a variant of HSVM that can share classes loaded by user-defined class loaders when appropriate, and it is expected that this will be integrated into a future version of MVM.

[2]The application is unloaded by the server discarding its classloader. However, the associated classes are only unloaded when the garbage collector determines they are unreachable.

[3] Analogous to the copy-on-write facility of many operating systems.

4 J2EERI on MVM

The most simplistic use of MVM is to run multiple instances of an "isolate-unaware" application in the same MVM instance. By "isolate-unaware" we mean an application that does not make any explicit use of the isolate API. In the J2EERI context this amounts to running multiple J2EE server instances using the Jlogin process described above. In this case, we are effectively replacing an entire JVM with an isolate, thereby providing a direct way to compare memory footprint and startup time.

Although the MVM system takes care of correctly managing the shared state at the Java platform level, it is still necessary to examine any use of state external to the Java platform for unintended sharing. For example, the J2EERI stores a great deal of state in the file system. Although MVM takes care that each server instance has independently controlled access to these files, it is the case that, by default, all instances would access the same pathname, just as if multiple instances of the server were run in individual JVMs.

As delivered, the J2EERI supports only one server instance per machine. It does, however, support the sharing of a single file system tree between separate machine instances, by encoding the machine name in all the pathnames to file system state. In order to support multiple server instances under MVM we augmented the pathname with an additional discriminant. We chose a very expedient solution to this problem that required no changes to Java code, only to the startup scripts for the server and related tools. At the root of the pathname we introduce a "servers" directory and, below this, named subdirectories for each server instance. This also has the virtue of supporting server-instance-specific configuration files, since, for example, each server instance must listen on distinct sockets for its various communication channels, e.g., HTTP, HTTPS, and this information is read from files on server startup.

In this arrangement, the server instances are logically distinct and are unaware of each other. In principle, each server could be configured and run with a different set of applications. However, to allow accurate footprint comparisons between an MVM-based system and an HSVM-based system, we always configure the servers identically. Figure 3 shows the resulting isolate structure of the multi-server (MS) setup. The arrows indicate the isolate creation relationship. In this case all isolates are created by the Mserver.

When comparing the MVM-based and HSVM-based systems, there are three data values that are of interest. The first is the total memory footprint associated with the set of server instances. The second is the startup time of the first and subsequent server instances, and the third is the performance of an application deployed on the server. While startup time may not seem especially important for a server application, it contributes to the Mean Time To Repair (MTTR) which is a key component in the calculation of total system availability.

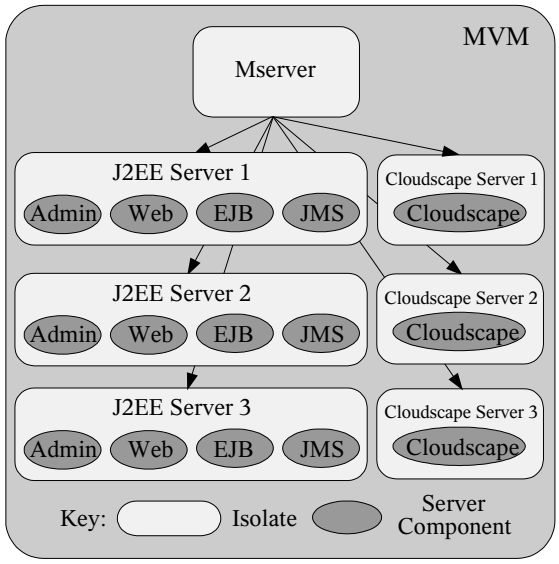

Figure 3: Multi-server isolate structure.

All tests described in this paper were run on a dedicated dual-processor, 1015Mhz, SPARC® server with 4GB of main memory.

4.1 Memory Footprint Measurements

Memory footprint is calculated from data generated by the Solaris *pmap* utility and correlated with JVM-specific data on the number and use of various memory regions. We ignore those regions that correspond to mapped *jar* files since, while these are large for a J2EE server – as much as

50% of the reported footprint – they are also read-once and potentially sharable. When calculating the footprint of multiple HSVM processes we only count the regions corresponding to shared libraries once. We show only the resident (physical) memory associated with the process, although additional (virtual) memory may have been reserved. Since the test machine is equipped with 4GB of physical memory, we can be confident that the reported resident memory accurately represents the maximum footprint.

A true apples-to-apples comparison of HSVM and MVM memory is impossible, because MVM's generational heap architecture has been modified to support multiple isolates efficiently by providing a new generation per isolate. In the current MVM implementation, the maximum number of isolates is fixed and certain data structures are sized accordingly on startup. In the tests we ran HSVM with the standard heapsize defaults (i.e., max 64MB) and MVM with a maximum of eight isolates and a max heap size of 128MB. However, it is important to stress that we only count resident (i.e., allocated) heap memory in the footprint.

4.1.1 Server Startup Footprint

We first measure the memory footprint for a set of five server instances, with a default configuration, and no deployed applications, as shown in figure 4.

Evidently, MVM demonstrates substantially improved scalability over HSVM. There is a very slight (0.05%) increase in the MVM footprint compared to HSVM for the initial server instance. A 31% reduction in total footprint occurs for two servers and a reduction of 43% reduction for three. The trend continues for additional servers, reaching a 55% reduction for five. The incremental per-server overhead for HSVM is approximately 20MB, against 5MB for MVM.

