Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Masters Thesis

Performance Analysis and Improvement of GuestVM for running OSGi-based MacroComponents

by
Jianyuan Li

Due date
22. August 2010

Advisors:
Jan Simon Rellermeyer
Adrian Schüpbach
Prof. Gustavo Alonso

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland

Abstract

MacroComponents defined as software components that run in isolated environments but without the full foundations of the traditional software stack is an alternative, lightweight and composable approach to virtualization. GuestVM, which is a bare-metal, meta-circular Java Virtual Machine, is the core of the MacroComponents in this work. It sits directly between Xen hypervisor and OSGi applications to play a role as an operating system as well as a JVM. GuestVM has the potential for good performance because of its minimal, all-Java software stack and the elimination of traditional operating system layer. Nevertheless, GuestVM performs very poor in reality according to the previous work. The main aim of this work is to analyze and figure out the performance bottlenecks and remove them to improve GuestVM's performance, and thus make it a possible solution for MacroComponents in practice.

Through the evaluations, GuestVM does not add overhead to Maxine in terms of memory access, and the performance is comparable to HotSpot JVM without considering the optimization of HotSpot JVM's working memory. By optimizing code, giving sufficient file system buffer cache and implementing a prefetching mechanism, the I/O performance of GuestVM on average can be improved to only 50% slower than HotSpot JVM (even outperforms HotSpot JVM in many cases), and 4 times faster than Maxine. To run the OSGi-based MacroComponents, this work provides a design for efficient communications between GuestVM domains by using shared memory as connection channels, and presents a monitoring and management GUI tool to help developers to manipulate the MacroComponents in a graphical way.

Contents

Chapter 1

Introduction

1.1 Motivation

As opposed to a monolithic architecture, where the smallest component is the whole application, modularity is to design software as distinct manageable components. Modularity can be done at many levels. Modularized source code (e.g. classes) can be compiled together into one executable; at a higher level, an executable can be used as a shared library. An even higher level is to modularize applications. Modularization on the level of applications means that an application is consisted of collaborating but autonomous sub-applications (modules), and functionalities can be added or removed by installing or uninstalling modules dynamically at runtime rather than at compile time. Such systems are called extensible component platforms [1] (The OSGi framework is a widely adopted instance). When it comes to enterprise applications, a higher modularization level is needed: the level of JVMs isolation.

Since enterprise applications are huge and highly complex, they have the requirement of performance isolation. Performance isolation means that the performance of a component of an application should not be affected by the other components or other applications [2]. The Java Virtual Machine (JVM) [3] is an abstract computing machine that runs compiled Java programs to make Java applications hardware- and operating system- independent. However it is designed for standalone applications, and all Java components that run on the same JVM share time and space resources. Hence when it comes to extensible component platform applications, such as OSGi applications where each application can include any number of modules and each module can run any number of Java programs in the JVM, current JVMs are not able to isolate components from each other. For example, misbehaving applications could freeze the complete system by consuming too much CPU or block resources or impact other components by blocking resources or modifying shared variables [4]. The best isolation is to run each component as the only application on a separate machine. But this causes a waste of resbources due to hardware underutilization. *Virtualization* provides a solution by allowing to run multiple virtual machines (VMs) concurrently on the same physical machine in isolated environments, thus grants the component exclusive access to time and space resources. Virtualization here means that a system pretends to be two or more of the same system.

Though virtualization gives a solution to performance isolation so that improves system utilization and security, it brings overhead. To allow multiple VMs to run on one physical host, system virtualization provides an isolated duplicate of the real machine for each VM, including full operating systems and a complete software stack, which results in a large system that consumes resources. But much of this foundation is not strictly necessary. To reduce the overhead, the concept of *MacroComponent* is introduced [5]. MacroComponent is an alternative, lightweight approach to virtualization. Basically, it reduces the overhead by eliminating the need of an operating system as well as making the software stack smaller. In system composed of MacroComponents, software components run in isolated environments from the rest of the system, but without the full conventional foundations, that is operating system and runtime environment. In this work, the MacroComponent is designed that runs the bare-metal Java Virtual Machine (GuestVM) directly on the hardware hypervisor (Xen), and the JVM supports to run extensible component platform applications, i.e. OSGi bundles here. R-OSGi is a middleware layer based on OSGi handling the communication between the isolated MacroComponents. Figure 1.1 shows the architecture of this MacroComponent.

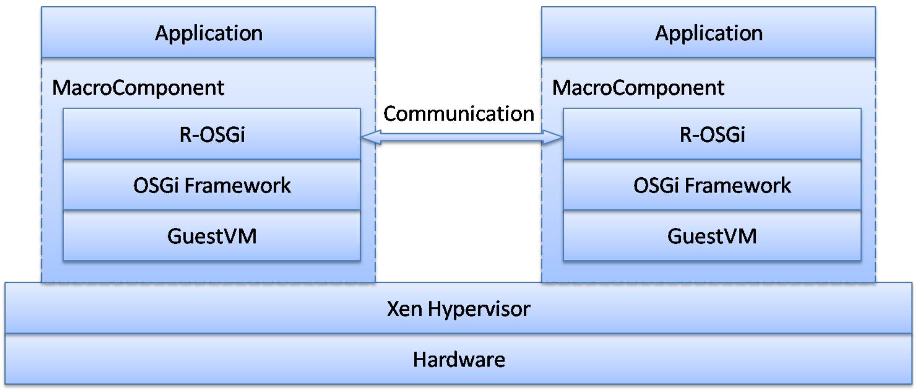


Figure 1.1: Architecture of MacroComponents

Since the primary goal of MacroComponents is to reduce the overhead brought by virtualization, its practical performance is a vital aspect that should be guaranteed.

In theory, software components with a small software stack should outperform the traditionally virtualized one with the complete foundations. But unfortunately, as the key part of the MacroComponent, GuestVM performs much worse than expected. From Schwammberger's work [2], GuestVM performs poorly when running SPEC JVM98 benchmarks. While Maxine is only a little slower than Sun's HotSpot JVM, GuestVM performs 8 to 9 times slower on average and is 26 times slower in the worst case compared to HotSpot JVM.

This work aims to analyze GuestVM performance and find which parts pose the performance bottlenecks and solve the problems to improve the performance, and then to make GuestVM a practical component of MacroComponents. In addition, to run multiple distributed OSGi-based MacroComponents, an efficient approach for communications between MacroComponents should be provided.

1.2 Related Work

MacroComponents introduce a concept for providing performance isolation with small overhead. GuestVM as a main part of MacroComponents in this work provides a complete software stack to potentially give high performance to specialized applications by removing the extraneous stuff of operating systems for general use. There are some other works related to this purpose. *IBM Libra* [6] gives another approach that relies on hypervisor (specifically, Xen) to transform existing software systems into specialized, high-performance systems without replacing the entire operating system. It is an execution environment specialized for running particular classes of workloads on the IBM's J9 JVM. In addition, the performance isolation could be achieved via operating system-level virtualization as *FreeBSD jail* [7]. It allows administrators to partition a FreeBSD-based system into several independent subsystems named jails, which are sealed from each other, thus providing an additional level of isolation and security. *Solaris 10 Zones* [8] is a further development of the idea of BSD jails. It gives an environment similar to virtual machine by creating an isolated process tree, but with minimal overhead.

1.3 Outline of the Thesis

Chapter 2 starts with some background on each component in the software stack of MacroComponents in a bottom up order. Chapter 3 presents the evaluations and improvement of GuestVM performance. Chapter 4 describes a design for communications between GuestVM domains and presents a GUI tool for monitoring and managing a MacroComponents system. Chapter 5 presents the idea of future work and concludes this thesis.

Chapter 2

Background

GuestVM [9] is the core component of the MacroComponent in this work. It is an experimental project from Sun that provides a lightweight runtime platform. GuestVM is an implementation of a Java virtual machine that built on Maxine virtual machine. It runs directly on the hypervisor paravirtualization API without the traditional operating system layer. Figure 2.1 presents the comparison between conventional software stack and GuestVM stack. GuestVM aims at building an all-Java software stack for server-side Java applications.

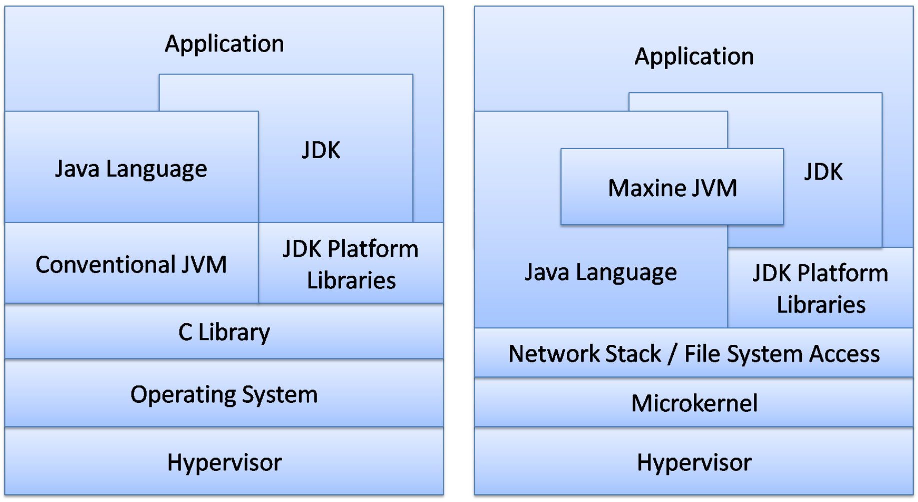


Figure 2.1: Conventional Stack vs. GuestVM Stack [10]

While conventional JVMs are written in C or C++, Maxine is implemented in Java, the same language that it interprets, i.e., a meta-circular JVM. Conventional JDK Platform libraries are implemented by JVM native code and operating system code layers. Maxine is a replacement for HotSpot JVM. It is fully JDK 6 compatible. In GuestVM, HotSpot JDK is unchanged in essence except for the native methods that depend on the VM details [10].

The following sections present every component in GuestVM stack in more detail, in a bottom up order.

2.1 Xen

Traditionally, an operating system, which acts as an intermediary between application programs and the computer hardware, runs directly on hardware and has exclusive access to all hardware devices. Because most hardware does not natively support being accessed by multiple operating systems, to run several virtual machines on one computer in parallel, where each virtual machine could run a separate operation system instance, it is necessary to introduce a software layer to schedule and allocate sharing resources (space and time), manage the hardware access, and monitor guest operating systems. *Hypervisor* [11] is such a layer, also known as *virtual machine monitor* (*VMM*). It sits between hardware and guest operating systems, and provides each guest with its own virtual device. *Xen* [12] is a *bare metal* (or *native*) hypervisor. In contrast to hosted hypervisors running within a conventional operating system, bare metal hypervisors run directly on the host's hardware.

Xen runs guests in domains, which encapsulate a complete running virtual environment. The privileged domain that can access the hardware is called *Domain 0* (*dom0*). It is the first guest to run when Xen boots, and its most obvious task is to handle devices. In contrast, other domains are referred to as *Domain U* (*domU*), where "U" stands for unprivileged. Communication between domains is by means of memory sharing. Figure 2.2 gives an example to show how an application in domU accesses the physical hardware.

When a packet is sent by an application running in a domU guest, it first goes through the TCP/IP stack as normal. Unlike a normal network interface driver, the bottom of the stack is a split device drive that puts the packet into some shared memory. The other half of the split driver, running on the dom0 guest, reads the packet from the shared memory, and inserts it into the firewalling components of the operating system that route it as it would a packet from a real interface. Finally, the packet is routed to the real device driver, which should already exist in the dom0 operating system. This is able to write to certain memory reserved for I/O. The physical network device then sends the packet [13].

The virtualization approach taken by Xen just brings a minimal performance overhead, but without sacrificing functionality. In the experiments for evaluating the overhead of the various virtualization techniques relative to running on the "bare metal", Xen performs just the same as Linux in the best case (The SPEC CPU2000 Integer suite [14]), and about 8% overhead at worst (PostgreSQL running the OSDB multiuser Information Retrieval (IR) benchmark); normally, the overhead is around 3% ˜ 4%. [15, 16, 17] The evaluation shows that Xen considerably outperforms than other VMMs such as VMware workstation.

