Rethinking the Java software stack: Optimisation opportunities in the face of hardware resource virtualisation

Georgios Gousios and Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
{gousiosg,dds}@aueb.gr

ABSTRACT

The purpose of the JVM is to abstract the Java language from the hardware and software platforms it runs on. Currently, there is an obvious duplication of effort in service provision and resource management between the JVM and the operating system that has a measurable cost on the performance of Java programs. The emergence of efficient hardware resource virtualisation mechanisms presents implementers with new opportunities for optimising the Java software execution stack.

In this paper, we examine the sources of the runtime overhead imposed on the Java programming language by the native execution environment. We use both synthetic and real world applications as benchmarks along with modern instrumentation tools to measure this overhead. We base our measurements on the assumption that the JVM can be directly hosted on virtualised hardware. Based on our findings, we also propose a cost estimation heuristic, which allows us to estimate the minimal gain to be expected when applications will be moved to hypervisor-hosted virtual machines.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interfaces; C.4 [**Performance of Systems**]: Performance attributes; D.4.7 [**Operating Systems**]: Organisation and Design; D.4.1 [**Operating Systems**]: Process Management

General Terms

Virtual Machine, Operating System, Performance

Keywords

JVM, `dtrace`, Performance, Operating System, Benchmarking, Virtual Machine

1. INTRODUCTION

In recent years, there has been a trend toward developing and running Internet and network servers using safe languages and processes-level Virtual Machine (VM)-based runtime environments. This trend is justified; VMs offer a more secure execution environment than native platforms, while they allow programs to be portable. Those facilities come at a cost: process-level VMs introduce several layers of indirection in the chain of management of computing resources. VM runtime environments offer services complimentary to those of operating systems, such as processing time sharing and preemptive multithreading, memory management, and I/O request handling. Despite the advances in automatic memory management and Just-In-Time (JIT) compilation, which brought the number crunching abilities of process VMs to nearly-native levels, there is a very small number, if any, of VM-based server software applications that can match the performance of widely deployed, natively compiled network servers, such as Apache or Samba.

On the other hand, hardware virtualisation has become mainstream. Hypervisors such as Xen [3] and VMWare ESX [32] along with specially designed instruction sets on modern processors enable hardware consolidation and efficient resource utilisation. Hypervisors have some interesting properties: they enable hosted programs to directly access processor and memory resources while presenting a portable abstraction to I/O devices. Older [34] and recent [9, 1] research has shown that it is entirely possible to run modified applications directly on top of virtualised hardware with increased performance.

Our research work involves modifying a JVM to run directly on top of virtualised hardware. In this paper, we evaluate the impact of the duplication in effort of resource management between the JVM and the operating system on the performance of contemporary JVMs. We also identify optimisation possibilities and quantify the gains in performance that can be expected in our implementation.

We conducted our experiments on version 1.6 of Sun's JVM and the latest available built (`nv66` as of this writing) of the OpenSolaris operating system. We also conducted a limited number of performance tests on an experimental port of the JikesRVM JVM to the OpenSolaris platform. The choice of this particular stack was mainly driven by three reasons:

- The availability of advanced instrumentation tools which can leverage data sources in both the OS and the JVM at the same time
- The availability of the source code for the entire stack
- The fact that Solaris and the Sun's JDK are used in production environments to host complex applications

Our experiments were deliberately run on standard PC hardware, instead of high performance servers, to demonstrate better the overheads involved. Unless stated otherwise in a particular experiment description, we utilised a Core 2 Duo dual core machine featuring 1GB of RAM and 2MB of L2 cache, shared among the processors. We used `dtrace(1)` [8] as our main performance evaluation tool and a combination of readily available and custom-developed benchmarks.

The major contributions of this paper are (1) a discussion on the necessity of services provided by the OS for the operation of the JVM and (2) a quantitative study of the effect of the OS protection and service handling mechanisms to the performance of Java. The paper also employs performance evaluation tools not previously considered for JVM performance measurements.

2. MOTIVATION

Virtualisation of computing resources can take place either at the hardware or at the software level [30]. Virtualisation allows resource sharing architectures to be stacked [13, 23]. VM stacking is common in current architectures as it enables lower level VMs to expose simple interfaces to the hardware or software they virtualise, which in turn allows higher level VMs to remain unaware of the hardware/software complications present in the lower levels. The *raison d' être* of the JVMs is to abstract the Java language from the hardware and software combination it runs on. In the Smith and Nair VM taxonomy [30], the JVM is a high level language VM, which also implies that it is a process VM.