These numbers demonstrate the substantial overhead from the duplication of class metadata that occurs in multiple HSVM instances for large programs like J2EERI. Unlike micro-benchmarks, which contain only a few classes, a J2EE server contains several thousand loaded classes by the time it is ready to accept requests. For J2EERI, the meta-data overhead from this large set of classes is 15MB per server instance, i.e., 75% of the total overhead. This meta-data is much larger than the code of the virtual machine itself, which only contributes 30% of the total footprint. MVM's mechanisms for sharing class metadata are therefore proportionally more effective on large systems like J2EERI and limit the footprint increase to the actual amount of server-specific data (as opposed to metadata). The following table shows the percentage of the footprint attributable to data, metadata and code for MVM as the number of server instances increases.

Memory %	***S1***	***S2***	***S3***	***S4***	***S5***
Data	20.06	29.62	37.13	41.41	45.43
Metadata	57.38	51.1	46.1	43.58	40.84
VM code	22.56	19.28	16.76	15.01	13.63

Data corresponds to objects allocated with the new operator. Meta-data is all other memory allocated by the virtual machine, including class meta-data, constant pool, bytecodes, compiled bytecodes and other runtime data structures. Note that the VM code, which is the only portion that can be shared by the standard operating system mechanisms, is a small and decreasing contribution to the footprint.

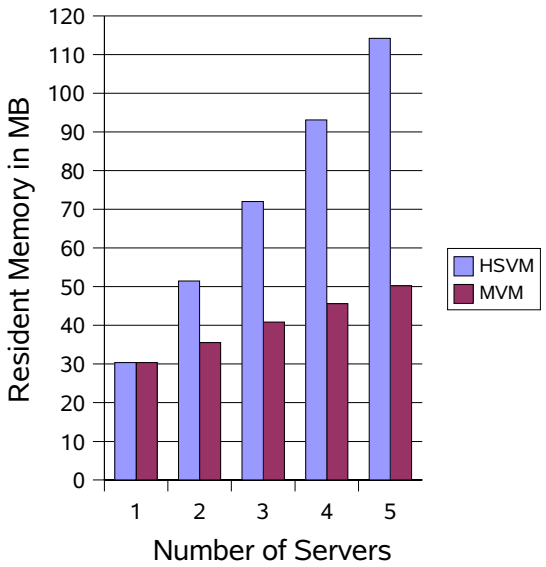

Figure 4: Memory footprint comparison of multiple J2EERI servers under HSVM and MVM.

4.1.2 Deployed Application Footprint

The server startup measurements provide a lower bound on the memory consumption. To get more realistic measurements. we deployed and ran an application on each server. We chose the well-known Petstore application [Sun03d], that was written to showcase the J2EE platform. Petstore requires an SQL database and we use a bundled copy of Cloudscape. In this test, we provided a separate Cloudscape database for each server instance, so that the J2EE server instances are completely independent, end-to-end.

There are two ways to bring up the system. First, start all Cloudscape instances, then all J2EERI instances, then deploy Petstore on all servers, and finally execute each instance of Petstore in turn. In the second approach, each instance is started, deployed and executed in turn. Both orderings have the same memory behavior in the case of multiple HSVM processes. For MVM, we measured both orderings and observed only small differences due to the interaction of the different orderings and the shared garbage collector. We show the results for the first order because it highlights the costs for each stage more clearly. To limit the size of the graph, we only measured three instances. In the HSVM case this corresponds to six HSVM processes, three for the Cloudscape servers and three for the J2EE servers. Note that the deployment tool requires an additional process, although the majority of the work actually takes place in the J2EE server. In the MVM case, we could have executed the deployment tool as an isolate but chose not to as it contributes little to the footprint.

Figure 5 shows the cumulative memory footprint for each stage of the system. The x-axis uses the following key: **Cn**: Cloudscape server n started; **Sn**: J2EE server n started; **Dn**: Petstore deployed on server n; **Rn**: Petstore run on server n. The application is accessed through a web browser and a single item is purchased from the store.

In this test, MVM has a 22% increase in footprint for the first Cloudscape server. This is due to Cloudscape having a very small footprint on startup (unlike the J2EE server in the previous test), so the additional overheads of the MVM dominate. However, by the third Cloudscape instance MVM has a 20% smaller footprint. The maximum footprint reduction of 46% occurs at the startup of the third J2EE server instance. During the deployment phase the reduction drops back to between 27-31% and the footprint growth rate is similar for HSVM and MVM. This is due to the fact that MVM is not able to share the classes that are loaded in separate classloaders. The rate of footprint increase for MVM during the run phase is very slow, and the total reduction reaches 41% after all three instances have executed. The reason why the HSVM footprint climbs so rapidly during the run phase is mostly due to Cloudscape, which loads a large number of additional classes during the execution phase as the database is accessed. There are effectively six[4] Cloudscape instances in the system, and MVM is able to effectively amortize the class loading cost.