Besides excellent performance, Xen also provides sufficient isolation. In the

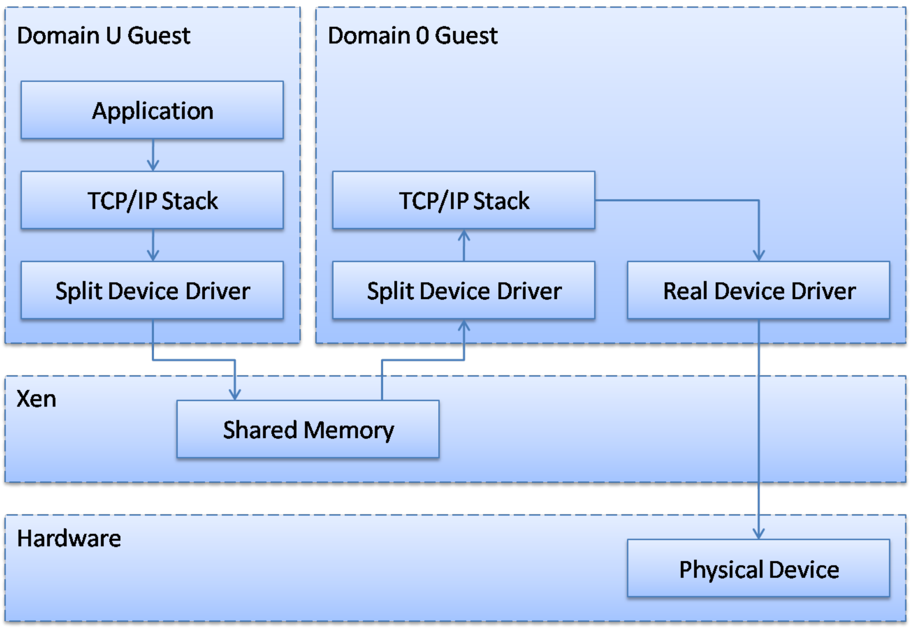


Figure 2.2: The travel path of a packet sent from an unprivileged guest [13]

experiment, which ran four domains concurrently, while two of them ran benchmarks, and the other two ran a disk bandwidth hog and a fork bomb in the background respectively, the two antisocial domains contributed only 4% performance degradation. This overhead comes from extra context switches and cache effects [15].

To obtain high performance and strong resource isolation, the virtualization technique *paravirtualization* is necessary. Paravirtualization is in contrast to *full virtualization*. Full virtualization provide a complete simulation of the underlying hardware, thus allow unmodified operating systems to be hosted without aware that they are being virtualized. This approach brings a number of drawbacks such as cannot see the real hardware, problematic for certain privileged instructions (e.g., traps), no real-time guarantees, and so on, hence results in inferior performance levels compared to paravirtualization [15, 18]. Paravirtualization avoids these drawbacks by presenting a virtual machine abstraction that is similar but not identical to the hardware. This approach provides some exposures to the underlying hardware, thus improves performance. Paravirtualization needs modifications to the operating systems, but no modifications to applications [15].

2.2 Maxine

Conventional virtual machines are written in C or C++, which have drawbacks such as lack of modularity, low-level programming, high potential for hard-to-find bugs, intricate interdependencies, etc. As existing virtual machines have progressed in performance and functionality, their implementations have become unmanageably complex [19].

Maxine [20] is a meta-circular Java Virtual Machine mostly implemented in Java, where “meta-circular” means the VM is written in the same language it executes, that is no VM / application code distinction. It applies Java methodologies and tools (IDEs) to JVM development to avoid the drawbacks caused by the low-level languages, and then makes VM development more productive.

Maxine is an ongoing project that still has many issues such as unimplemented JVM_* C functions, GC bugs, etc.. According to the test results from Sun, the performance of Maxine is slower than HotSpot by a factor of 5 on average. By using SPEC JVM98, Maxine passed all the benchmarks and is 1.6 to 5 times slower than HotSpot; in the tests of DaCapo, Maxine passed 5 out of 12 benchmarks, and is 5 to 10 times slower than HotSpot [19].

2.3 GuestVM

GuestVM builds a complete software stack, implemented entirely in Java and hosted on Xen, with just the thinnest layer of C code in between. GuestVM is lightweight. It only needs 70 megabyte of disk space (63 megabyte for Maxine, 1 megabyte for GUK, and 6 megabyte for GuestVM itself) and additional 72 megabyte for a Java Runtime Environment, whereas a minimal Ubuntu installation, which provides only a command line interface, still needs 750 megabyte, let alone the full operating systems with graphical user interfaces that take up easily a couple of gigabyte. It aims at giving Java applications just what they need. Notice that GuestVM is a server-side project, i.e., headless (headless mode is a system configuration in which the keyboard, mouse or display device is lacking [21]), so it does not support *Abstract Window Toolkit* (*AWT*) in java. Typically, the situation for a production server that is running a big java application is that there is just the operating system and the java application. Basically, it is a one-to-one relationship, although a few other processes related to networking may exists. A single huge application can be spread logically over multiple machines, if one machine is not big enough. In this context, the operating system is just doing one thing - supporting the Java application. Java’s needs are very specific whereas the operating system is very general. Therefore, GuestVM cuts the extraneous stuff to minimize the consumption of server resources and makes better use of the machine.

Conventional Java platform libraries are implemented in C/C++ that brings some issues. For example, crossing layers written in different programming languages at run-time can be expensive; dynamic compiler does not have visibility into all layers; development environment is complicated, and so on. GuestVM gives the potential advantages: improved performance, enhanced developer productivity, better thread management, and easier administration [9]. All because the entire stack is implemented in Java.

GuestVM consist of three parts: Maxine VM that has been introduced in section 2.2, the Guest VM Microkernel (*GUK* for short), and GuestVM itself.

GUK is a derivative of the Xen Minimal OS, Mini-OS, and sits between Xen and the GuestVM Java layer. It is a quite thin (about 850 kilobyte) layer written in C that is used to support the GuestVM Java platform. It is in charge of microkernel thread scheduling, memory management, and block device access in fairly simple ways, while smarts are at the Java level. GUK requires a 64-bit x86 machine and hypervisor.

GuestVM contains four main components: GuestVM, GuestVMNative, JNodeFS, and YANFS. GuestVMNative is a native shim between GuestVM as well as Maxine Java code and the microkernel, that is, simply maps the certain GuestVM and Maxine functions to GUK functions. JNodeFS is an ext2 file system implemented in Java from the JNode (Java New Operating System Design Effort) [22]. YANFS (Yet Another NFS) is a Java implementation of NFS from [23]. GuestVM is the main body of Java code. It provides an implementation of the Java thread scheduler and defines a virtual file system (VFS) interface that corresponds to the native method layer.

2.4 OSGi

The OSGi [24] framework is a Java-based, centralized service-oriented platform that is being widely used as an execution environment for developing extensible applications. The key features of OSGi can be summarized as: modularization, runtime dynamic, and service orientation, which are handled by the module layer, the life cycle layer and the service layer respectively.

The module layer is the foundation of OSGi Framework that defines a modularization model for Java by introducing the concept of bundle, a module unit that explicitly defines its dependencies to other modules and its external API via additional meta information. The dependencies and life cycle of bundles are managed by the life cycle layer that defines how new bundles are added and how existing bundles are started, stopped, updated and removed dynamically at runtime.

The service layer connects bundles in a dynamic way. In the OSGi model, any Java object can be published as a service to be used by other bundles. A service is always defined by a Java interface that is separated from its implementation to make bundles loosely coupled. This approach allows developers to bind to services only through interface, thus the implementation can be exchanged even at runtime. The OSGi framework maintains a central registry for bundles to publish their services, and retrieve and use services provided by others.

Because of the dynamic nature of the OSGi service model, the changes of the state of a service require to be tracked. The typical pattern of service usage is to listen to service events (add / remove / stop / update services) and react, for example, disable a certain bundle when it becomes unavailable. Traditionally, each event sources maintains its own registry of subscribed listeners and sends events to all subscribers as the events take place. The OSGi white pattern [25] provides a simpler and more efficient way by using OSGi service registry. Instead of dynamically track all sources of events, listeners register themselves as a service with the OSGi framework and implicitly acquire a global subscription to all existing and future event sources.

2.5 R-OSGi

R-OSGi [26, 27] is a middleware layer on top of OSGi. It enables the developers to turn the centralized OSGi implementations into distributed applications transparently using the same modularity features of OSGi. It does not change the OSGi framework implementation and makes seamless embedding meaning that existing OSGi applications can be distributed using OSGi without modification. All that a service provider has to do is registering a service for remote access, and then service customers can connect to the service provider to get access.

When a bundle requests a service form a remote peer, R-OSGi retrieves the service in the distributed service registry which is based on SLP [28]. If the service is found, the client generates a local proxy dynamically. The proxy redirects service calls to the original remote service and sends the method response back to the local client. To a client side, the proxies behave as a local service.

R-OSGi communications are messaged-based. By default, a network channel in R-OSGi is a persistent TCP connection, with a lightweight binary protocol over it. To reduce the overhead for TCP handshake, it is kept open as long as traffic exists within the timeout.

R-OSGi is remarkably lightweight and efficiency. It has a small footprint and its performance is comparable to RMI and two orders of magnitude faster than UPnP.

Chapter 3

GuestVM Performance

3.1 Test Setup

All the experiments run on a computer with Intel(R) Core(TM)2 Quad CPU Q9400 @ 2.66Hz and 4GB Memory. The operating system is Ubuntu 2.6.24-4.6-generic with the Linux 2.6.24-27-xen kernel and has Sun's HotSpot JDK_1.6.0_15 installed running in server mode. The number of CPUs allocated to the GuestVM domain is 1; the initial domain memory is 1GB and the maximum domain memory is 2GB; one virtual disk is attached to domain sized 1GB (except in the experiments that read 1GB file from disk, where the virtual disk size is 2GB). The hypervisor layer is Xen-3.2.1. The GuestVM version is 0.2.4, which is identified by guestvm repository version f545552add0e, guk repository version db9dad98b331, and compatible Maxine version cf1c5f6686d8.

3.2 Running SPEC JVM98 Benchmarks

Theoretically, GuestVM has the potential to give better performance because of lightweight and all-Java software stack. But whether there is a performance benefit is an open question. Computer science's intuition about performance is poor. The reasons for this can range from insufficient understanding of the complicated system, to silly coding mistake [29].

Figure 3.1 is from [2]. It presents the comparison of running specJVM98 [30] on different JVMs. From this Figure, Maxine performs excellent and is only slightly slower than Sun's HotSpot [31] JVM. As the groundwork of GuestVM, Maxine is a good prerequisite for performance. Whereas GuestVM is 8 to 9 times slower than Maxine on average and is up to 26 times slower than its fastest opponents (HotSpot JVM server). Its runtime for _200_check and _228_jack failed respectively with an array out of bounds exception and a compilation error.

This experiment has been repeated in this work, but got different measurements from the previous tests. Since GuestVM does not support GUI, all the tests in this work for running SPEC JVM98 benchmarks are carried out from the com-

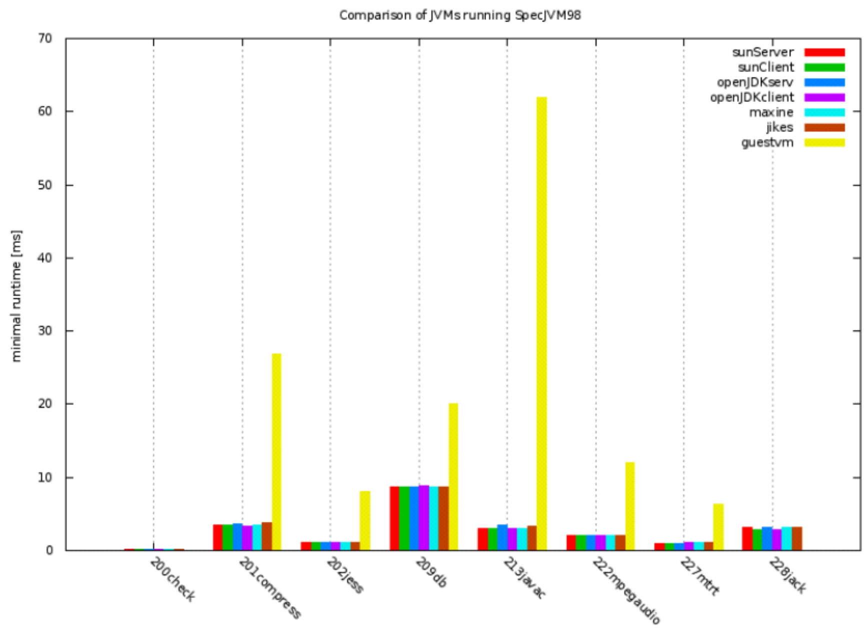


Figure 3.1: specJVM98 on different JVMs [2]

mand line [32], including the ones for HotSpot JVM and Maxine VM, because the results from GUI are not strictly identical to the ones obtained from the command line. Listing 3.1 and Listing 3.2 give the way to run JVM98 benchmarks in Maxine and GuestVM respectively. Figure 3.2 shows the results of running SPEC JVM98 on different JVMs and compared them.