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 machine code 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, there is no provision in the JVM specification about I/O. Most implementations use the OS to access I/O devices while these services are provided to Java programs through the Java library, using some form of binding to the OS native library functionality. The observation that the JVM is both a provider and a consumer of equivalent classes of services leads to the question of whether the JVM can assume the role of the resource manager.

Generic architectures usually sacrifice absolute speed in favour of versatility, expandability and modularity. In our opinion, this need not be the case for purpose-specific systems, such as application servers and embedded devices. The basic question that our research is trying to answer 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 our work is to assess the necessity of certain services offered by the OS to the operation of the JVM operation and to measure their effect on the JVM performance, to obtain an estimate of the possible speedup that could be expected if the JVM assumed the role of the resource provider/broker, in the context of purpose specific systems.

We are currently in the process of implementing the JikesXen [16] hypervisor-hosted JVM. As its name implies, we use the Xen hypervisor and JikesRVM as our runtime system; a thin native code layer is placed between the two and is responsible for initialisation and interrupt processing. We try to push our system's performance enhancing possibilities by implementing hardware drivers for the Xen-exported devices in Java. Drivers and application software run in the same heap and are able to share objects using the JSR-121 [26] mechanisms. The Java system library (classpath) is modified to interact directly, via method calls, with a thin resource management layer, which also includes the device drivers. Our system uses the M:N threading model and thread switching inside the JVM, as it is currently implemented in JikesRVM. The JikesRVM virtual processor threads are mapped directly on the available processors, although currently our system does not support more than one physical processors. The system memory manager uses a non-segmented memory space as its garbage collected heap. All memory is allocated at boot time and currently there is no support for memory swapping. Finally, our system does not support loading or executing native code, through interfaces such as the JNI.

2.1 What is overhead?

The software stack used to execute Java programs currently consists of three layers of software: the JVM, the JVM implementation language native library (usually `libc`) and the OS kernel. Table 1 summarises the most important resource management tasks for each resource type that are performed in each one of the three software layers. In the context of JikesXen, many of the

Resources	JVM	System Library	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, Multiplexing of requests, Provision of 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 JikesXen

resource management tasks are redundant, while others are performed in the JVM, using our resource management layer. With the term *overhead*, we refer to the execution time spent in redundant resource management tasks, for services that were requested by the JVM in order to support the execution of the Java programming language. Services offered by the JVM and which are hard requirements for the Java language to work efficiently, such as the JIT compiler, the garbage collector or the bytecode verifier, are not considered to impose overhead.

Most of the resource management tasks that are described in Table 1 can be regarded as redundant. For example, since our system does not use page swapping, no memory management tasks should be executed outside the JVM. On the other hand, the time spent in executing device driver code cannot be considered an overhead as device drivers are also required in our system. In order to calculate the native execution environment overhead, we must subtract from the set of all the resource management tasks those which are required in both the current implementations of the Java stack and also in the standalone JVM. Therefore, with the term *guaranteed overhead* we refer to the time that is currently spent in resource management tasks not required for JikesXen. The calculation of the guaranteed overhead value for the current Java execution stack provides us with an estimate of the performance enhancements we should expect from an OS-less execution stack and also sets the performance bar for the development of JikesXen.

In the following section, we analyse the factors that contribute to the generation of guaranteed overhead for each resource category.

3. ACCESSING OPERATING SYSTEM SERVICES

Languages which are considered safe, such as Java, cannot trust user code to access external services, such as those provided by the operating system. The JVM is required to check all data exchanges with external software, especially buffer lengths. On the other hand, interoperability and compatibility requirements with existing code necessitate a generic mechanism for accessing data in the Java program memory space while the JVM is executing the program. A mechanism for enabling this type of access but disallowing inappropriate memory manipulations is thus required.

Task	JNI Calls	% Copying	Bytes
Tomcat serving a 10k web page	590	18	22k
Java2D demo graphics	2.5M	12	360M
1M 10-byte random read-writes	6.5M	18	10M

Table 2: Number of JNI calls required to perform routine tasks

The mechanism used to handle interaction with native code is called Java Native Interface (JNI) [24]. The JNI, among other things, defines the naming conventions and the data types required for Java code to call native functions (*downcalls*) and the necessary interface for native code to manipulate Java objects in a live Java heap (*upcalls*). The JNI specification does not define a universally applicable mechanism for invoking native functions (i.e. stack layout, argument passing), or for returning values from the native call to the Java code. It allows each JVM to handle the return semantics differently. In a downcall, the function arguments are passed to the native code either by copying, if they are primitives, or by reference, if they are objects. Referenced objects need to be copied prior to being manipulated, due to the indirect referencing mechanism that is used. The return values must be copied to the Java heap since the native code calls execute in the calling thread local context.