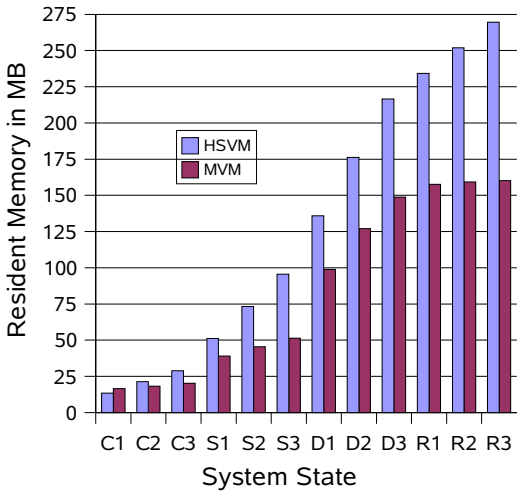

Figure 5: Memory footprint comparison of deployed Petstore application with Cloudscape database.

4.2 Startup Time Measurements

We hand-instrumented J2EERI to measure the elapsed time for the startup of the server components. This immediately showed a considerable speedup for second and subsequent server runs, due to two expensive one-time costs. These are the creation of a secure random number seed,

[4] Three for the database servers and three for the internal JMS implementation.

which is cached in a server instance-specific file, and the initialization of the JMS database, which initializes a large number of files. To focus on the normal startup time of a server, all reported measurements are for second server runs.

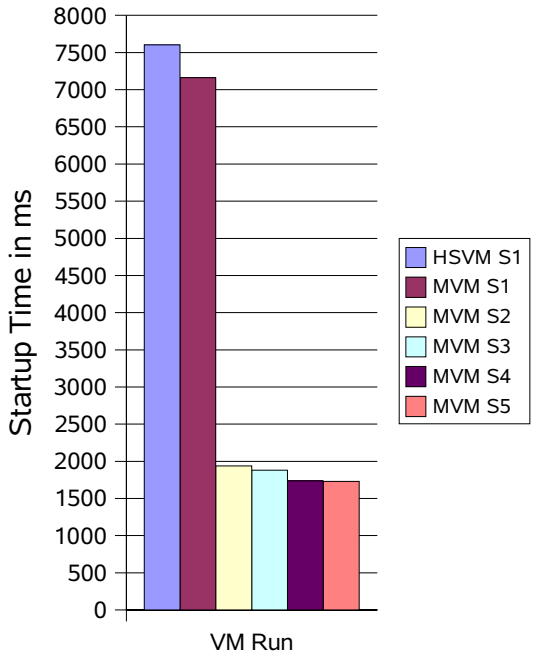

Figure 6: Startup time comparison of J2EERI server instances under HSVM and MVM

The first server instance that is run under MVM shows a 5.8% decrease in startup time (figure 6). This is actually atypical as there is normally a slight increase caused by the small JVM overheads needed to support isolates. However, in this case, the speedup is due to the fact that the Mserver has already loaded the network classes to support the Jlogin communication, so this cost is not paid by the J2EERI server, which makes extensive use of the network classes. The second and subsequent server instances show a marked decrease in startup time, reaching 77.3% at the fifth instance, that is due to the main body of the server classes already being loaded. The small reduction between the second and fifth instance is due to another virtue of MVM: the threshold counter for compilation of methods is global to all isolates and, therefore, the latter isolates do not incur this compilation cost.

4.3 Throughput Measurements

MVM introduces a small amount of additional compiled code, for example, to support the virtualization of class static variables. On micro-benchmarks the performance impact of these changes has been measured as a penalty of less than 2% [CD01]. To determine the impact on a J2EE server, we ran a simple throughput test by putting a simple application under constant load for a long period.

The application is very simple. The persistent data is a simple relation containing products identified by name and price. The data is modeled using an entity bean that is mapped to the database schema using container managed persistence. The throughput test consists of a servlet that creates a number of products and then executes some queries to locate specific subsets that are returned in a web page.

The throughput test was run twenty times for five minutes duration each, with the client repeatedly invoking the servlet with no think time, recording the number of successful transactions, which were then averaged across the twenty runs.

With the Cloudscape server running in a separate HSVM process, the J2EE server running on MVM showed a 1% increase in throughput and similar request response times. However, when the Cloudscape server was co-located in the same MVM, throughput was increased by 11% and the request response time was reduced by 36%. We attribute this improvement to the removal of the process context switching in the co-located case.

5 USING ISOLATES IN J2EERI

In the simple use of MVM described in the preceding section, there is no explicit use of the Isolate API and correspondingly no control point for managing the entire system. The virtue is that no code changes are required. However, it has the following limitations:

- Resources, such as heap size and CPU time, are managed according to the default behavior of MVM. Although MVM strives for fairness, explicit control would be useful. While beyond the scope of this paper, this can be achieved through isolate-specific resource management facilities [CHS+03].

- The components[5] of the individual server instances are not isolated from each other. Failure of an individual component implies failure of the entire server instance (but not failure of the entire system of instances).

If control of the individual servers is the main concern, we should note that it can be achieved in part by exploiting the ability to install a custom isolate factory class in the Mserver. Since this class can interpose on isolate creation and startup, it can exploit any of the features of the isolate API to control its child isolates. We have developed such a class to monitor J2EE servers and optionally notify registered listeners of the creation and destruction of server instances. This could be developed further to create a basic administration agent for a family of servers, without changing the main server codebase.

Going further and explicitly introducing isolates into the J2EERI implementation, the limitations can be addressed more completely. However, in a system as complex as J2EERI, there are a variety of ways in which isolates might be used, not all of which are advantageous.