Listing 3.1: Running SPEC JVM98 Benchmarks on Maxine

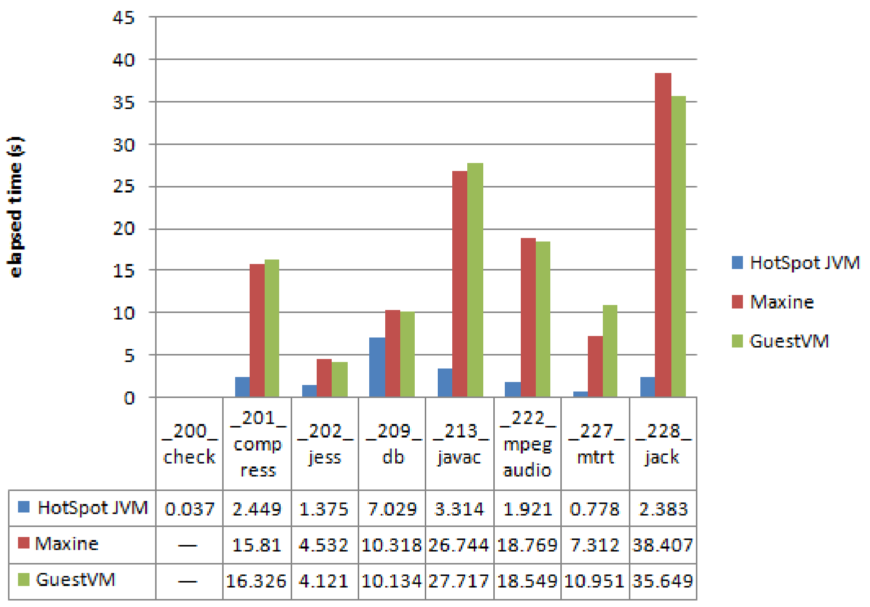
```
# cd $JVM98_DIR
# ./shrc
# $MAXINE_DIR/bin/max vm −cp $JVM98_DIR SpecApplication [size]
    benchmark
```

Listing 3.2: Running SPEC JVM98 Benchmarks on GuestVM

```
# cd $GUESTVM_DIR/GuestVMNative
# bin/run_benchmarks −cp /guestvm/java/jvm98 SpecApplication [size]
     benchmark
```

_200_check is used to verify the validity of the JVM but not used in the performance metrics of the system. It consists of a set of tests to test logical, arithmetic, shift, and so on operations, as well as perform bounds checking on array indexing. Maxine and GuestVM failed it because of an index out of bounds exception.

From the benchmark results, Maxine VM does not perform as well as showed in



	_200_ check	_201_ comp ress	_202_ jess	_209_ db	_213_ javac	_222_ mpeg audio	_227_ mtrt	_228_ jack
🟦 HotSpot JVM	0.037	2.449	1.375	7.029	3.314	1.921	0.778	2.383
🟥 Maxine	—	15.81	4.532	10.318	26.744	18.769	7.312	38.407
🟩 GuestVM	—	16.326	4.121	10.134	27.717	18.549	10.951	35.649

Figure 3.2: specJVM98 on different JVMs

[2]. In the test _227_mtrt, GuestVM is about 50 percent slower than Maxine. _227_mtrt is a ray tracer that creates a scene depicting a dinosaur, and uses a multi-threaded driver, where the threads render the scene from an input file. Except _227_mtrt, the time used by GuestVM and Maxine are very close for running all the other programs in the benchmark suite. In _228_jack, which failed in the tests of [2], GuestVM is even a bit faster than Maxine. More details of the benchmark information are available in [33].

Compared to HotSpot JVM, GuestVM performs 44 percent slower in the best case (_209_db), and 14 times slower at worst (_228_jack). In other cases, GuestVM is 2 to 9 times slower than HotSpot JVM.

SPEC JVM98 Benchmark Suite is used to verify the validity and measure performance of JVMs. More than 70 percent of its instructions are about load and store items on to the stack, alter the stack's contents, and accesses to object fields (memory accesses); more than 15 percent are about arithmetic operations logical statements [33], while it tests nothing about I/O. GuestVM is more than a conventional JVM. It runs directly on hypervisor, thus besides using Maxine to replace the conventional JVM layer, it takes over the role of operating systems as well. For this reason, SPEC JVM98 benchmark is not sufficient to measure GuestVM's performance all around such as larger I/O performance. Therefore, the following sections give a comprehensive analysis of GuestVM performance to figure out which parts of GuestVM pose the performance bottleneck and fix them.

3.3 GuestVM Performance Analysis and Improvement

GuestVM neither provides any monitoring tool as Linux, like *iostat*, *vmstat*, *free*, *uptime*, etc., nor has any profiler, i.e., performance analysis tool. Existing profilers, such as OProfile [34], a system profiler for Linux, are not supported by GuestVM. Since GuestVM plays two roles in the software stack of Macro-Components: as a JVM and as an operating system, to figure out the sources of performance problems, experiments are designed in respect of memory accesses and I/O.

Without specially mentioned, all the experiments in the following sections consist of three parts: set up, run, and tear down, where run is the real experiment to execute, set up is to initialize the required environment, and tear down is to clean the set up, e.g., close the opened file system, and check whether the experiment has been executed successfully. Every experiment needs to be warmed up, that is, run the experiment for multiple times before the measurements being recorded. The final experiment result is a measurement whose standard deviation is less than ten percent of the mean value.

3.3.1 Memory Performance

Memory access performances were evaluated by measuring elapsed time for allocating, reading and writing memory. The task of memory allocation [35] consists of finding a block of unused memory of sufficient size. The dynamic memory allocation algorithm used to organize the memory area and allocate and deallocate chunks can impact performance considerably.

Memory Allocation

The memory allocation time was evaluated by seven sets of data obtained from allocating 1 kilobyte to 200 megabyte data to memory on different JVMs. Allocating more than 300 megabyte data to memory failed for an out of memory error. In Java, memory is allocated only to objects. There is no explicit allocation of memory (such as *malloc*), but only the creation of new objects. Thus, memory allocation is via the following code:

Listing 3.3: Code for Memory Allocation

```
memory = new byte[MEM_SIZE];
```

The conclusions drew from the seven sets of measurements are more or less identical: the time used for memory allocation on Sun's HotSpot JVM, Maxine VM, and GuestVM, is close. GuestVM performs even a bit better than HotSpot JVM in some cases. Figure 3.3 gives two typical examples of the seven results. Figure 3.4 uses logarithmic coordinates to present all the measurements for allocating different size of data to memory when running the tests on different virtual machines.

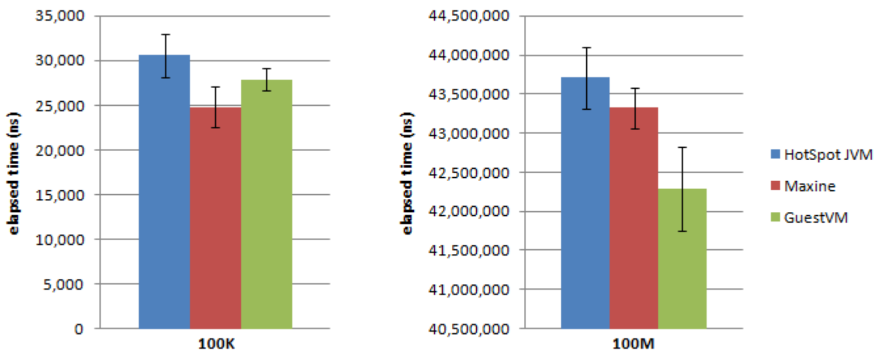
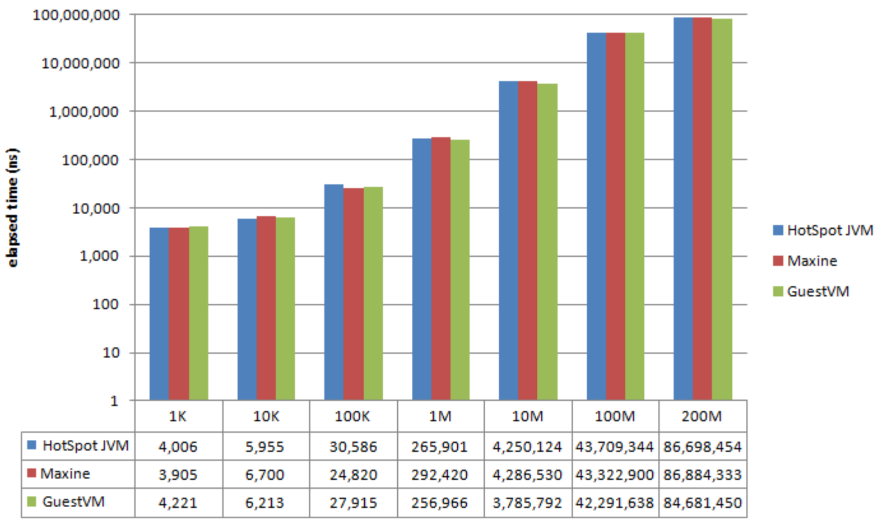


Figure 3.3: Time used for allocating 100KB and 100MB data to memory



	1K	10K	100K	1M	10M	100M	200M
🟦 HotSpot JVM	4,006	5,955	30,586	265,901	4,250,124	43,709,344	86,698,454
🟥 Maxine	3,905	6,700	24,820	292,420	4,286,530	43,322,900	86,884,333
🟩 GuestVM	4,221	6,213	27,915	256,966	3,785,792	42,291,638	84,681,450

Figure 3.4: Time used for allocating different size of data to memory

Memory Read

Memory reading time was evaluated by five sets of measurements obtained from randomly reading 10 kilobyte to 20 megabyte data from memory. The time for reading 1 kilobyte memory is hard to converge. No valid result could be obtained from 20 times tests on HotSpot JVM, where valid means the standard deviation of the measurements is less than ten percent of the mean value. Tests for reading more than 50 megabyte memory failed with out of memory errors. The memory reading performance was tested by the code in Listing 3.4, where the variable *indices* is an array holding *MEM_SIZE* random integers ranged from 0 to *MEM_SIZE*. The variable *memory* and *data* are declared in two ways: common declaration without the keyword *volatile* and with *volatile* (Shown in Listing 3.5). The variable *memory* is stored in the heap as a byte array; whereas the variable *data* is a primitive type that holds the value directly instead of a reference, and it is stored on the stack [36]. Because stack that has direct support from the processor via its stack pointer is extremely fast and efficient for storage access, variable *data* as a primitive type can minimize the influence of memory writing in the tests for memory reading. Figure 3.5 presents two specimens of the five sets of measurements. Figure 3.6 uses logarithmic coordinates to show all the results for randomly reading different size of memory data on HotSpot JVM, Maxine and GuestVM.

Listing 3.4: Code for Reading Memory

```
for (int i = 0; i < MEM_SIZE; i++) {
    data = memory[indices[i]];
}
```

Listing 3.5: Variable Declaration in Two Ways

```
private byte[] memory;
private byte data;

OR

private volatile byte[] memory;
private volatile byte data;
```

From Figure 3.5 and 3.6, the memory reading performance of HotSpot JVM can be degraded by the keyword *volatile* by a factor of 4 to 7. Whereas the time used by Maxine and GuestVM is very close and is completely not affected by *volatile*. Without the impact of *volatile*, HotSpot JVM performs up to 14 times faster than Maxine and GuestVM, and 4.5 times on average as the memory data to read is larger than 1 megabyte. In the tests declaring memory and data variables *volatile*, HotSpot JVM can perform at most 2 times as fast as Maxine and GuestVM; as the size of the memory to read growing, the performance advantage of HotSpot declines, and in the cases reading more than 10 megabyte memory data, Maxine and GuestVM can be even 10 percent faster than HotSpot.

Declaring a volatile Java field means that all reads and writes will go straight to main memory, instead of using a cache value from working memory. Java Main

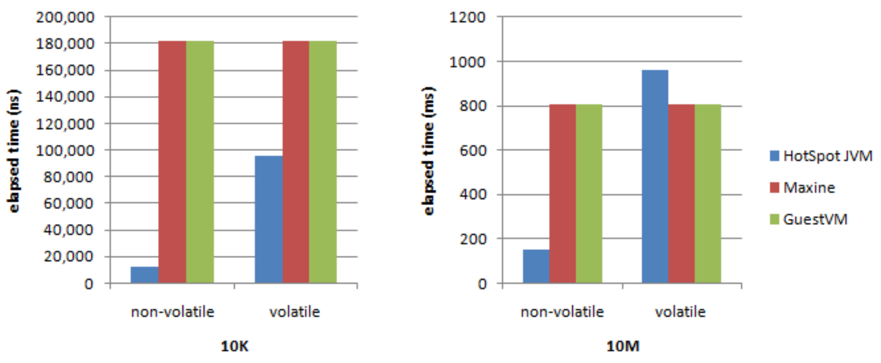
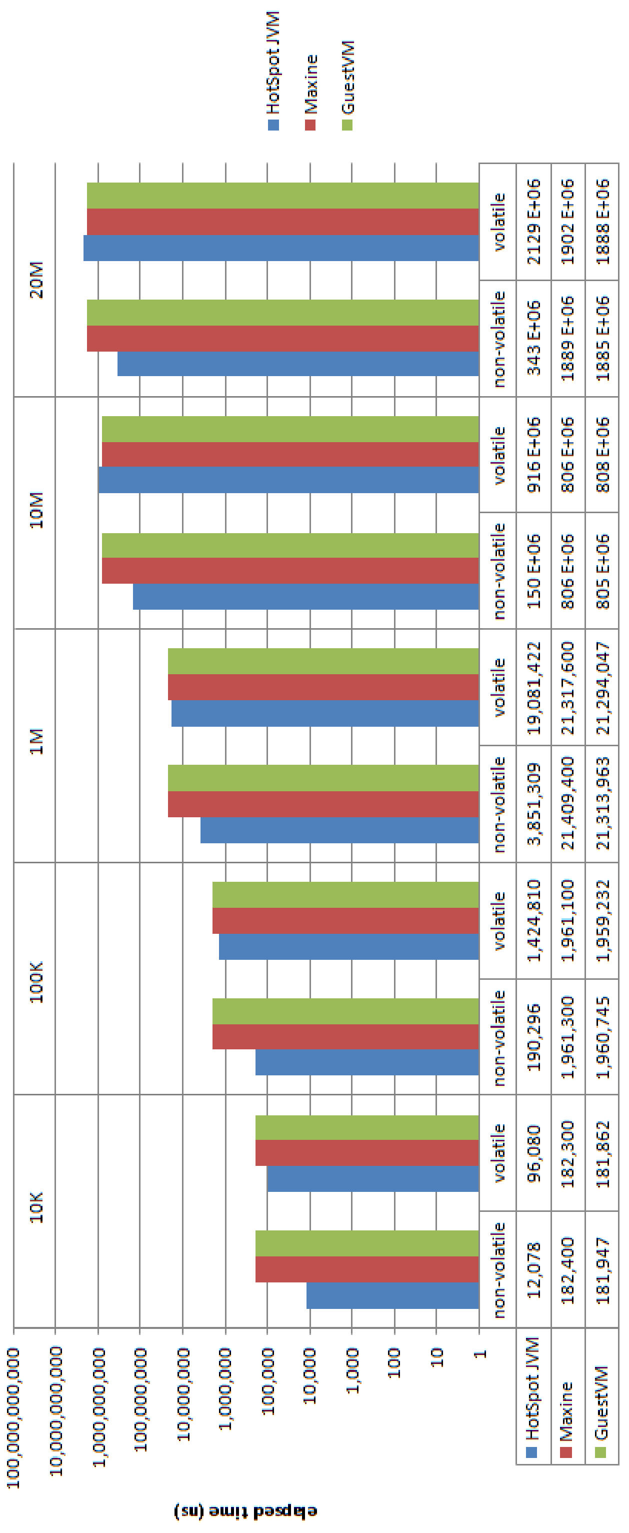