The JNI is used extensively in various JVMs as it is the only universally accepted mechanism for interfacing Java code to I/O services offered by the OS. As by specification the JNI is a copying mechanism, all accesses to

I/O facilities need to pass through a copying layer of software, which in turn restricts the overall I/O capacity of the Java language. Table 2 illustrates the extend of use of the JNI copying mechanisms in the Java execution environment. To gather the data, we ran a simple `dtrace(1)` script that counts the total number of calls to the JNI layer and the bytes that were copied during those calls, on a collection of programs that perform a variety of tasks. As copying functions, we only consider JNI functions that copy entire memory regions, such as arrays and strings. On both the Sun JVM and JikesRVM the results are almost the same, with small variations that can be attributed to different bootstrapping sequences. From the table, we can see that a large number of JNI calls copy data, an operation which systems designers try very hard to avoid. All Java programs, especially those relying on I/O to function, are handicapped performance-wise by a poorly designed data exchange mechanism.

In addition to the JNI, the two JVMs we examine feature private interfaces to the OS. Those interfaces abstract the operating system functionality at the cost of a function call to facilitate the porting of each JVM across operating systems. All classpath method calls in addition to each VM's internal API calls that require native function implementations to access OS services are routed through the indirection layer. The particular implementation details differ in each case, but the fact remains that another layer of indirection is placed between the JVM and the OS, causing unnecessary overhead. The cost in the case of Sun's JVM is equal to that of a virtual method dispatch, while in JikesRVM the indirection layer functions are prebound at compile time. However, in our experiments with heavy I/O workloads we did not find the indirection layer to contribute more than 1% to the real execution time in both JVMs.

Since all interactions between the executing Java code and the native execution environment have to pass through either the JNI or the indirection layer, or both, those two mechanisms are responsible for the lion's share of the guaranteed overhead imposed by the OS access mechanisms to the Java language.

3.1 Input/Output

3.1.1 Blocking I/O

Blocking I/O is the most common form of I/O in Java; it is based on proved primitives such as streams and the `read/write` system calls. Java supports blocking I/O through the `java.io` class hierarchy. It also supports random access I/O through the `RandomAccessFile` interface. All Java I/O functions are mapped to native calls that access the OS-provided I/O services through the system library. The exact same mechanism is also used for accessing both network and graphics devices and therefore similar restrictions and performance bottlenecks apply.

In order to demonstrate the overhead imposed by the JNI copying semantics, we used `dtrace(1)` to instrument a program that, in a tight loop, performs writes of 10 byte arrays to the system's `null` device, implemented in both C and Java. The `dtrace(1)` sensors count the number of invocations of the `memcpy(3)` library function and the amount of data passed to the `write(2)` system call. The results of the experiment are presented in Table 3.

The C implementation is more than 100% faster than the Java implementation. This fact cannot be attributed to differences to the C compiler or the Java VM that were used, since the program's critical path only includes a loop construct and a system call, none of which can be optimised. In fact, fully optimised versions of the native program did not exhibit any measurable performance increase. Also, only 750 system calls were performed during the JVM initialisation. The time that might be required by the JIT to compile the used methods to native code is also too small to affect performance considerably, as the method that performs the system call cannot be compiled to machine code. Therefore, the reason for these significant performance differences can be mainly attributed to the differences of the operating system services calling semantics between the C and the Java languages, the most important being the JNI layer.

In the case of blocking I/O the guaranteed overhead is generated by: (1) the copying in the JNI layer and (2) the OS kernel `copyin/out` functions. The JNI layer overhead is assessed by measuring the time spent in the `memcpy` function. Due to technical limitations in the current implementation of `dtrace(1)`, the cost of the kernel copying functions cannot be measured, but it can be approximated with good precision. [1] The approximation method we used was to set it equal to the cost of the library `memcpy` function. By calculating the bytes that each I/O system call is pushing to or pulling from the kernel, we can approximate the total time spent in the kernel copying functions by diving the number of bytes with the constant cost of copying, eg 100 bytes. For our benchmark system, the constant time to copy 100 bytes was found to be equal to 10μsec. The guaranteed overhead in the case of I/O is 2.2sec, which os time only devoting to copying data.