Recall, in particular, that there is no sharing of data permitted between isolates. All communication must be performed by passing values through existing mechanisms such as serialization over sockets or RMI, or by the link facility of the isolate API. Therefore, without extensive reprogramming, one can only place components in separate isolates that are already architected to communicate using these mechanisms. Fortunately, the J2EE architecture was designed from the outset to be capable of spanning multiple machine tiers, and therefore key components were specified to support (remote) communication, passing arguments by value.

5.1 Large-grain Isolation

Key large-grain components of J2EE architected to communicate remotely are:

- Application and browser-based clients
- Web server
- EJB server
- JMS server
- Database server

In a conventional large-scale J2EE deployment, each of these servers would be located in a separate JVM or process. Further, there would likely be multiple instances of some sub-servers, for example Web servers, possibly hosted on a separate tier of machines. Even in environments that do not have a separate Web server tier, it is still common to run the Web server and the EJB server in separate JVMs, although the resulting communication overhead is causing a trend towards co-location in the same JVM.[6] Separation is only strictly necessary when the Web server is not Java technology based, e.g., Apache [ASFb].

While beyond the scope of this paper, we should note that the isolate programming model is applicable to multi-tier environments, and that work is underway to extend the implementation to support machine clusters. In this paper we concentrate on the existing MVM implementation and single machine scenarios. In either case it is important to understand the costs and benefits associated with a particular component architecture.

5.2 Small-grain Isolation

Evidently the components architected for distribution in J2EE are large-grain and at the level of sub-servers. This is based on the well-known guidelines for distributed computing: if a component has a low-bandwidth communication interface, makes few references to data objects in other components, and performs significant computation, then distribution will be beneficial.

However, it may be possible to find smaller subcomponents within the sub-servers that might benefit from isolation. For example, one might place an individual servlet in its own isolate (section 6.1). Another example would be the use of an embedded compiler to compile generated code for EJBs. Whether such isolation will perform adequately depends largely on the data access patterns of the component. Potential benefits include the additional control and modularity that results

[5] We use the term *component* to indicate any set of objects (and associated classes) that provide a well-defined function – as large as an entire sub-server or as small as a servlet.

[6] Essentially, modern J2EE servers are becoming multi-faceted systems that can act as web servers, EJB servers, or both as the situation demands.

from the isolation, but these must be balanced against potential loss of performance resulting from inter-isolate communication.

It is possible that the additional control opportunities that arise from isolation may be significant enough to outweigh the performance concerns. For example, compilers, which might be dynamically invoked to compile Java Server Pages, can be prolific consumers of heap space. In a monolithic server design, this data is mixed in with that from other components and may have a negative impact on overall garbage collection performance. If the component is isolated, it is very much easier to both control the heap space allocated to the compiler and also do expedient cleanup. Recall that MVM allocates each isolate its own new generation in the heap. In a standard JVM, the compiler's allocations in a shared new generation might trigger a full garbage collection.

5.3 Sharing Considerations

Before considering how we might use isolates in a small-grain approach, it is important to discuss the impact of shared components in a J2EE system.

While at one extreme, every component might be replicated (shared-nothing), in practice, constraints, such as a shared database, may require that certain components be shared. In this case, one must be careful to avoid unintended replication of logically shared resources. Nevertheless, considerable configuration flexibility is available within the J2EE framework. For example, one might choose to provide each application with a separate HTTP server, and hence a separate port number, or share a single (isolated) HTTP server, with a port common to all applications.

Similar considerations apply to the database. For example, while a single Cloudscape server instance is capable of handling a number of different databases, multiple servers are also an option provided that we take care to avoid conflicts such as port numbers and database file locations. The multiple servers may realize an overall performance improvement because each database server is focused on one database and one set of queries. On the other hand, it may be the case that all the applications running in the server logically share the same database. In this case, Cloudscape requires that a single instance manage the database, and it would be an error to replicate the Cloudscape server in each application isolate. Similar considerations apply to the JMS server which, internally, also uses Cloudscape for persistent message queues.

Therefore we can see that logically shared components will typically impose some top-level structuring into isolates. Beyond that there are two basic approaches to further decompose a server into isolates, *architecture-based* isolation and *application-based* isolation, that we discuss below.

5.4 Architecture-based Isolation

The straightforward use of isolates follows the logical architecture of J2EE and places each sub-server in a separate isolate. The advantages of this approach are:

- Simple and minimal changes to J2EERI
- Extends easily to a multiple-machine environment

The disadvantages are:

- Communication overhead is maximized
- Resource management must be implementation-component-based rather than application-based

Evidently this approach is driven primarily by the J2EE architecture and by the particulars of the J2EERI implementation architecture. In essence this approach uses an isolate in place of a separate JVM process. This has the potential benefit of faster communication between sub-servers if the code is modified to use inter-isolate links instead of sockets.

In this approach a complete J2EE application would span multiple isolates (sub-servers), and these sub-servers would also support multiple applications simultaneously. Since the architectural decomposition is predominant, it is difficult to reconstruct an end-to-end application view and therefore difficult to manage resources at the application level. The resulting structure would be similar to that found in many large legacy systems that use operating system processes as a decomposition mechanism.

5.5 Application-based Isolation

This approach takes the position that the isolation of *whole applications* from each other is the most important goal. One or more complete applications are placed in an isolate that we refer to as an *application domain*. The advantages are:

- Communication overheads are minimized. Local, pass-by-reference interfaces may be used, e.g., EJB local interfaces
- Interference between possibly conflicting application demands is minimized
- Supports application-level resource management

The disadvantages are:

- Each application domain has its own copy of a sub-server. While the code of the sub-server instances can be shared automatically by MVM, the sub-server data cannot, which may lead to a larger footprint
- There is no fault isolation of individual components

In its purest form every sub-server is replicated in this approach, including, for example, the database.