Figure 3.5: Time used for randomly reading 10KB and 10MB data from memory

Memory (also known as Java Heap Memory) is the memory area used by JVM for dynamic memory allocation. All variables and objects reside in the main memory and are shared between all threads. To optimize performance, every thread has a working memory (an abstraction of caches and registers), in which it may hold its own working copy of variables. As a thread executes a program, it operates on these working copies rather than accessing the main memory [37].

JVM (specifically HotSpot) uses *memory barrier* instruction to prevent the volatile variables from storing in working memory. When applied to a field, the Java volatile guarantees that two different threads always see the same value of a certain variable at any moment, but at the same time, it gets rid of the optimization of JVM and adds overhead caused by flushing cache and writing data back to memory each time after the values of variables are modified [37, 38].

The memory reading time used by Maxine and GuestVM is not influenced by the keyword *volatile*. This means it is not supported by Maxine to raise efficiency through using working memory as cache for each thread. The reason is that Maxine uses read/write barriers when accessing memory except garbage collector (GC). Maxine defines *Reference* as well as *Grip* as a runtime value of type "java.lang.Object". They are almost the same except that *Reference* access operations may incur barriers. The mutator refers to objects and parts thereof by using *Reference*, whereas the GC uses grips instead to avoid barrier [19].



	10K non-volatile	10K volatile	100K non-volatile	100K volatile	1M non-volatile	1M volatile	10M non-volatile	10M volatile	20M non-volatile	20M volatile
🟦 HotSpot JVM	12,078	96,080	190,296	1,424,810	3,851,309	19,081,422	150 E+06	916 E+06	343 E+06	2129 E+06
🟥 Maxine	182,400	182,300	1,961,300	1,961,100	21,409,400	21,317,600	806 E+06	806 E+06	1889 E+06	1902 E+06
🟩 GuestVM	181,947	181,862	1,960,745	1,959,232	21,313,963	21,294,047	805 E+06	808 E+06	1885 E+06	1888 E+06

Figure 3.6: Time used for randomly reading different size of data from memory

Memory Write

The design for memory writing experiments is similar to that of the memory reading ones. The memory writing performance was tested via the code in Listing 3.6, where the variable *indices* is an array holding random integers and the variable *data* is a random byte. Variable declarations are the same as memory reading experiments shown in Listing 3.3. The size of the memory to write ranges from 10 kilobytes to 20 megabytes. The time for writing 1 kilobyte memory is difficult to converge as reading, and writing more than 50 megabyte memory failed because of out of memory. Figure 3.7 shows two representative examples from five set of measurements for randomly writing different sizes of data to memory on different JVMs, and Figure 3.8 presents all the results in logarithmic coordinates.

Listing 3.6: Code for Writing Memory

```
for (int i = 0; i < MEM_SIZE; i++) {
    memory[indices[i]] = data;
}
```

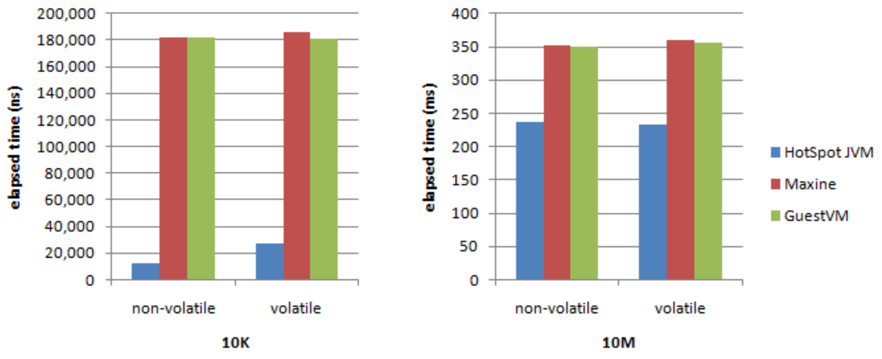
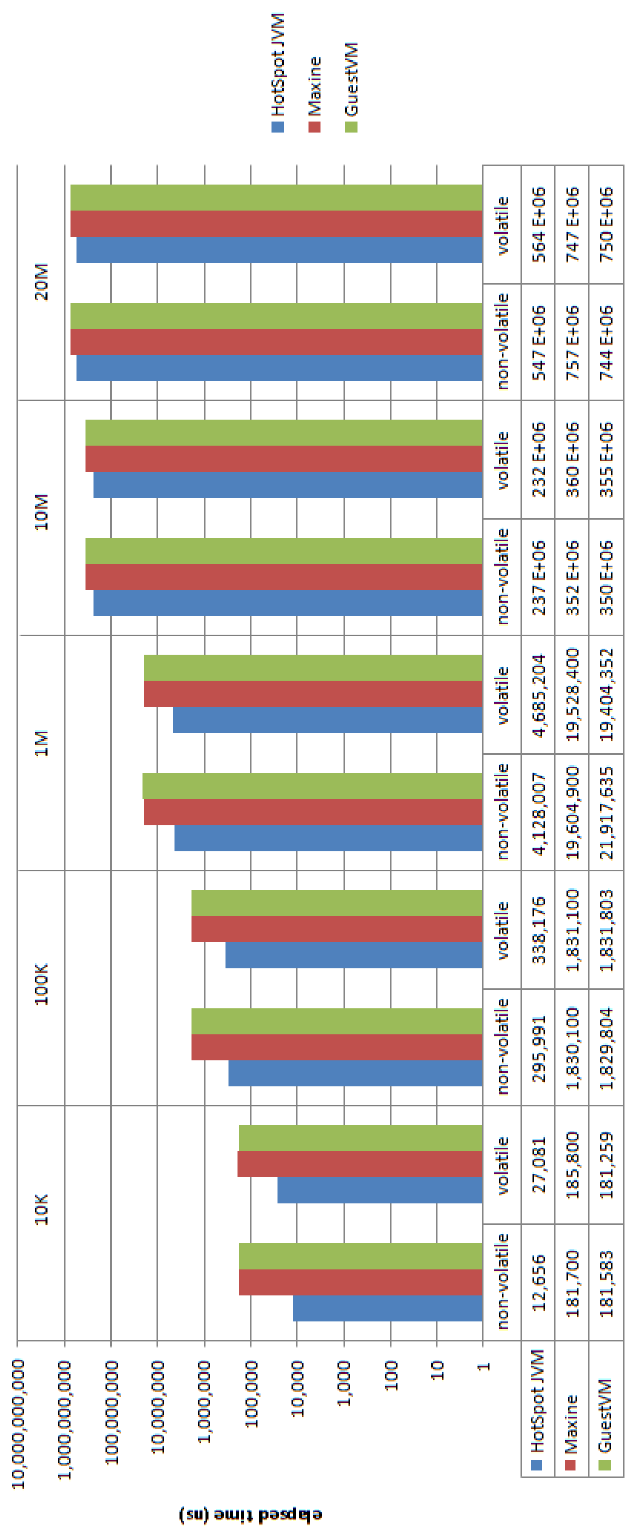


Figure 3.7: Time used for randomly writing 10KB and 10MB data from memory



	10K		100K		1M		10M		20M	
	non-volatile	volatile	non-volatile	volatile	non-volatile	volatile	non-volatile	volatile	non-volatile	volatile
🟦 HotSpot JVM	12,656	27,081	295,991	338,176	4,128,007	4,685,204	237 E+06	232 E+06	547 E+06	564 E+06
🟥 Maxine	181,700	185,800	1,830,100	1,831,100	19,604,900	19,528,400	352 E+06	360 E+06	757 E+06	747 E+06
🟩 GuestVM	181,583	181,259	1,829,804	1,831,803	21,917,635	19,404,352	350 E+06	355 E+06	744 E+06	750 E+06

Figure 3.8: Time used for randomly writing different size of data to memory

Figure 3.7 and 3.8 shows that the memory writing performance of Maxine and GuestVM are very close as memory reading. In respect of HotSpot JVM, the impact of volatile on writing is less than on reading. HotSpot using volatile is about twice as slow as without using it as writing a small amount of data (10 kilobyte); when writing larger size that more than 1 megabyte, volatile hardly enhances memory writing performance. Maxine and GuestVM are up to 7 times as slow as HotSpot JVM without the influence of working memory. As the size of memory to write increasing, the gap can be decreased to 1.5 times.

To sum up, in respect of accessing memory, GuestVM performs extremely close to Maxine. GuestVM builds on Maxine, uses Maxine's implementation of the Java memory model, and does not add overhead to Maxine in terms of memory access. Without considering the optimization of working memory (i.e., using *volatile* to remove the influence of cache), GuestVM performs practically as excellent as HotSpot JVM on average in terms of allocating and reading memory. Regarding writing memory, GuestVM is 50% times slower than HotSpot JVM at best.

3.3.2 I/O Performance

The experiments for reading disk use Java *FileInputStream* to read the contents of a file as a stream of bytes. The performance is evaluated from eight sets of experiments that read different sizes of files ranged from 1 kilobyte to 1 gigabyte from disk. All the results are shown in Figure 3.10 using logarithmic coordinates. To make the illustration clear, Figure 3.9 gives two sets of typical results. The experiment for reading 10 megabyte data running on GuestVM did not converge in 20 times execution, where converge means the standard deviation is no larger than 10 percent.

