The JikesXen Java Server Platform

Georgios Gousios

Athens University of Economics and Business
gousiosg@aueb.gr

Abstract

The purpose of the JVM is to abstract the Java language from the hardware and software platforms it runs on. For this reason, the JVM uses services offered by the host operating system in order to implement identical services for the Java language. The obvious duplication of effort in service provision and resource management between the JVM and the operating system has a measurable cost on the performance of Java programs. In my PhD research, I try to find ways of minimizing the cost of sharing resources between the OS and the JVM, by identifying and removing unnecessary software layers.

Categories and Subject Descriptors C.0 [*General*]: Hardware/software interfaces; C.4 [*Performance of Systems*]: Performance Attributes; D.4.7 [*Operating Systems*]: Organization and Design

General Terms Performance, Languages

Keywords JVM, Performance, Operating System, Virtual Machine

1. Background and Motivation

The *raison d' être* of contemporary JVMs is to virtualize the Java language execution environment to allow Java programs to run unmodified on various software and hardware platforms. For this reason, the JVM uses services provided by the host OS and makes them available to the executing program through the Java core libraries. Table 1 provides an overview of those services.

The JVM is a software representation of a hardware architecture that can execute a specific format of input programs. Being such, it offers virtual hardware devices such as a stack-based processor and access to temporary storage (memory) through bytecode instructions. The JVM is not a general purpose machine, though: its machine code includes support for high level constructs such as threads and classes, while memory is managed automatically. On the other hand, I/O is handled by the OS and access to I/O services is provided to Java programs through library functions. The observation that the JVM is both a provider and a consumer of services leads to the question of whether the JVM can assume the role of the resource manager.

Generic architectures sacrifice absolute speed in favor of versatility, expandability and modularity. A question that emerges is whether the services provided by the OS are strictly necessary for the JVM to execute Java programs and, if not, what will be the increase in the performance of the JVM if it is modified so as to manage the computing resources it provides to programs internally. Part of my work is to assess the necessity of certain services offered by the OS to the operation of the JVM and to measure their effect on the JVM performance, to get an estimate of the possible speedup that could be expected if the JVM replaced in the OS in the role of the resource provider/broker, in the context of purpose-specific systems.

My initial findings show that Java pays a large price to the OS for services it does not necessarily require. All standard OSs are designed for a specific purpose: to protect programs from accessing each other's memory and to protect I/O resources from being accessed concurrently. As others have shown [2], if the execution environment is based on a type-safe language and therefore protects itself against invalid memory access, then memory protection, as it is currently implemented by OSs, is mostly unnecessary. My currently unpublished findings show that the mechanisms employed by JVMs to overcome the restrictions placed to them by the OS APIs can deteriorate their performance, mainly when executing memory and I/O intensive applications.

2. Our Research

The specific question my research tries to answer is the following: *Can the performance of Java server platforms be improved by allowing them to manage directly the computing resources they require?*

For the purposes of my research, I make the following assumptions — based on worked published by other researchers:

Resources	JVM	System Library	OS Kernel	JikesXen
CPU	Java to Native Thread Mapping	Native to Kernel Thread Mapping	Thread Resource Allocation, Thread Scheduling	Multiplexing Java threads to CPUs, Thread initialization
Memory	Object Allocation and Garbage Collection	Memory Allocation and Deallocation from the process address space	Memory protection, Page Table Manipulation, Memory Allocation	Object Allocation and Garbage Collection
I/O	Java to System Library I/O Mapping, Protect Java from misbehaving I/O	Provide I/O abstractions	System call handling, Unified access mechanisms to classes of devices	Driver infrastructure, I/O multiplexing

Table 1. Resource management tasks in various levels of the Java execution stack and in JikesXenJVM

- A JVM is able to manage efficiently computational resources by scheduling Java threads to physical processors internally. Multiprocessing can be implemented within the JVM very efficiently [1].

- Virtual memory is deteriorating the performance of garbage collection [4] and thus it should be omitted in favor of a single, non-segmented address space.

- A specially modified JVM can be run directly on hardware with minimal supporting native code [3], and therefore the unsafe part of the system can be minimized down to a very thin hardware abstraction layer.

To investigate my research question, I am in the process of designing and building the JikesXen virtual machine, which is a port of JikesRVM to the Xen VMM platform. JikesXen is based on a very thin layer of native code, the nanokernel, whose main purpose is to receive VMM interrupts and forward them to the Java space and also to initialize the VMM on boot. I use parts of the Xen-provided MiniOS demonstration operating system as the JikesXen nanokernel. The system itself is essentially a blob that lumps together the JikesRVM core image and stripped down version of the classpath which are already pre-compiled to native code during the JikesRVM build phase. JikesXen does not require a device driver infrastructure; instead, the Java core libraries will use the devices exported by the VMM directly, through adapter classes for each device. The adapter classes are also responsible to prevent concurrent access to shared resources, such as I/O-ports using Java based mutual exclusion primitives (e.g. synchronized blocks). It also implements other functionality relevant to the device class such as caching, network protocols, file systems and character streams. Table 1 presents an overview of the functionality of JikesXen as a resource manager.

My work falls into the bare metal JVM research stream. But how does it differ from already existing approaches? The distinctive characteristic of my system is that it does not run directly on hardware. The use of the Xen VMM for taking care of the hardware intricacies, allows me to focus on more important issues than communicating with the hardware, such as providing a copy-less data path from the hardware to the application. Additionally, since all access to hardware is performed through the classpath, I plan to

use the JVM locking mechanisms to serialize access to it. This will render the requirement for a resource manager, or a kernel, obsolete. Finally, my system's focus being a single multithreaded application, I do not consider a full-fledged process isolation mechanism. Instead, we base our object sharing models on proven lightweight object sharing mechanisms (isolates [1]).

3. Conclusions

My work is in its early stages of development. I am currently constructing a build system that will compile the classpath and JikesRVM in a single binary image. I have already succeeded in loading the JikesRVM boot image in the Xen hypervisor, but at the moment the system does nothing useful as important subsystems remain to be implemented.

Acknowledgments

References

[1] Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer. A multi-user virtual machine. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*, pages 85–98, San Antonio, Texas, USA, June 2003. USENIX.

[2] Galen Hunt et al. An overview of the Singularity project. Microsoft Research Technical Report MSR-TR-2005-135 MSR-TR-2005-135, Microsoft Research, 2005.

[3] Georgios Gousios. Jikesnode: A Java operating system. Master's thesis, University of Manchester, September 2004.

[4] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, New York, NY, USA, 2005. ACM Press.