3.1.2 Memory Mapped I/O

Memory mapping is a special case of I/O in that it employs the OS's virtual memory system to create file mappings to a process's address space. No system calls

[1] The `fbt dtrace(1)` provider is not able to instrument functions which do not setup a stack frame and the OpenSolaris kernel implements `copyin/copyout` as inlinable, hand-optimised assembly routines.

Function	C			Java			JikesRVM		
	Calls	**Bytes**	**Time**	**Calls**	**Bytes**	**Time**	**Calls**	**Bytes**	**Time**
`memcpy(3)`	14	68	10μsec	10^6+	10^7+	1.1sec			
`write(2)`	10^6	10^7	—	10^6	10^7	—			
`copyin(9F)`	10^6	10^7	1sec	10^6	10^7	1sec			
exec. time		21.62sec			45.16sec			sec	

Table 3: The cost of copying in blocking I/O

are required for accessing or updating data in memory mapped files, although any possible performance gains can be quickly amortised if the whole file is accessed. Java supports mapping of files in the JVM process space, through the NIO I/O framework [28]. The `MappedByteBuffer` class allows a file to be accessed as a byte array. The NIO API allows both sequential and bulk reads and writes. The two systems we examine handle memory mapped I/O differently. The Sun JVM

is an interesting difference on how operations involving primitive arrays and operations involving objects or array ranges are handled:

- Bytes in container objects and array ranges are copied in a loop directly to the mapped file, one byte per loop iteration.
- On the other hand, byte arrays are copied from the executing thread context to the mapped file area using the `memcpy(1)` library function

The memory mapping functionality in Java does not pose any significant overhead to the Java language execution, apart from the overhead introduced by the page manipulation functions at the kernel level. The cost of memory mapping at the kernel level does not contribute to the guaranteed overhead, as an equivalent mechanism will be required to be present in the standalone JVM implementation.

3.2 CPU resource sharing

3.2.1 *Java and* OS *threads*

The Java language offers built-in support for threads and also for thread related functionality such as synchronization and semaphores. For efficiency, most JVMs do not implement Java threads internally, but instead they map them to OS-managed threads. The Sun JVM uses an 1:1 threading model: all instances of the Java `Thread` class are mapped to a native OS thread, through the system's native thread library. Other JVMs, such as the JikesRVM, use more sophisticated M:N threading models, which enable them to manage thread priorities internally and put less pressure on the OS scheduler in highly multithreaded environments.

The performance of the OS threading architecture is a critical component for the performance of the JVM.

System	100 Threads Init Time
Opensolaris, Pentium III 733 MHz	72.4
MacOSX, Core 2 Duo, 2.33GHz	21.0
Solaris 10, Dual Sparc, 750MHz	17.3
Linux 2.6.11, Dual Opteron, 2.2GHz	7.2

Table 4: NoOp threads creation time across different platforms. Times are in milliseconds

When a JVM needs to create service threads for a large workload, it must initialize both the Java related data structures and the native thread data structures. Our experiments show that allocating and starting 100 Java threads with an empty `run()` method, (`NoOp` threads) can take from 7.2ms to 21ms on various combinations of modern 2GHz multiprocessing hardware and operating systems (see Table 4). Given that the JVM is the same program version across almost all the platforms we tested, this small experiment proves that threading is indeed causing an overhead and that overhead is mostly OS-dependent.

In the case of the threading, both the kernel thread initialization and cleanup and the library equivalents contribute to the guaranteed overhead. Accurate measurements of the guaranteed overhead introduced by the native threading services can be performed by inserting `dtrace(1)`-based counters at each layer's thread initialization code. The entry point to each layer is not too difficult to be discovered; on Solaris, calls to the `thr_create` function denote entry to the system's C/threading library while system calls initiated while in the threading library are performed to create the appropriate kernel threads. The entry points to all the layers of the threading stack were used to construct a `dtrace(1)` script that counts the processing time for each layer. The script was run against the `NoOp` threads program described above. Except from our dedicated test machine, we also run the experiment on a Pentium III machine.

Table 5 presents a breakdown of the time devoted to each layer of the threading stack. As a reference, we also calculated the time required by Java to initialize

System	Java		Native		Overhead
	Java	jvm	libc	kernel	
OpenSolaris, Pentium III	110.3	64.1	69.9	89.9	91%
Solaris 10, Dual Sparc	127.3	51.9	15.7	52.2	37%

Table 5: Cost of various levels of the threading stack on Java performance. Times are in microseconds

thread related structures, in both the Java and the JVM layers. On our Solaris test machine, the native threading initialization time is imposing a 37% overhead on the Java platform. The situation is much worse on the Pentium III uniprocessor machine, where the native execution environment slows down the Java platform by a factor of 94%.