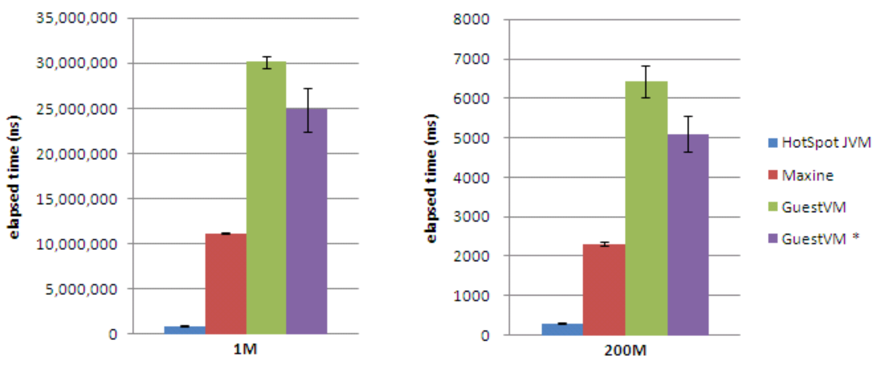
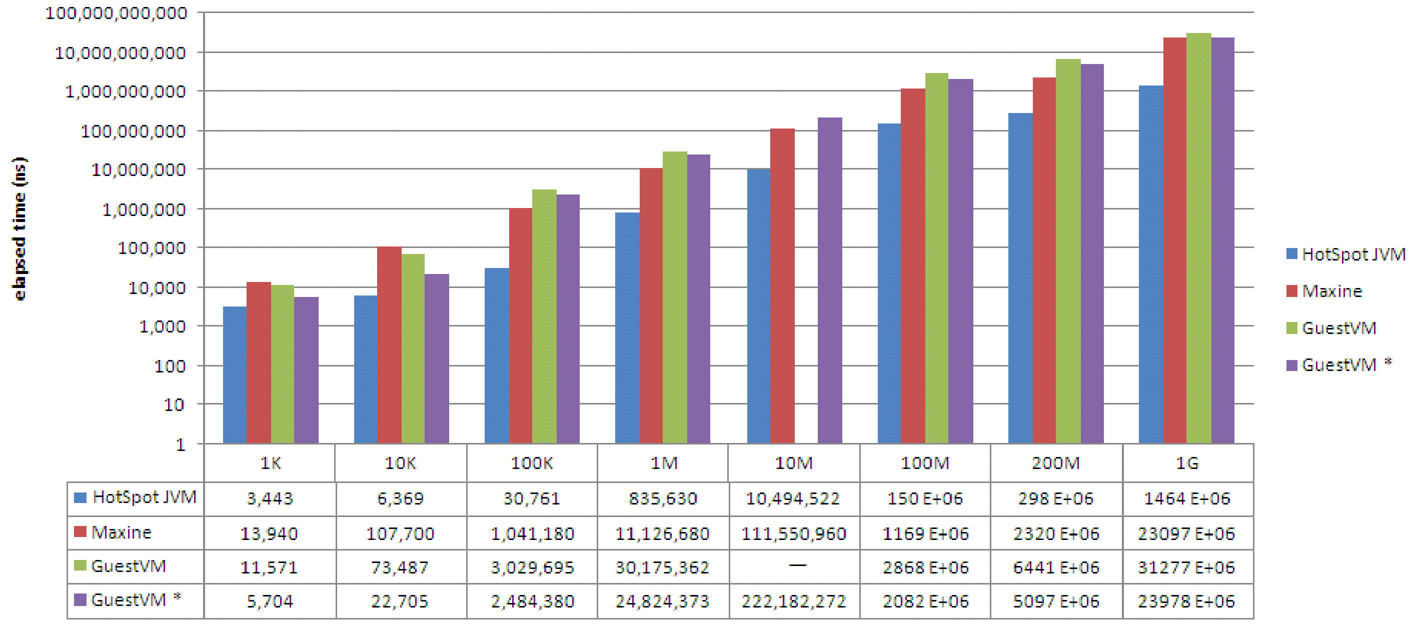


Figure 3.9: Time used for reading files sized 1MB and 200MB from disk



	1K	10K	100K	1M	10M	100M	200M	1G
🟦 HotSpot JVM	3,443	6,369	30,761	835,630	10,494,522	150 E+06	298 E+06	1464 E+06
🟥 Maxine	13,940	107,700	1,041,180	11,126,680	111,550,960	1169 E+06	2320 E+06	23097 E+06
🟩 GuestVM	11,571	73,487	3,029,695	30,175,362	—	2868 E+06	6441 E+06	31277 E+06
🟪 GuestVM *	5,704	22,705	2,484,380	24,824,373	222,182,272	2082 E+06	5097 E+06	23978 E+06

Figure 3.10: Time used for reading files of different sizes from disk

From the results of experiments, the I/O performance of GuestVM is very poor. GuestVM performs 2.5 times slower at best for reading a file of 1 kilobyte and 100 times slower in the worst case for reading a file of 100 kilobyte. On average, GuestVM performs slower than HotSpot JVM by a factor of 25, while Maxine performs slower approximately by a factor of 10. The time used by Maxine is proportional to the size of a file to read; the time used by GuestVM is proportional to the file size when the size is larger than 100 kilobyte; whereas the performance of HotSpot JVM decreases as the size of data to read increasing except for the files larger than 100 megabyte. The advantage of HotSpot for reading small size of data from disk could benefit from a more sophisticated implementation of cache hierarchy in Java memory model.

Reclamation of file space is a future work of GuestVM. The delete operation can only remove the file entry, but cannot release the space. Hence, the tests that may runs tens of thousands of times to get a reasonable value cannot be handled when writing large size data.

Based on the software stack comparison between classic and GuestVM as shown in Figure 2.1, the performance overhead may reside in Maxine, GuestVM file system, and / or Guest VM Microkernel (GUK).

Maxine is a replacement of HotSpot JVM on Linux. In the experiments, they were both running on the same operating system. But Maxine is about 10 times slower for the disk reading. If the I/O implementation of GuestVM is based on Maxine, Maxine is not a guaranteed prerequisite for the good performance of GuestVM. From Figure 3.10, when reading a small file that is no larger than 10 kilobyte, GuestVM can perform even better than Maxine. This could benefit from the smaller software stack of GuestVM compared to Maxine, or the GuestVM's I/O could be independent of Maxine. Actually, GuestVM has re-implemented the native methods that depend on the VM details. By tracing the call tree of reading disk, the procedure that consists of locating the data to read and fetching them from the disk to a read buffer has nothing to do with Maxine. The only thing Maxine involved is to copy the data in read buffer to a return array. Thus, the Maxine's impact on the I/O performance of GuestVM is very limited.

GuestVM is a bare-metal JVM that runs directly on Xen's hypervisor without requiring an operating system. The role as an operating system is taken over by GuestVM Microkernel. Operating system performance could be affected by multiple factors, for instance, the CPU scheduler, the virtual memory manager and so on [39], whereas there is hardly any profiling or performance monitoring tool for analyzing or tuning these subsystems in GuestVM, thus *RDTSC* (read time-stamp counter) [40] instruction was introduced to GUK to trace the CPU clocks used by each set of statements on the path for block reads, and then to locate the sources of performance problems. By doing so, no suspicious part that may bring large amounts of overhead was found. Because GUK is a minimal operating system kernel that handles the block device access in a quite simple way, smarts are at the Java level.

Therefore, by using exclusive method, the performance bottleneck is likely to reside in GuestVM file system that is implemented in Java. GuestVM defines a Virtual File System (VFS) interface that corresponds to the native method layer, and borrows the ext2 file system implementations from JNode. The main aspects of performance tuning include code optimization and caching strategy [41]. From the comparisons between GuestVM and HotSpot JVM shown in Figure 3.10, GuestVM is only 2.5 times slower than HotSpot when reading a file sized 1 kilobyte, but performs rather poor for reading large files. This could result from caching strategy. More analysis and experiments are presented in the following sections.

Code Optimization

To figure out which portions of code have potential to be optimized, *System.nanoTime()* statements are injected to the file system to trace which parts are time consuming. It was found that to fit the file system interface of JNode, GuestVM copies each data block twice when reading, from a file to a byte buffer and then from the byte buffer to a byte array. The performance was improved by modifying the interface and copying data from the file to a byte array directly without copying to the intermediate byte buffer.

Figure 3.10 shows the improvement after reducing one time data copy, where "GuestVM *" is the improved version. When reading a small file no larger than 10 kilobyte, the performance can be improved by up to 3 times compared to the original GuestVM. When reading a file larger than 100 kilobyte, the elapsed time is reduced by 25 percent on average. However, GuestVM is still close to 20 times slower than HotSpot JVM. The following experiments are based on this enhanced GuestVM, and "GuestVM" means this improved version in the next sections without specially mentioned.

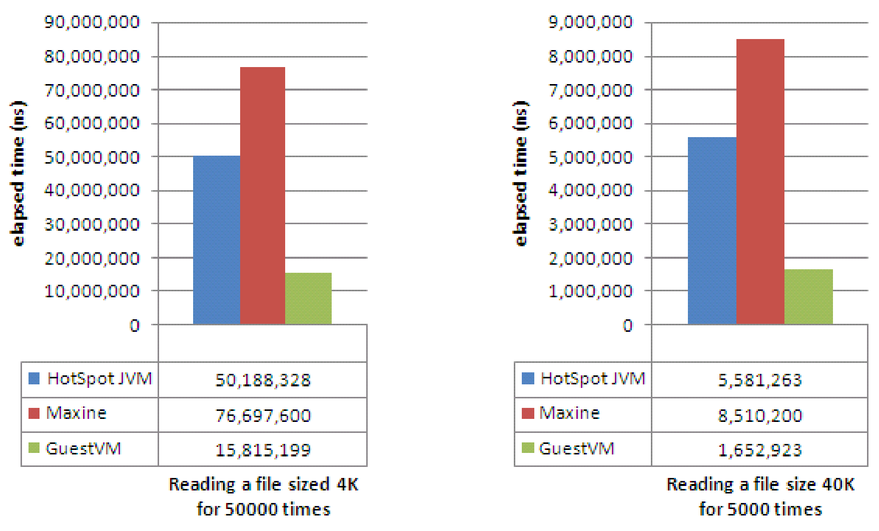
Cache Strategy

For efficiency, Linux file systems support three caches for inodes, directory entries, and data blocks respectively. GuestVM file system maintains two of them, the inode cache for most-recently used inodes, and the buffer cache for most-recently requested data.

The inode cache of GuestVM is implemented as a hash table, which is an object inheriting from *java.util.Hashtable*. The buffer cache is managed as a least recently used (LRU) hash map, which is implemented by using *java.util.LinkedHashMap* [42]. One feature of LinkedHashMap is to maintain the iteration order based on access-order. It upholds a doubly-linked list running through all of its entries. When an item is added to the cache, and every time it is accessed after that, its entry is moved to the head of the list. This kind of map also includes a way to remove the entry at the tail of the list automatically when the cache is full. This makes it well suited for building LRU caches.

Like most modern operating systems, the buffer cache of GuestVM buffers data as blocks, which are the smallest units of disk I/O. The block size is 4 kilobyte. The cache size is fixed at 10 blocks (i.e., 40 kilobyte). The cache mechanism of Linux is more sophisticated. To make the efficient use of memory, Linux automatically allocate all free RAM for buffer cache, which could be hundreds megabytes or even more than gigabytes. When other applications need more memory, Linux makes the cache smaller automatically [43]. When reading large amounts of data, e.g. 200 megabyte, from disk in Linux, all the data is cached in memory after warming up. This can be observed by tracking the change of memory usage with monitoring tool *free* in Linux. That is, the measurement, e.g. 298 ms for reading 200 megabyte shown in Figure 3.10, is actually the time used to read from memory. Since the effectiveness of a cache is principally decided by its size [44], compared to Linux, the small and poor buffer cache of GuestVM is a main reason of bad performance. In GuestVM, when repeatedly reading a file larger than 40 kilobyte, every block is read from disk, cached in memory, and flushed from the cache before it is reused when new block comes. In such a case, the elapsed time shown in Figure 3.10, excluding the cases of reading a file sized 1 and 10 kilobyte in GuestVM, all comes from disk reading; the small cache is next to useless.

To verify this, Figure 3.11 presents the time used for reading a file size 4 kilobyte for 50000 times and a file sized 40 kilobyte for 5000 times. As the reading file size is no larger than GuestVM's buffer cache size, after warming up, all the data to read are in buffer cache. In this way, the influence of insufficient buffer cache and prefetching (which will be discussed in the next section) is eliminated. From the results, GuestVM performs 2.3 times faster than HotSpot JVM and 4 times faster than Maxine on average, which is excellent. This could benefit from its minimal and all-Java software stack.



	Reading a file sized 4K for 50000 times
🟦 HotSpot JVM	50,188,328
🟥 Maxine	76,697,600
🟩 GuestVM	15,815,199

	Reading a file size 40K for 5000 times
🟦 HotSpot JVM	5,581,263
🟥 Maxine	8,510,200
🟩 GuestVM	1,652,923

Figure 3.11: Time used for reading small size of files repeatly

To further verify the influence of cache, the buffer cache size was increased to 220 megabyte, which is large enough for holding a file no larger than 200 megabyte after warming up as Linux. Based on the previous analysis, the read performance should be greatly improved. Figure 3.13 presents the time used to read different sizes of files in GuestVM with sufficient buffer cache, and compares them to Maxine and HotSpot JVM. To make the comparison clear, Figure 3.12 gives two typical examples from the five sets of measurements.