Note also that, unless there are *some* global (shared) sub-servers, this approach is equivalent to the multiple-J2EE server instances discussed in Section 5. The only difference is whether the creation of the isolates is implicit through the Jlogin process or explicit in the code of the J2EE server startup class.

5.6 Combining Both Approaches

It is possible to combine the two approaches since an isolate may create additional isolates.[7] For example, within an application domain isolate, one could create extra isolates that represent an architectural decomposition, e.g., a separate Web server, EJB server, JMS server and database server, provided this doesn't violate any sharing constraints. The main advantage of this is increased fault isolation. In principle a failed component could be restarted without restarting the entire domain.

This order of decomposition is relatively easy to achieve without major code changes. In contrast, it is much harder to add application domains to an architectural decomposition since, unlike multi-tier capability, this requirement was not anticipated by the original designers and therefore inevitably requires significant changes to the source code.

6 EXPERIMENTS

In this section we describe two experiments that were carried out to evaluate the above approaches.

6.1 Servlet Isolation

In an earlier experiment with the Tomcat web server, we evaluated the benefits of placing an individual servlet in an isolate.

In this experiment, the Tomcat servlet engine, Catalina, runs in the base isolate of an MVM. The modified engine can run servlets in the base isolate in the standard way, or *iservlets* in their own isolate. An iservlet is indicated in the servlet code by implementing a tagging interface Iservlet.[8] To minimize changes to the Catalina code base, the existence of servlet isolates is hidden from Catalina and encapsulated in a `ProxyIServlet` class. In fact, all the major servlet classes have proxies that simply forward requests to and from the iservlet on links.

This is an example of the architectural approach. Since, in this experiment, the only J2EE sub-server was the Web server, the initial architectural decomposition was trivial. Then, regardless of how many iservlets are defined in the J2EE application, each iservlet is placed in its own isolate, essentially mimicing the architectural decomposition of the application into iservlets.

The experiment highlighted the issues that arise due to the lack of object sharing between isolates. There are a number of situations where the servlet API allows direct communication between servlets and also indirect communication through objects created by Catalina. For example, nothing prevents a servlet passing a reference to itself to another servlet, which can then invoke methods

[7] Any apparent hierarchical isolate structure is an illusion. In particular, the API provides no way to discover or exploit implied parent-child relationships.

[8] It could also have been indicated in the deployment descriptor.

on the first servlet. Session state is an example of state that is shared between servlets and the base engine. While all of these cases can be handled by appropriate use of proxies and inter-isolate communication, the richness of the sharing possibilities makes the implementation relatively complex and possibly subject to performance problems. It is not clear from the servlet specification whether all of these possible inter-servlet interactions are considered good practice. However, it shows how difficult it is to enforce boundaries in a system initially developed in a single-address space environment.

This experiment clearly demonstrated that it is not trivial to take a system that was designed in the context of a shared address space and restructure it into multiple address spaces. Approximately 7000 lines of code were required to implement iservlets without modifying the Catalina codebase. The experiment discouraged us from pursuing the use of small-grain isolates in other areas of J2EE, such as individual EJBs.

In practice, it is likely that inter-servlet interaction would be confined to the set of servlets associated with a single J2EE application which, therefore, would make a better unit of isolation. We have not pursued this idea solely in the context of servlets but it underpins the ideas in the following section.

6.2 J2EE Application Domains

This experiment attempts to combine architecture-based isolation and application-based isolation in an optimal way. However, application-based isolation is the principal focus and, therefore, the end result closely resembles the multiple server instances of Section 4. The essential difference is that certain key services, for example common aspects of deployment, are shared and consolidated into an *administration* isolate. The administration isolate is responsible for creating additional isolates, called application domains, each of which can host one or more applications. The unit of management is therefore the domain rather than an individual application, but the latter can be achieved as a special case. This structure is similar to that provided by many commercial application servers that provide an administration server instance to manage domains of other server instances [Sun03a]. However, in contrast, the administration isolate is much simpler.

By default each application domain is a single isolate and contains a Web server, a JMS server and an EJB server. This configuration is referred to as a *combined* domain and is shown in figure 7.

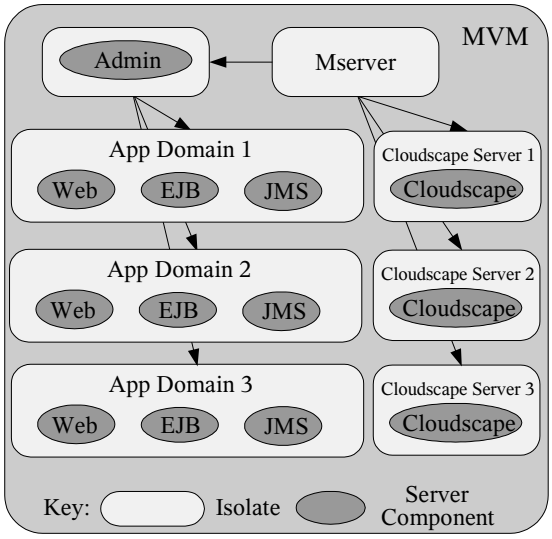

Figure 7: Combined domain isolate structure

Each application domain optionally can be further structured such that the Web server, JMS server and and EJB server are instantiated in separate isolates. This configuration is referred to as a *separated* domain, shown in figure 8.