3.2.2 Locking and Mutual Exclusion

Related to threading is the issue of locking a resource from concurrent access and excluding concurrent execution of a specific code path. The Java language provides inherent support for both locking and mutual exclusion via the `synchronized` keyword and library-JVM co-design. The JVM handles the locking of Java objects internally and therefore no overhead is introduced by requiring the OS to do the job for the JVM. OS-level locking is required by the JVM to implement locking of shared OS-provided resources, such as files or sockets, when performing operations on those resources on behalf of Java code.

3.3 Memory

The JVM uses the system library as the interface to the operating system memory management mechanisms. The JVM maintains an internal memory allocator which co-operates with the garbage collector. The allocator requests memory in chunks of increasing size; most often, the allocated memory is not returned to the operating system until the end of the JVM lifetime. The system library uses the `brk(2)` system call to request memory from the operating system, while it also maintains internal structures, usually segregated lists, to keep track of allocations. This is a serious source of overhead for the Java language: while the JVM features advanced memory management mechanisms, the overall performance of the Java memory subsystem depends heavily on, and is limited unnecessarily by, the native memory allocation mechanisms, which were repeatedly proven to be slow [5, 21].

The guaranteed overhead in the case of memory management is the sum of the time spend in the system library and the kernel page manipulation functions. In order to evaluate the effect of the native resource management tasks we instrumented the system library memory related functions (the `*alloc()` family, `free()`) and also the `brk(2)` system call to report the total time spent in memory operations. We use the memory intensive DaCapo [6] benchmark suite as our workload. The results of our experiment are presented in Table 6. The average guaranteed overhead, which was calculated by comparing the time reported by our instrumented functions to the total execution time, of the system library memory management subsystem on the JVM was 5%, excluding the `jython` benchmark whose the performance was an order of magnitude worse. This overhead is entirely superfluous, as the service offered by the native library memory manager is not required for our standaloneJVM to function. On the other hand, the OS kernel was found not to affect the memory subsystem performance significantly.

Program	Exec time	libc	kernel	Overhead
`antlr`	35382	1179	3.9	3.3%
`bloat`	294101	2587	4.2	0.9%
`chart`	96169	5737	2.4	5.9%
`eclipse`	377705	31037	10.6	8.2%
`fop`	17640	1185	3.1	6.7%
`hsqldb`	59964	930	1.7	1.6%
`jython`	223652	100628	4.6	45.0%
`luindex`	43978	1724	1.8	3.9%
`lusearch`	87812	4156	2.6	4.7%
`pmd`	98809	1492	3.2	1.5%
`xalan`	226861	27375	3.8	12.0%

Table 6: Memory allocation costs for the DaCapo benchmark. Times are in milliseconds

3.4 Implicit sources of overhead

3.4.1 Context switching

The operating system, in order to protect its state from misbehaving processes, runs in privileged mode. When the JVM, which runs in a less privileged mode, requests a service from the kernel, it must issue a software interrupt. At this point, the processor's protection level must be escalated which involves saving the requesting process's runtime state (registers and stack) and loading the operating system's previous runtime state. This mechanism, refereed to as context switching, has been studied extensively and is known to affect a process's execution time by both the processing required, and most importantly, by invalidating the processor's caches. The cost of process-initiated context switching is directly proportionate to the number of I/O operations; in I/O-bound workloads, the kernel and processor architecture play an important role on minimizing the effects of context switching.

3.4.2 Interprocess Communication

Interprocess communication (IPC) is a term used to describe the mechanisms required for two processes running in different process spaces, or even on different machines, to exchange data. In current operating systems, the generic mechanisms offered are mainly shared memory and message passing. In all cases, the operating system is responsible to establish the communication, to grant access to the exchanged information and to transfer control between the sending and the receiving process. This means that IPC is an operating system based class of services and therefore programs written in natively compiled languages can use those mechanisms efficiently. Due to garbage collection and memory consistency requirements, Java programs cannot use shared memory to exchange objects or other types of information; they must rely on cross address space mechanisms such as sockets and named pipes for message passing. This need not be the case in the standalone JVM system: research [27, 19] has shown that is feasible and practical to share objects between Java programs sharing the same heap.