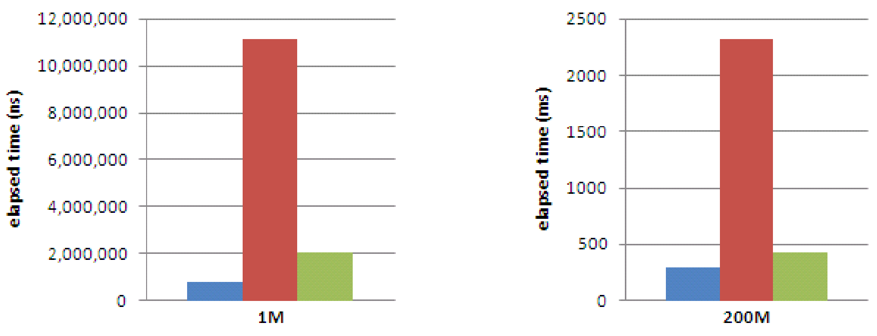
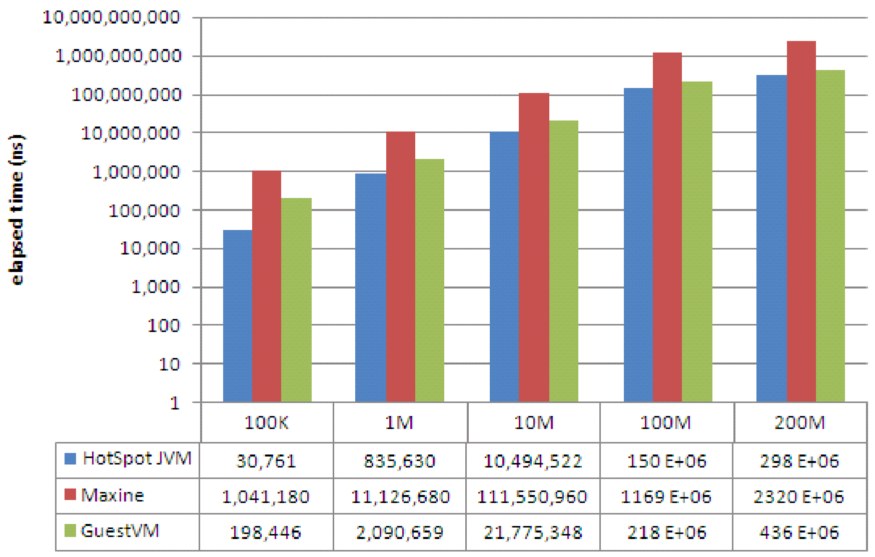


Figure 3.12: Time used for reading files sized 1MB and 200MB with enough buffer cache



	100K	1M	10M	100M	200M
🟦 HotSpot JVM	30,761	835,630	10,494,522	150 E+06	298 E+06
🟥 Maxine	1,041,180	11,126,680	111,550,960	1169 E+06	2320 E+06
🟩 GuestVM	198,446	2,090,659	21,775,348	218 E+06	436 E+06

Figure 3.13: Time used for reading different sizes of files with enough buffer cache

From Figure 3.12 and 3.13, GuestVM is once slower than HotSpot on average, whereas 4 times faster than Maxine. Its performance advantage increases as the file size growing. For the file sized larger than 100 megabyte, GuestVM is only 1.5 times slower compared to HotSpot JVM.

In the case of sequentially accessing a file, the buffer cache is useless unless the cache is large enough for caching the entire file. Otherwise, the blocks in cache are always swapped out before being reused by new ones. However, in the case of randomly access a file, the performance can be improved as the size of buffer cache increasing, even though it is not sufficient to hold the whole file. Figure 3.14 shows the measurements from the experiments running on GuestVM that randomly reads a file sized 5 megabyte with different sizes of buffer cache ranged from 40 kilobyte (the original buffer cache size of GuestVM) to 8 megabyte (that is able to holding the entire file), where "GuestVM-N" means GuestVM with a file system buffer cache sized N. They were tested through the code in Listing 3.7. In this experiment, the overhead introduced by line 4 to generate a random offset to read from is eliminated by subtracting the time used for running the code of line 4 for block number times, which is 31'897, 86'300, and 84'367 (ns) respectively for HotSpot JVM, Maxine, and GuestVM.

Listing 3.7: Code for Randomly Reading a File

```
1  byte[] data = new byte[BLOCK_SIZE];
2
3  for (int i = 0; i < DATA_SIZE / BLOCK_SIZE; i++) {
4      offset = random.nextInt(DATA_SIZE - BLOCK_SIZE);
5      file.seek(offset);
6      file.read(data);
7  }
```

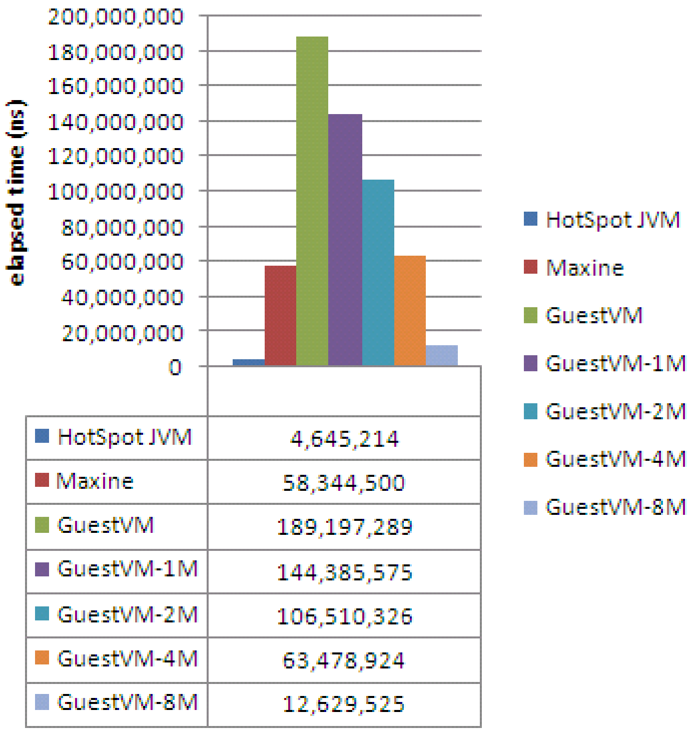
From Figure 3.14, the randomly reading performance of GuestVM increases as the size of buffer cache growing. By giving enough cache, it can be improved by 15 times compared to its original size (40 kilobyte), and even 4.6 times faster than Maxine, but still 2.7 times slower than HotSpot.

GuestVM file system misses the directory cache, which is used to speed up accesses to commonly used directories. In Linux, as directories are looked up by a file system (e.g., ext2), their details are added to the directory cache. The next time when the same directory is used, for example to list it or open a file within it, it will be found in the cache. Missing directory cache could slow down the creation of files.

Prefetching

Prefetching is a significant technique for improving disk I/O performance. Consider a program sequentially accessing a file, caching is not sufficient to explore spatial locality. The basic idea for prefetching in Linux is to read up to 64 blocks ahead when it detects long sequential runs. Prefetching can hide I/O latency by fetching data into cache before they are requested by programs. Notice that prefetching expects each file is continuously stored on the physical disk, i.e., there is no severe fragmentation. This is guaranteed by Linux file system as long as the storage space is not overused [45].

GuestVM does not supply any means for reading ahead. It reads from disk block by block. By tracing the outputs of the number sequence of the blocks to read in the process of disk reading, the file data blocks and the inode blocks



🟦 HotSpot JVM	4,645,214
🟥 Maxine	58,344,500
🟩 GuestVM	189,197,289
🟪 GuestVM-1M	144,385,575
🟦 GuestVM-2M	106,510,326
🟧 GuestVM-4M	63,478,924
🟦 GuestVM-8M	12,629,525

Figure 3.14: Time used for randomly reading a file sized 5MB with different sizes of buffer cache

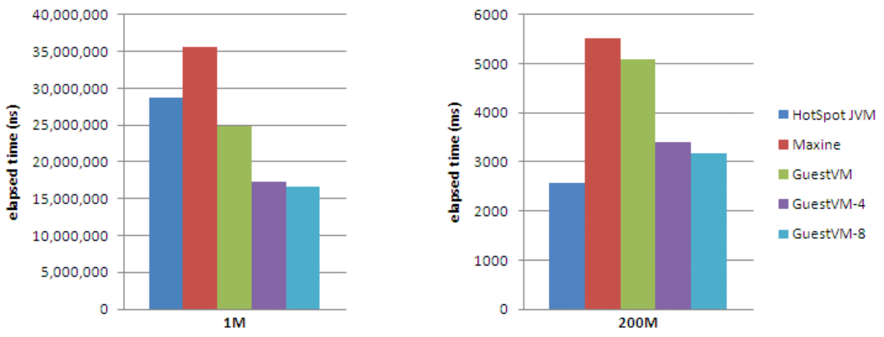
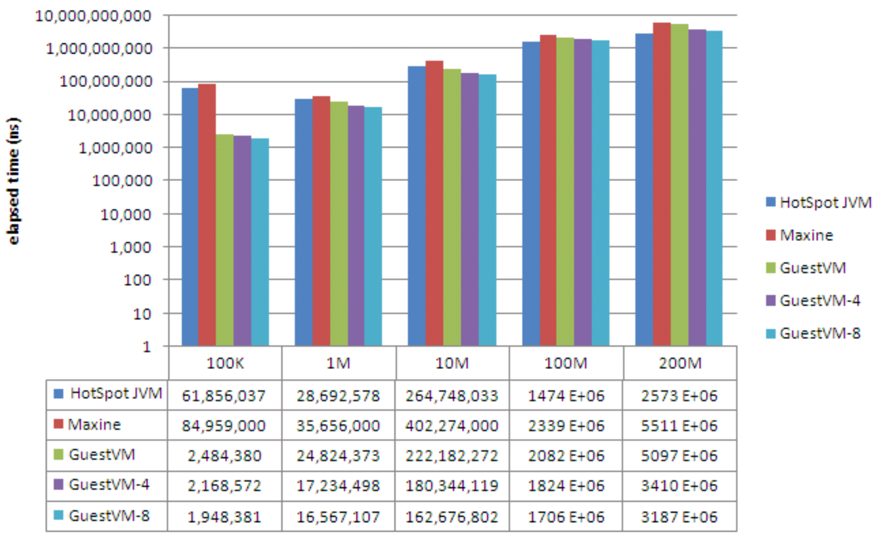


Figure 3.15: Time used for reading files sized 1MB and 200MB with prefetching

that map the inode number of a block to an absolute block number in the file system are read alternatively. This means even there is only one application running in the virtual machine that sequentially reads a file, the blocks to read are not accessed in sequential positions on the disk. That is, the access time is spent much on seek time and rotational latency, the proportion of transfer time is small, thus the disk throughput is low. Reading ahead is to use large I/O instead of small I/O to increase the efficiency and throughput of the disk.

This work gives a simple implementation of prefetching in GuestVM file system part. But it can only read up to 8 blocks ahead because of a disk access bug in GuestVM. The native read in GUK does not guarantee the validity for reading more than one block size of data in a single I/O. To verify the correctness of the implementation of prefetching, the GuestVM file system was extracted from GuestVM, and the file reading tests that use GuestVM file system but without going through GUK have been passed successfully. However, the experiments that read up to 8 blocks ahead were given to evaluate the performance improvement brought by prefetching (as shown in Figure 3.16, and a clear representation in Figure 3.15, where "GuestVM-N" means reading N blocks ahead), and compared it to HotSpot JVM and Maxine. The measurements of HotSpot JVM and Maxine came from reading a file with prefetching mechanism in Linux, but without file system caching. The influence of Linux page cache was removed by using *echo 1 > /proc/sys/vm/drop_caches* before each time reading. While GuestVM uses its original buffer cache, i.e., 40 kilobyte, which is next to useless in this case.



	100K	1M	10M	100M	200M
🟦 HotSpot JVM	61,856,037	28,692,578	264,748,033	1474 E+06	2573 E+06
🟥 Maxine	84,959,000	35,656,000	402,274,000	2339 E+06	5511 E+06
🟩 GuestVM	2,484,380	24,824,373	222,182,272	2082 E+06	5097 E+06
🟪 GuestVM-4	2,168,572	17,234,498	180,344,119	1824 E+06	3410 E+06
🟦 GuestVM-8	1,948,381	16,567,107	162,676,802	1706 E+06	3187 E+06

Figure 3.16: Time used for reading a file with prefetching

From Figure 3.15 and 3.16, the reading performance on average can be improved by 30 and 40 percent respectively by prefetching 4 and 8 blocks compared to GuestVM without prefetching. The time used to read 100 kilobyte data directly from disk by HotSpot JVM is quite large, about 2000 times more than from caching. Without consider this particular case, GuestVM performs approximately 25 percent faster than HotSpot JVM on average.

Chapter 4

Managing MacroComponents

One crucial principle to design MacroComponents is reducing the overhead introduced by virtualization by minimizing software stack. Chapter 3 improved the performance of GuestVM to make it a possible solution for MacroComponents. The architecture of the MacroComponents in this work is shown in Figure 4.1. GuestVMs sit directly between Xen hypervisor and OSGi applications to play a role as operating systems as well as JVMs. GuestVMs run as Domain U, whereas Linux runs as privileged Domain 0. The inter-domain communications are expected to be handled by R-OSGi. To run such MacroComponents, this Chapter describes a design for efficient communications between GuestVM domains by using shared memory as connection channels, and then presents a monitoring and management GUI tool to help developers to manipulate the MacroComponents in a graphical way.