In both configurations, the communication between the administration isolate and the application domain isolates is handled using links. Each domain maintains a thread that listens for messages from the administration isolate. The majority of messages concern the domain-specific aspects of deployment and requests to shut-down. In the separated configuration, communication between the individual isolates continues to use the existing CORBA-based mechanisms, although this could also be replaced with link-based communication.

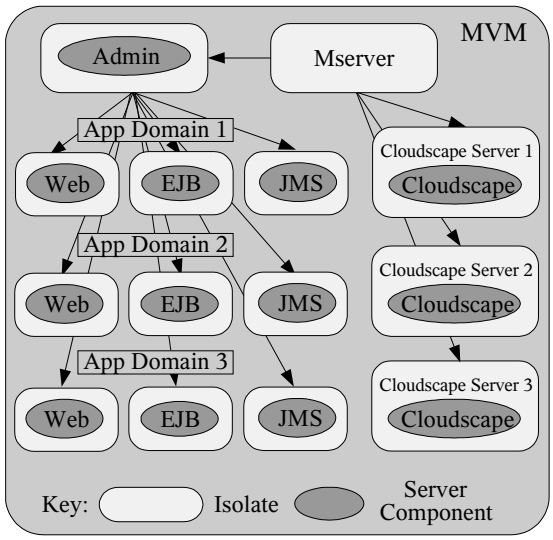

Figure 8: Separated domain isolate structure.

6.2.1 Footprint and Startup Time Comparisons

The existence of the administration isolate in the application domain system slightly increases the memory footprint compared to the multiple server case.

Figure 9 compares the memory footprint of five server instances against five application domains, in both combined and separated modes. The X-axis label is nX, where X is M,C,S, for multi-server (MS), multi-domain combined (MDC) and multi-domain separated (MDS), respectively, and n is the number of instances. We also show the footprint for the administration domain in the multi-domain configurations. To see the relative growth in different contributions to the footprint, it is broken down into VM code, meta-data and data. The MS data is the same as that presented earlier in figure 4. The overhead for MDC configuration relative to the MS configuration is 2.76MB for one application domain, which is due mainly to the existence of the administration domain, and the consequent duplication of several core services. However, the average overhead for domains two through five is only 0.97MB per domain. This is because the web server instance used for client stub downloading only exists in the administration isolate whereas it is replicated in each instance in MS configuration.

In the MDS configuration the footprint is noticeably increased due to the replication of the core services, such as the ORB, in each sub-server. The average per-domain overhead compared to MDC is 7.4MB. Of this about 5MB is actual server data, and the remainder is additional meta-data. The number of isolates also increases, at three per domain, and this contributes a small part of the meta-data increase. Note, however, that the total MDS footprint is still about 20% below that incurred by multiple servers each in their own JVM (see figure 4).

Startup time in the MDC configuration is actually slightly improved for the second and subsequent domains over the MS configuration, as shown by figure 10. In the MDC configuration the first domain value (D1) includes the time to start the administration domain, that hosts the singleton web server used for client stub downloading. The reduced startup time for subsequent domains is due to not having to start this server in these domains. As with memory footprint, the need to duplicate core services in each domain, causes an increase

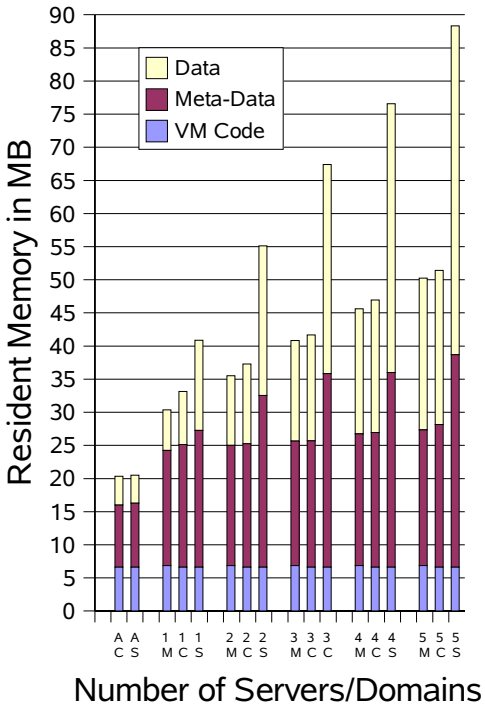

Figure 9: Memory footprint comparison of multiple server instances and multiple application domains.

in startup time for the MDS configuration, although it remains on par with the MS configuration.