4. COST ESTIMATION

Up to this point, we have shown that running the JVM on top of an OS limits its performance in a way that it is both existing and measurable. The question that emerges is whether we can approximate the guaranteed cost that the native execution environment inflicts on the JVM in a generic, OS-independent way, that will allow us to predict the cost prior to deploying a service. For this reason, we can think the JVM as a consumer of OS-provided services. The OS service provision interfaces are well defined and, additionally, similar across a multitude of OSs. Each interface provides a unique service that is exported to the JVM through the system library. Interfaces can be stacked or otherwise combined; for example the thread interface can hold references of the filesystem interface. The total cost is the sum of the costs incurred by each OS-provided service.

$$C_{os} = C_{thr} + C_{I/O} + C_{mem} \tag{1}$$

The cost C_{thr} refers to the overhead of establishing service threads. On a given platform, the guaranteed cost for creating a thread (C_{nt}) is, mostly, constant; Therefore the total cost for thread creation depends on the number of threads N_{trh} to be created.

$$C_{thr} = n_{thr} * C_{nt} \tag{2}$$

The cost $C_{I/O}$ for I/O can be further broken down to the individual costs for establishing an I/O link, such as opening a file (C_{file}) or accepting a server network connection request (C_{link}), and the cost of reading and writing bytes to the link. Since the cost $C_{I/Ochunk}$ of performing I/O operations of constant chunk size using blocking I/O primitives, such as the `read` and `write` system calls, can be measured relatively easily on most platforms, it is helpful to analyze the I/O operation cost to the number of chunks times the cost per chunk. Furthermore, since the cost of writing to networking sockets is not very different to that of writing to files, at least in fast networks, we can either opt to calculate separate costs for file I/O and networking or to calculate the mean value:

$$\begin{aligned} C_{I/O} &= C_{net} + C_{file} && (3) \\ C_{net} &= n_{links} * ((n_{chunks} * C_{I/Ochunk}) + C_{link}) && (4) \\ C_{file} &= n_{files} * ((n_{chunks} * C_{I/Ochunk}) + C_{file}) && (5) \end{aligned}$$

Finally, the cost C_{mem} for allocating and freeing memory depends heavily on the JVM used, as each JVM features different policies for allocating and for freeing memory. In previous work [17], we have witnessed different heap expansion patterns for two production JVMs, even between different configurations of the GC on the same JVM. In the same study, we observed that the heap size remained constant when the workload stabilized, as the executed application filled its object pools with adequate objects. Those two observations combined mean that it is not straightforward to predict the cost of allocating memory but this cost is only paid once during the application ramp up period, so it can be regarded as a constant value that can be discovered through experimentation.

The initial cost formula can be simplified if we consider the structure and operation of server applications. Those applications have an initial startup cost, which is very small in comparison with the runtime cost when the application is under full load. Also, once the application has reached a steady state, the only OS-related costs that apply are those related to the number of threads (n_{thr}) started and to the number (n_{chunks}) of basic I/O operations.

$$C_{os} = n_{thr} * (C_{nt} + C_{link} + n_{chunks} * C_{I/Ochunk}) + C_{mem} \tag{6}$$

The heuristic presented above is an approximation for the total cost, expressed in terms of runtime slowdown, that makes some necessary, albeit non-intrusive, simplifications. It does not represent a formal cost evaluation model but, as proven by our experiments, it captures the measured runtime cost with good approximation accuracy. The constants C_{nt}, $C_{I/Ochunk}$ and C_{link} need to be measured via experimentation for each platform/hardware combination. For our test platform, the values, measured in microseconds, are:

$$\begin{aligned} C_{nt} &= 173 \\ C_{I/Ochunk} &= 3.3 \\ C_{link} &= 140 \end{aligned}$$

5. EXPERIMENT

Microbenchmarks, such as those presented in the previous sections, serve to isolate a specific performance bottleneck of a single subsystem. In order to understand how the problems that were illustrated using microbenchmarks affect real-world applications, we run a series of mostly I/O and kernel-bound applications, which we stressed using load generators. Specifically, we tested a dynamic web content application and a message room application, both of which put a significant load on the operating system.