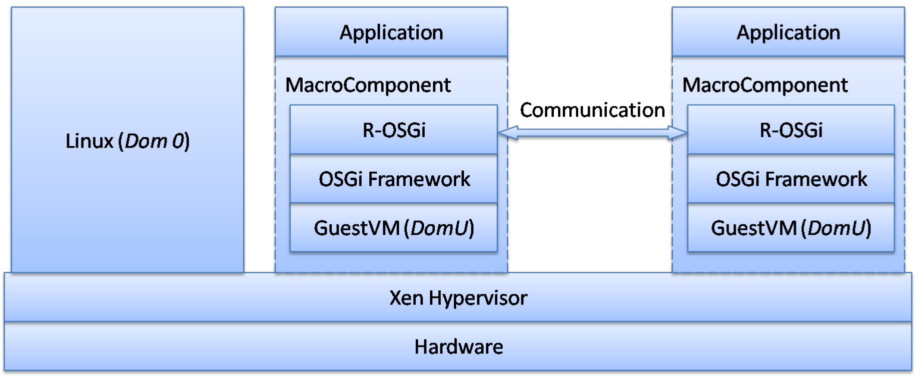


Figure 4.1: Architecture of MacroComponents

4.1 Design for Communication between GuestVM Domains

Performance isolation and lightweight runtime can be achieved by running several MacroComponents in parallel directly on Xen hypervisor, where MacroComponent, in this work, is GuestVM together with the applications running on it. With isolation, however, there are no more direct ways for the components to communicate. Basically, communication between domains is via shared medium, usually is a network or shared memory [5]. GuestVM provides a complete TCP/IP network stack. However, communication through network stack is expensive because Xen virtual network results in low throughput at max CPU usage caused by Xen hypercalls and network stack, where a packet sent from Dom U1 to Dom U2 needs to go through the TCP-stack twice (Dom U1 $\rightarrow$ Dom 0 $\rightarrow$ Dom U2) [46]. Shared memory is an alternative efficient approach for interdomain communication simply by creating overlapping address spaces [2].

Xen provides a grant table mechanism that allows memory pages to be transferred or shared between virtual machines. Each domain has its own grant table shared with Xen. Entities in the table are identified by grant references [13].

To share memory between GuestVMs, two components are involved: one offers the pages, whereas the other maps it. To offer the shared memory, the sender guest domain creates a grant table, and fills it with the domain ID of the domain being granted foreign privileges and the address of the shared memory. Then the receiver domain maps page frames associated with a given grant reference, domain pair to its own address space, and then use the shared pages via a grant handler. Notice that before memory mapping, the grant reference and the sender domain ID need to be communicated via an out-of-band mechanism, XenStore. XenStore is an information storage space shared between Xen guests, maintained by Domain 0 and accessed through a shared memory page and an event channel. It is not meant for large data transfers [47]. By now, the basic channel for efficiently exchanging data has been established.

To achieve asynchronous communication, the shared memory area can be designed as a ring structure, as Xen *I/O Ring*. Since the communication structure of R-OSGi is messaged-based, the ring buffers can be used as a queue for exchanged messages. One domain places a request message in the ring, whereas the other removes it and inserts a response message. Basically, five components are involved in a ring: start and end pointers for producer and customer, and the buffer itself. *Event channels*, which are the standard mechanism for asynchronous notifications within Xen, need to be used to signal that data is available [13].

Network channels in R-OSGi are persistent TCP connections. To use shared memory for communication between domains via R-OSGi, the TCP connection channel (TCPChannelFactory class) should be replaced with a shared memory connection channel. The low level code that use Xen grant table for memory sharing is written in C, thus a shared library file from the native code should be created and integrated in GuestVM's GUK kernel, and invoked via JNI calls in R-OSGi.

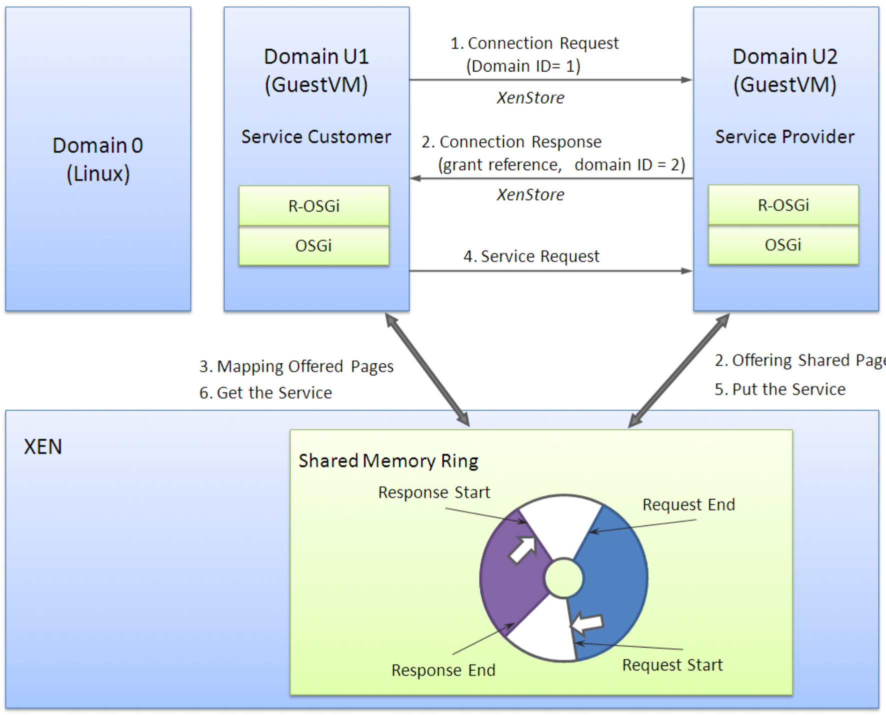


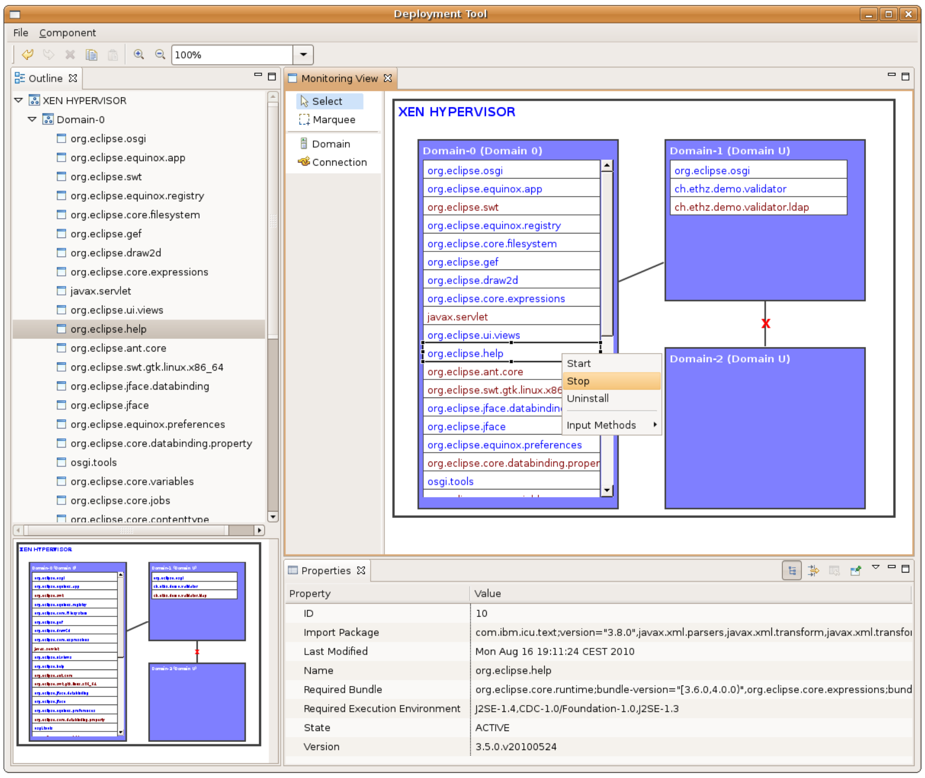
Figure 4.2: GuestVM Inter Domain Communication

To sum up, the communication between GuestVM domains can consist of the following steps (as shown in Figure 4.2):

1. The service customer sends a connection request via XenStore to the service provider with its domain ID;
2. The service provider offers a piece of memory for sharing, which could be in a ring structure, and sends a response with the related information for communication, including grant reference and its domain ID;
3. The service customer maps the shared memory to its domain's address space (that is, the pseudo-physical address), then the connection has been established;
4. The service customer sends a service request with the service identifier;
5. The service provider delivers the requested service to the shared memory, and an event is trigged to deliver a notification to tell the service customer that there is a response waiting;
6. The service customer fetches the service and closes the connection;
7. The service provider closes the connection and unmaps the shared pages.

4.2 MacroComponents Monitoring And Management GUI Tool

Multiple MacroComponents as virtual machines could be running concurrently on a physical machine to isolate performance. The only way for monitoring the domains and managing the applications running on them is through terminals. This is difficult especially when multiple domains are involved. Developers have no sense of the overall picture of the system as well as the network communication among the components. The MacroComponent monitoring and managing GUI tool has been written as a Rich Client Platform (RCP) [48] Eclipse plug-in that provides a graphical overview and management of multiple MacroComponents running on a hardware hypervisor based on Eclipse's Graphical Editing Framework (GEF) (as shown in Figure 4.3) [49]. It has a friendly user interface and can be used to visualize the structure of a MacroComponent system, to monitoring the properties of components, to install new OSGi bundles, and start, stop, uninstall the existing bundles.



Property	Value
ID	10
Import Package	com.ibm.icu.text;version="3.8.0",javax.xml.parsers,javax.xml.transform,javax.xml.transfoı
Last Modified	Mon Aug 16 19:11:24 CEST 2010
Name	org.eclipse.help
Required Bundle	org.eclipse.core.runtime;bundle-version="[3.6.0,4.0.0)",org.eclipse.core.expressions;bund
Required Execution Environment	J2SE-1.4,CDC-1.0/Foundation-1.0,J2SE-1.3
State	ACTIVE
Version	3.5.0.v20100524

Figure 4.3: Screenshot of the MacroComponent Monitoring and Managing GUI Tool

The GUI design is based on a Model-View-Controller (MVC) architecture [50] from GEF that distinguishes between model and view. Model is to represent the real entities such as bundles. All information and data about an entity is persisted and only persisted in the model. View is to display the model using figures from Draw2D plug-in. The model and the view do not hold any references to each other. The controller is the bridge between that drives the view depending on the model and modifies the model depending on actions carried on the view.

The architecture of this GUI tool is shown in Figure 4.4. All components to display in this tool are modeled as nodes that hold their properties (e.g., name, status, and layout). There are five kinds of nodes represented in GUI: *Monitor*, *Xen*, *Domain*, *Bundle*, and *Connection*. A Xen could contain multiple Domains as children; a Domain in turn could have a list of Bundles; a Connection is to connect two Domains; the Monitor is the top that holds all the nodes.

The tool has the basic functions such as undo, redo, delete, zoom, keyboard shortcuts. Undo and redo are only in charge of graphical operations like changing layout, adding or deleting a node, but not used to handle operations on components, for example, installing a bundle or starting a domain cannot be undone. In addition, an outline is provided to add a view to show the graph as a tree, and a miniature of the graph is added in the outline view, which is very useful when using zoom functions. Four context menus bound to right mouse click are provided for the operations of Xen, Domain, Bundle and Connection respectively. Also a property window is implemented to display and edit properties of the component nodes. A color property is given to Domains that allows user to customize background color of a domain, which is useful when using zoom together with miniature view where the characters are too small to recognize. New graphical elements, Domain and Connection, can be added by using tools on palette by using drag and drop, and marquee tool is used to select multiple nodes to facilitate mass actions.

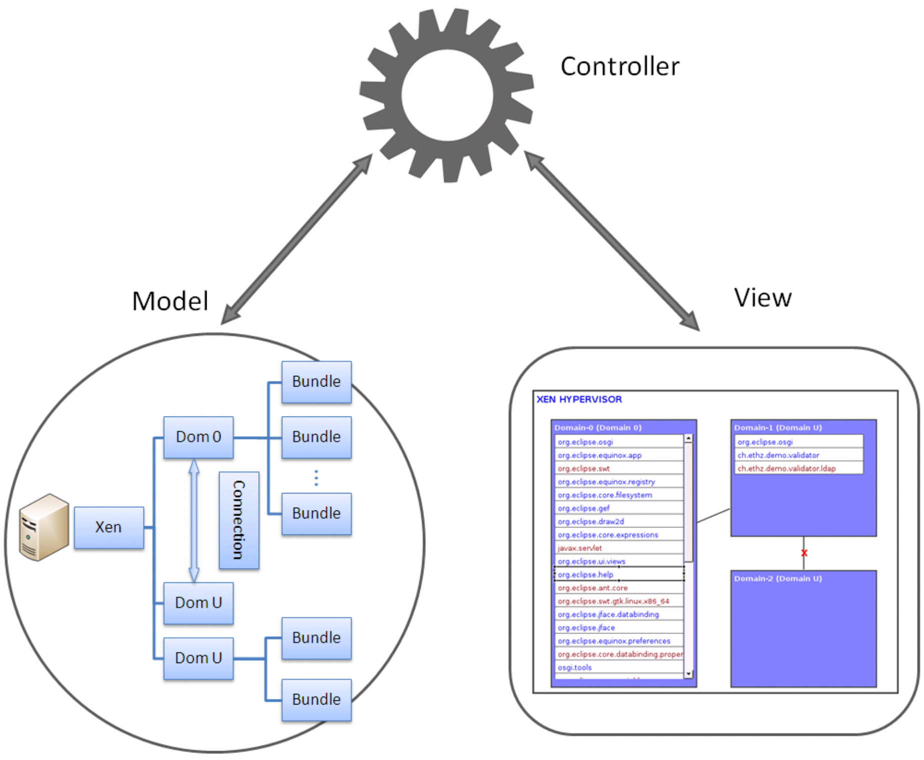
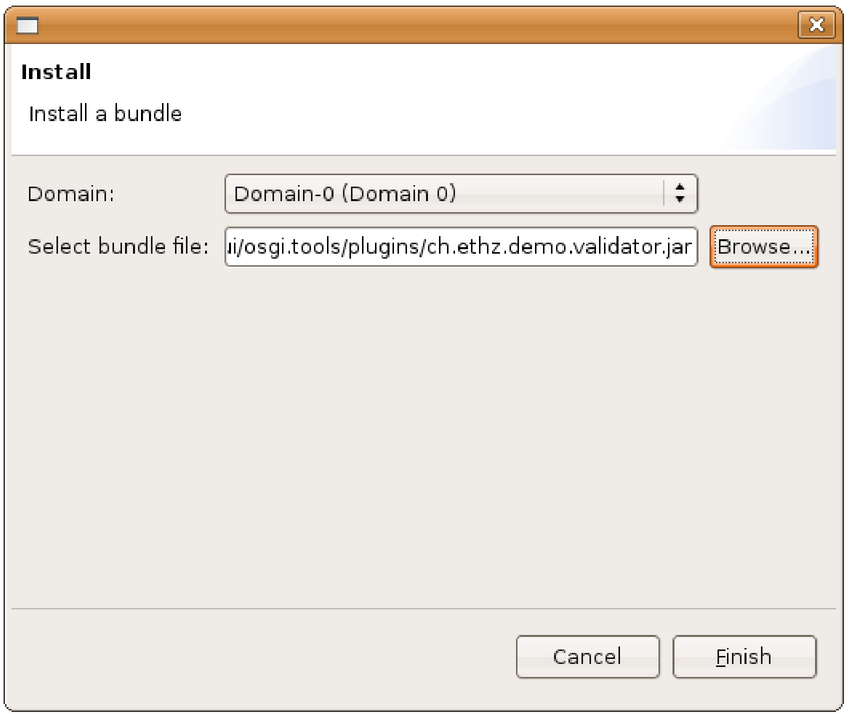