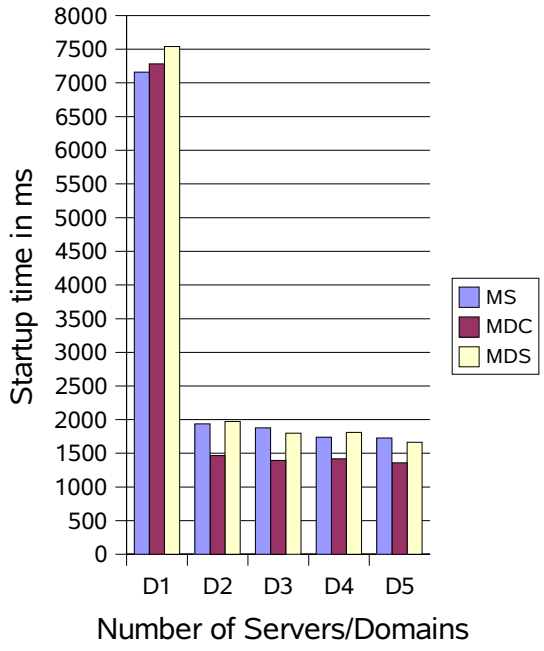

Figure 10: Startup time comparison for multiple server instances and multiple application domains.

We repeated the throughput test (section 4.3) in MDC configuration with equivalent results to MS configuration.

7 RELATED WORK

[BPW+01] describe a serially reusable JVM for transaction processing environments that was based on an earlier prototype JVM for the IBM OS/390 [DBC+00]. Many of the goals are similar to those of MVM but, as the title indicates, only one transaction at a time can execute in the JVM, which can be reset quickly to a clean state after the transaction completes. The system makes explicit provision for trusted middleware code, such as is found in a J2EE server. Middleware code is loaded in a separate classloader and can retain state across transactions provided that it maintains the "clean" invariant of the JVM. Callbacks are provided for middleware code to clean up after a transaction completes. The heap is also structured in a generational fashion to mimic the system structure, allowing fast cleanup of transient objects allocated by the transaction. However, middleware code must be careful not to retain references to application objects for this to work correctly.

[KKL+02] outlines the SAP VM Container, an application server framework that attempts to achieve the isolation of operating system processes without the cost of a complete JVM process per user transaction, through the use of a *Process Attachable Virtual Machine (PAVM)*. A PAVM captures all the JVM computational state in an isolated shared memory block, and can be attached to one of a small pool of worker OS process as needed.

The .NET platform [Micr02] defines a*pplication domains,* similar to the notion of isolates. Instances of System.AppDomain are virtual process isolating applications from one another. Multiple application domains can exist in a single OS process. However, there is no multi-user support – all application domains in the same OS process execute on behalf of the same user. Moreover, unlike MVM's isolates, .NET's application domains cannot safely use arbitrary native code. AppDomains are being used in some .NET servers, e.g., ASP.NET.

The Merlin sub-project of the Apache Avalon project [ASFc] provides a framework for constructing servers out of well-defined building blocks. Merlin makes extensive use of hierarchical classloader structures to both share and isolate components.

8 FUTURE WORK

Application domains provide the basis for future work that will exploit the resource management capabilities of MVM [CHS+03] to apportion resources between J2EE applications according to flexible policies.

Safe and transparent sharing of long-lived subserver data would further reduce the footprint of J2EE servers and would remove the major remaining benefit of classloaders. We plan to analyze the characteristics of long-lived data in a J2EE server and study ways to to extend the existing meta-data sharing capabilities of MVM to such data. This would be particularly beneficial to the separated domain configuration, as much of the long-lived data of the replicated components

in the separated domains is believed to be identical and mostly immutable.

Further research is merited into the more widespread use of fine-grain isolates for middleware code, particularly in the context of clustered systems. Current server architectures tend to be monolithic and/or result in large numbers of inter-object references that limit the opportunity to restructure for isolation, security, resource management or distribution. A more explicit, fine-grained, architectural approach such as SEDA [WCE01], coupled with isolates, would be worthy of investigation.

9 CONCLUSIONS

We have described how MVM, an efficient implementation of isolates, can support multiple instances of the J2EE 1.3.1 Reference Implementation, with a significant reduction of total memory footprint and server startup time compared with a standard HSVM. This was achieved with no changes to the server code, and no loss in performance, confirming that MVM can execute multiple applications transparently and efficiently. Throughput and response time improvements were shown to result from the co-location of the Cloudscape database server with the J2EERI server in the same MVM, while retaining isolation.

We discussed how a J2EE server might exploit the isolate programming model internally and described how we restructured the server to support the concept of *application domains,* which allow groups of J2EE applications to be fully isolated from each other, while sharing the same server infrastructure. Memory footprint and startup times were compared for two variants of application domains, *combined*, in which the server components occupied the same isolate and *separated,* where they occupied different isolates. The combined variant showed the best footprint and startup times. However, both variants had smaller footprints than the traditional approach of one JVM per server.

Overall, MVM has been shown to provide an efficient and scalable platform for hosting large, complex, multi-server applications with sound and flexible isolation. MVM is an important step towards a complete operating environment for enterprise servers based on the Java platform.

Acknowledgments. Kirill Kouklinski helped with the development and debugging of the application domain variant of J2EERI. Glenn Skinner and Jeanie Treichel read drafts of this paper and provided helpful feedback.

10 REFERENCES

[ASFa] The Apache Software Foundation. *Apache Http Server Project.* http://httpd.apache.org/

[ASFb] The Apache Software Foundation. *Apache Tomcat*. http://jakarta.apache.org/tomcat.

[ASFc] The Apache Software Foundation. *The Apache Avalon project.* http://avalon.apache.org

[BPW+01] Borman, S., Paice, S., Webster, M., Trotter, M., McGuire, R., Stevens, A., Hutchison, B., and Berry, R. *A Serially Reusable Java Virtual Machine for High Volume, Highly Reliable, Transaction Processing*. IBM TR 29.3406, Hursley, UK.