The first application we selected is an open source variation of the well-known Java Pet Store application, called JPetStore. The JPetStore application simulates an e-commerce site; it uses dynamic web pages and an embedded database to store information.It also features an object persistency layer in order to minimize the cost of accessing the database. The JPetStore application is run by the Tomcat application server. By default, the JPetStore application returns web pages which are very small in size; we increased the size of the return pages in order to match real world situations. For the purposes of the experiment, Tomcat was configured with a large connection thread pool and a heap space equal to 1GB. We used a custom-developed lightweight HTTP load generator in order to be able to control the workload parameters, namely the number of concurrent threads and the number of pages each thread retrieved before it halted.

The second application we measured was the VolanoMark workload. VolanoMark simulates a Internet Relay Chat environment, where users are able to create chat rooms to which other users are able to connect and exchange messages. The benchmark load generator allows the specification of the number of users, rooms and exchanged messages to be simulated. Each user connection generates two server threads, which listen for and dispatch incoming messages from the chat room object to the user socket and vice versa. The VolanoMark benchmark is mainly taxing the operating system scheduler and the synchronization methods on both the JVM and the OS. When a large number of messages are exchanged by many threads can the networking subsystem can also be exercised.

The workload generator for both benchmarks was run on a different machine than the workload itself. The machines used 100Mbps networking for interconnection. During benchmarking, we paid particular attention to stress the tested applications adequately,but not bring the tested machine down to its knees as this would introduce unacceptable delays in I/O related operations. The maximum load on the workload machine did not climb to more than 95% of the machine's capacity.

5.1 Instrumentation

Figure 1: VolanoMark experiment results

We used the `dtrace(1)` tool exclusively for instrumenting the benchmark. In order to demonstrate better the overheads involved, we only instrumented the factors that were identified in the previous sections to contribute to the generation of guaranteed overhead. Those are the JNI-initiated data copying, the kernel initiated data copying and the time spent in the native threading services respectively. The latter was found to have only minimal impact and was therefore not included in the benchmark results.

For each service interface whose cost we wanted to assess, we collected and organized the respective library functions or system calls into a single `dtrace(1)` probe which calculated the total time in each thread spent in them. We took extra steps to prohibit a thread from entering more than one `dtrace(1)` probe at once. The memory related overheads were measured directly using the appropriate counters in the `dtrace(1)` probes. In the case of kernel copying functions, we followed the approximative approach described in Section 3.1.1. In any case, we avoided measuring I/O related activities, beyond the data copying stage, as this would introduce unacceptable unpredictability to our results.

We run the benchmark workload generators four times for each benchmark, each time increasing the workload. For the Tomcat benchmark, we concurrently increased both the number of threads and the number of retrieved web pages, leading to quadratic increase in the total workload. We could not do the same for the `volano` benchmark due to server hardware limitations; in that case, a more conservative approach of doubling the workload in each benchmark run was used.

5.2 Results

The results of the VolanoMark mark benchmark are presented in Figure 1. The guaranteed overhead ranges from 23% to 40% with a tendency to decrease as the

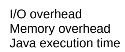

Figure 2: Tomcat experiment results

load increases. This can be justified by the heavily multithreaded execution profile of the benchmark. As the number of threads increases, so does the pressure on the runtime system, leaving little execution time for actual work. The VolanoMark benchmark takes multithreading to the extreme; on full load, there were 800 threads being executed in parallel. Also, when under full load, VolanoMark requires that on message receipt at least 20 threads should be waken up and use their open socket to push the incoming message to the client.

On the other hand, the results for the Tomcat benchmark were much closer to what we expected and to what the microbenchmarks indicated. The native execution overhead in that case ranges from 15% to 45%. The Tomcat application software is a production grade application that has gone through several optimization phases. It thus uses several techniques to minimize the overhead imposed by the communication with external resources, such as connection and thread pooling. The result is that almost no overhead could be measured, except for I/O.

5.3 Cost heuristic evaluation

In order to evaluate the heuristic we introduced in Section 4, we run the Tomcat experiment with an increasing number of connections. First, we used the values we calculated for our platform as input to the heuristic formula 4.6. The average page size for our modified JPetStore application was 133 I/O chunks. We set the C_{mem} cost to zero, as our application's heap size was considered stable. The real overhead was calculated by running the same analytical `dtrace` script as in our main experiment. The results are presented into Table 7. As it turns out, our heuristic is able to capture, with some variation, the cost of the OS on the JVM. However, as the number of served connections increases, the difference also increases. This may be due to the in-

# connections	Estimated Cost	Real Cost	Diff
100	72.5	77.1	5.9%
1000	725.0	851.0	14.8%
3500	2537.0	3024.0	16.1%

Table 7: Estimated vs Real Cost. Times are in milliseconds.

creased synchronization costs, which are not taken into consideration in our heuristic definition. More effort needs to be put towards analyzing the overhead caused by native synchronization primitives.