Figure 4.4: The design architecture of the GUI tool

The tool can be used to visualize the structure a MacroComponent system, to monitor the domains running on Xen hypervisor, the OSGi bundles running in a domain, and the connections between domains (the connections are implemented only at a graphic level, and the functionalities behind need the support of a connection channel between domains, which has not been implemented yet. More details are given in section 4.1). The bundles inside Eclipse running in Domain 0, i.e., Linux, are obtained from bundle context, and the properties about OSGi bundles are presented by the tool, including all the information from manifest such as the bundle version, import packages, required bundles. The bundles listed in a domain are sorted by their bundle ID and colored by their state, which gives a clear overview of bundles. A new bundle can be installed from a URL to a domain as shown in Figure 4.3. After installed successfully, a child added event is fired to refresh the visuals of bundles in the domain and add the bundle to the end of list colored by its state. Also, the tool allows starting, stopping and uninstalling each installed bundles. In case of failed, exception information is given with failure reasons.


Install

Install a bundle

Domain: Domain-0 (Domain 0)

Select bundle file: ıi/osgi.tools/plugins/ch.ethz.demo.validator.jar

Figure 4.5: Screenshot of installing a bundle

To handle the OSGi applications running on Domain U, a standalone OSGi server should be run on GuestVM. To achieve this, *org.eclipse.osgi.jar* and application bundles (e.g., helloworld.jar) need to be copied to the virtual disk of GuestVM, which is mounted as */guestvm/java*. In the directory of GuestVM-Native, the OSGi server is started via the command in Listing 4.1. And then, the operations are the same as the usual case. For instance, install a bundle from a certain URL as shown below.

Listing 4.1: Commands for running a OSGi server on GuestVM

```
# bin/run_jre -jar /guestvm/java/osgi.jar -console
osgi > install file:/guestvm/java/helloworld.jar
```

To start a OSGi server on GuestVM through the management tool, *Runtime.exec()* is used to execute the command and return a process. The interaction between the tool and GuestVM domains can be carried out by writing the commands as input to the output stream of the process, and getting the response information from the input stream of the process. This is the way to operate on bundles and to get information of the bundles from GuestVM. However, GuestVM cannot exit as normal when stopping the OSGi server because of a *GuestVM: symbol Java_java_io_FileDescriptor_sync not found, exiting* exception. This crash exit results in the file system of GuestVM mounted as read-only, and cannot run an OSGi server again. This problem is one reason that prevents to run OSGi applications on GuestVM in practical use.

Chapter 5

Conclusions

5.1 Future Work

This work gave a design for inter-domain communications between GuestVM via shared memory. The next step should be implemented and integrated it to R-OSGi connection channels. After that, more evaluations can be carried out for the different implementations of inter-domain communications and compare the results. The next step could be running large applications on such Macro-Components that use GuestVM as Domain U. One convincing application is to run TCP-W benchmark [51] on Tomcat [52] as application server and H2 [53] as database for evaluation. Tomcat and H2 database are written in Java, and is possible implemented as OSGi modules. But before that, numbers of GuestVM bugs need to be fixed.

5.2 Conclusions

High performance is a prerequisite of MacroComponents; otherwise complete operating systems can be used instead. Theoretically, as the key part of Macro-Components, GuestVM has the potential for good performance because of its all-Java and minimal software stack, but in practice it performs quite poor from [2]'s work. This work improved the performance of GuestVM to make it a practical solution for MacroComponents.

GuestVM builds on Maxine, uses Maxine's implementation of Java memory model, and hardly adds overhead to Maxine. But since Maxine is approximately 5 times slower than HotSpot JVM considering the optimization of Java working memory, GuestVM cannot obtain faster memory accesses until the performance of Maxine is improved. However, the GuestVM's I/O performance has been improved that on average is 5 times as fast as Maxine, and only 50% slower than HotSpot JVM. In many cases, GuestVM can even outperform HotSpot JVM. This improvement is via optimizing code, providing enough buffer cache and supporting prefetching.

It is worthy of note that although GuestVM is able to meet the performance requirements of MacroComponents in a way, as an experimental project, it still has numbers of bugs need to be fixed to run big applications in practice.

For running multiple distributed OSGi-based MacroComponents, a design that uses shared memory for communications between GuestVM domains and integrates this communication approach to R-OSGi connection channels is given. It aims to provide an efficient approach for inter-domain communications compared to traditional network. Finally, this work presents a GUI tool that provides a friendly user interface and can be used to visualize the structure of a MacroComponent system, to monitoring the properties of components, to install new OSGi bundles, and start, stop, uninstall the existing bundles.

Acknowledgements

I would like to thank Jan Simon Rellermeyer and Adrian Schüpbach for their guidance and advice and my family and friends for their support and encourage during this thesis. Also I would like to thank Prof. Gustavo Alonso for his support during the master program at ETH Zurich.

Bibliography

[1] P. Parrend and S. Frenot. Security benchmarks of OSGi platforms: toward Hardened OSGi. *Softw. Pract. Exper.*, 39(5):471–499, 2009.

[2] R. Schwammberger. Performance Isolation for Component Systems. Master's thesis, ETH Zurich, 2005.

[3] *Wikipedia entry for Java Virtual Machine.* http://en.wikipedia.org/wiki-/Java_Virtual_Machine.

[4] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java Virtual Machine for Component Isolation in OSGi. In *International Conference on Dependable Systems and Networks (DSN 2009)*, Estoril, Portugal, June 2009. IEEE Computer Society.

[5] Chris Matthews and Yvonne Coady. Virtualized Recomposition: Cloudy or Clear? In *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 38–43, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Glenn Ammons, Dilma Da Silva, Orran Krieger, David Grove, Byran Rosenburg, Robert W. Wisniewski, Maria Butrico, Kiyokuni Kawachiya, and Eric Van Hensbergen. Libra: A library operating system for a jvm in a virtualized execution environment. In *In VEE (Virtual Execution Environments*, pages 13–15, 2007.

[7] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[8] *Solaris Containers (Zones).* http://www.oracle.com/technetwork/systems/containers/index-jsp-140778.html.

[9] Oracle Corporation, http://labs.oracle.com/projects/guestvm/. *GuestVM Project page.*

[10] M. Jordan. *Project Guest VM - A JavaTM platform implemented in Java and hosted on the Xen hypervisor.* Sun Labs, January 2009.

[11] *Wikipedia entry for Hypervisor.* http://en.wikipedia.org/wiki/Hypervisor.

[12] *Xen Official Website.* http://www.xen.org/.

[13] David Chisnall. *The Definitive Guide to the Xen Hypervisor.* Prentice Hall, 2008.

[14] *SPEC CPU2000.* http://www.spec.org/cpu2000/.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[16] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the Art of Repeated Research. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 47–47, Berkeley, CA, USA, 2004. USENIX Association.

[17] University of Cambridge Computer Laboratory. *The Xen^{TM} Virtual Machine Monitor.* http://www.cl.cam.ac.uk/research/srg/netos/xen/performance.html, 2006.

[18] P. Dhawan T. Abels and B. Chandrasekaran. *An Overview of Xen Virtualization.* Dell Inc., Auguest 2005.

[19] B. Mathiske. *The Maxine Virtual Machine.* Sun Labs, 2008. presentation at the JavaOne conference.

[20] Oracle Labs, http://labs.oracle.com/projects/maxine/. *Maxine Project page.*

[21] A. Ananiev and A. Redko. *Using Headless Mode in the Java SE Platform.* Sun Developer Network, http://java.sun.com/developer/technicalArticles/J2SE/Desktop/headless/, June 2006.

[22] *JNode Official Website.* http://www.jnode.org/.

[23] *YANFS Project page.* https://yanfs.dev.java.net/.

[24] *Osgi Service Platform Core Specification*, April 2007.

[25] OSGi Alliance. Listeners Considered Harmful: The "Whiteboard" Pattern, 2004.

[26] *R-OSGi.* http://r-osgi.sourceforge.net/.

[27] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *In Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, 2007.

[28] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol, 1997.

[29] B. Randell. *Operating Systems: The Problems Of Performance and Reliability.* University of Newcastle upon Tyne, Computing Laboratory, 1971.

[30] *SPEC JVM98 Benchmarks.* http://www.spec.org/jvm98/.

[31] *Sun's Java SE HotSpot page.* http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.

[32] *Run SPEC JVM98 Benchmarks from the command line without GUI controls.* http://www.spec.org/jvm98/jvm98/doc/commandLine.html.

[33] Kyle R. Bowers and David Kaeli. Characterizing the SPEC JVM98 Benchmarks On The Java Virtual Machine, 1998.

[34] *OProfile.* http://oprofile.sourceforge.net.

[35] R. E. Bryant and D. R. O'Hallaron. *Computer Systems A Programmer's Perspective*, chapter 10.9 Dynamic Memory Allocation. Prentice Hall, November 2001.

[36] B. Eckel. *Thinking in Java*, chapter 2.2 You must create all the objects. Prentice Hall, fourth edition edition.

[37] Guy Steele James Gosling, Bill Joy and Gilad Bracha. *The Java Language Specification*, chapter 17 Threads and Locks. Sun Microsystems, Inc., third edition edition.

[38] Doug Lea. *Concurrent Programming in Java^{TM} Design principles and patterns*, chapter 2 Synchronization and the Java Memory Model. Addison-Wesley, November 1999.

[39] E. Ciliendo. *Tuning Red Hat Enterprise Linux on IBM eServer xSeries Servers.* IBM International Technical Support Organization, July 2005.

[40] Intel Corporation. *Using the RDTSC Instruction for Performance Monitoring*, 1997.

[41] *Performance tuning.* http://en.wikipedia.org/wiki/Performance_tuning.

[42] *Java^{TM} 2 Platform, Standard Edition, v 1.4.2 API Specification, LinkedHashMap.* http://java.sun.com/j2se/1.4.2/docs/api/java/util/LinkedHashMap.html.

[43] Alex Weeks. *The Linux System Administrator's Guide*, chapter 6.6 The buffer cache. Version 0.9 edition.

[44] *A Non-Technical Look inside the Ext2 File System.* http://linuxgazette.net/issue21/ext2.html.

[45] F. Wu. *Prefetching Algorithms in Linux Kernel.* PhD thesis, University of Science and Technology of China, August 2008.

[46] Xiaolan Zhang, Suzanne Mcintosh, Pankaj Rohatgi, and John Linwood Griffin. XenSocket: A high-throughput interdomain transport for VMs. Technical report, 2007.

[47] *XenStore - Xen Wiki.* http://wiki.xensource.com/xenwiki/XenStore.

[48] *Rich Client Platform.* http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[49] *Eclipse Graphical Editing Framework.* http://www.eclipse.org/gef/.

[50] *GEF Description - Model-View-Controller Architecture.* http://wiki.eclipse.org/GEF_Description#Model_-_view_-_controller_architecture.

[51] *TCP Benchmark^{TM} W (TCP-W).* http://www.tpc.org/tpcw/.

[52] *Apache Tomcat.* http://tomcat.apache.org/.

[53] *H2 Database Engine.* http://www.h2database.com/html/main.html.