[CD01] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution.* ACM OOPSLA'01, Tampa, FL.

[CDN02] Czajkowski, G., Daynes, L., and Nystrom, N. *Code Sharing among Virtual Machines.* ECOOP'02, June 2002, Malaga, Spain.

[CDT03] Czajkowski, G., Daynes, L., and Titzer, B., *A Multi-User Virtual Machine.* Usenix Annual Technical Conference, San Antonio, Texas, June 2003.

[CHS+03] Czajkowski, G., Hahn, S., Skinner, G., and Soper, P., Bryce C. A *Resource Management Interface for the Java Platform.* Sun Microsystems Laboratories Technical Report, SMLI TR-2003-124, May 2003.

[DBC+00] Dillenberger, W., Bordwekar, R., Clark, C., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M., Samuel, F., and St. John, R. *Building a Java virtual machine for*

server applications: The JVM on OS/390. IBM Systems Journal, Vol. 39, No 1, 2000.

[GJS+00] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification. 2nd Edition.* Addison-Wesley, 2000.

[IBMC] *The Cloudscape database.* http://www.ibm.com/software/data/cloudscape/

[JBOSS] Fleury, M. and Reverbel, F. *The JBOSS Extensible Server*. Proceedings of Middleware 2003, LNCS 0558, Springer-Verlag, pp344-373, ISBN 3-540-40317-5.

[JCP01] Java Community Process. JSR-121: Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp.

[KKL+02] Kuck, N., Kuck, H., Lott, E., Rohland, C, and Schmidt, O. *SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers.* Work-in-Progress Session, Java Virtual Machine Research and Technology Symposium, San Francisco, August 2002.

[LB98] Liang, S. and Bracha, G. *Dynamic class loading in the Java virtual machine.* ACM OOPSLA'98, Vancouver, Canada.

[Micr02] Microsoft Corp. *.NET Web Page*. http://www.microsoft.com/net. 2002.

[MM01] Mauro, J., and McDougall, R. *Solaris Internals – Core Kernel Architecture.* Prentice Hall, 2001.

[PCD+02] Palacz, K., Czajkowski, G., Daynes, L., and Vitek, J. *Incommunicado: Efficient Communication for Isolates.* ACM OOPSLA'02, Seattle, WA, November 2002.

[Sun00a] Sun Microsystems, Inc. *Java HotSpot™ Technology*. http://java.sun.com/products/hotspot.

[Sun03a] Sun Microsystems, Inc. *Sun Java System Application Server.* http://wwws.sun.com/software/products/appsrvr/home_appsrvr.html

[Sun03b] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition (J2EE)*. http://java.sun.com/j2ee/index.jsp

[Sun03c] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition (J2EE) - Version 1.3.1 Release*. http://java.sun.com/j2ee/sdk_1.3/

[Sun03d] Sun Microsystems, Inc. *The Java Petstore Demo 1.3.2.* http://java.sun.com/developer/releases/petstore/

[WCE01] Welsh, M., Culler, D., and Brewer D. *An Architecture for Well-Conditioned, Scalable, Internet Services*. Proc. of the 18th Symposium on Operating Systems Principles, Banff, Canada, Oct 2001, pp230-243.

ABOUT THE AUTHORS

Mick Jordan is a Senior Staff Engineer at Sun Microsystems Laboratories. His interests include programming languages, programming environments, persistent object systems and systems software. He has a Ph.D in Computer Science from the University of Cambridge, UK. He was a member of the team that designed and implemented the Modula-3 programming language. He is currently working on the Barcelona project in Sun Labs investigating the application of the Multi-tasking Java Virtual Machine to the J2EE environment.

Laurent Daynès is a Senior Staff Engineer at Sun Microsystems Laboratories. His current research interests are in virtual machines for modern programming languages with advanced features such as orthogonal persistence, automated rollback and application concurrency control, multi-tasking, sharing of runtime compiled code across programs, and automated isolation of native method execution. He holds a Ph.D. in Computer Science from the University Pierre & Marie Curie (Jussieu Paris 6, France) for research he carried at INRIA.

Grzegorz Czajkowski is a Senior Staff Engineer at Sun Microsystems Laboratories, which he joined in the summer of 1999, after obtaining a Ph.D. in Computer Science from Cornell University. He is interested in operating systems, programming languages, and middleware. Currently he is leading the Barcelona project, which focuses on architectures for improving the scalability and reliability of the Java platform.

Marcin Jarzab is a Ph.D. student at the University of Science and Technology of Krakow, Poland. He obtained a computer science degree at the same university in 2002. He worked as a software consultant at Consol* Solutions and Software from 2000-2002, participating in many projects for Telco companies. He was an intern at Sun Labs in the latter half of 2003. His research interests include the tuning and performance evaluation of distributed systems, design patterns, frameworks and architectures of autonomic computing environments.

Ciarán Bryce is Assistant Professor at the University of Geneva, Switzerland. He obtained a computer science degree at Trinity College Dublin in 1991 and a Ph.D. From the University of Rennes in 1994. He worked as a researcher at INRIA-Rennes (in France) from 1991 to 1994 and at GMD (German National Research Centre for Information Science) from 1994 to 1997. He has been with Geneva University since 1997, and his research interests include computer security, object-orientation, and more recently, wireless information systems.