6. RELATED WORK

As with every layer of software, resource management layers incur some overhead to the layers running on top of them [23, 36, 25]. Studies of the performance of the JVM mainly evaluate the overhead of garbage collection [5, 20], or that of JVM-level threads running on top of OS threads [18, 4]. However, while the performance of JIT compilation and garbage collection has been subject to extensive study, there is not so much work in the direction of evaluating the performance of I/O in Java or accessing the overhead of overlapping responsibilities between the JVM and the OS.

The Java platform was one of the first general purpose programming environments to offer support for automatic memory management, with the introduction of garbage collection as a standard feature of the language. Due to its, initially, low performance, the garbage collection process [20, 5, 11] and the memory allocation patterns of Java programs [29, 35] have been studied extensively. Other researchers tried to employ the operating system's virtual memory system to co-operate with the JVM garbage collector in order to improve its performance [2, 31, 21]. The DaCapo suite of applications has emerged as the standard memory management benchmark [6].

In reference [10], Dickens evaluates the I/O capabilities of the Java platform in the context of scientific computing. The paper examines the basic I/O capabilities of the language, as these are exposed through the core API, and also proposes several ways to by-pass the Java's inability to write to direct memory buffers of any data type. The authors also benchmark the proposed solutions in multithreaded environments and conclude by proposing a stream oriented architecture for Java. In reference [7], Bonachea proposes a mechanism for performing asynchronous I/O in Java, also in the context of scientific computing. The Java platform is found to offer very weak I/O performance. Finally, Welsh and Culler [33], present Jaguar, a mechanism to access raw memory and operating system services such as memory-mapped files without the need of the JNI layer. Jaguar performs I/O by translating predefined bytecode sequences into inlined library calls, which in turn can perform operations directly on hardware devices. Jaguar was targeted to a custom, high-performance I/O architecture prototype and not to generic hardware, though. A lightweight evaluation of the JNI overhead is also presented in the paper. A good overview of the default Java platform I/O capabilities is presented in reference [4]. The authors use both blocking and non-blocking I/O to implement server software that they expose to high client volume. The paper presents evidence that heavy threading causes significant performance degradation as a consequence of context switching, although the authors do not identify the cause.

The JikesXen Java runtime environment borrows ideas from exokernel systems [12] and bare metal JVMs [15]. However, it can not be categorised as either of the former, although it bares some resemblance with type safe operating systems [14, 22]. The basic idea behind exokernel systems is to move resource management to the application instead of the system kernel, which is effectively reduced to a hardware sharing infrastructure. Applications use library OSs to help them communicate with the hardware. Similarly, JikesXen manages computing resources inside the VM, but it does not require an external libOS, as all functionality is self-contained. Bare metal JVMs and type safe OSs must implement driver infrastructures and protect shared subsystems from concurrent access, as they are designed to run multiple concurrent applications. On the other hand, JikesXen will be required to implement drivers for the Xen virtual devices, but since the hypervisor interface is the same across all supported hardware no extra layers of infrastructure software will need to be developed. Several optimisation opportunities emerge from this architecture while resource management will be greatly simplified.

Finally, Libra [1] is a recent effort to build an a system that is in many aspects similar to JikesXen. Libra is a library operating system designed to support slightly modified applications to run as DomU Xen partitions. The interesting idea behind it is that instead of using the Xen virtual devices, it defers I/O to an Inferno 9P server running on Dom0. The IBM J9 JVM has been successfully ported on Libra. The performance figures presented in the paper show a very good speedup in heavy multithreaded applications but performance is not as good when considering I/O against the standard J9 on Linux setup. BEA systems has implemented a surprisingly similar, in principle, "bare-metal" JVM [9], but little information is known about it.

7. CONCLUSIONS

We examined the runtime overhead imposed by the operating system and the native execution environment on the Java programming language. Our work demonstrates that performing I/O in Java is an expensive operation, due to limitations of the current interfacing mechanisms employed by the JVM and the structure of current operating systems. Memory allocation and threading also affect the JVM performance, but their effect is mostly apparent in short running applications. A simple heuristic for calculating the total overhead of threaded applications was also presented in the paper and, was found adequately accurate. We also presented selected bits of our work on the JikesXen Java runtime environment, which targets the exact problems we have presented in this work.

Acknowledgements

This work is partially funded by the Greek Secretariat of Research and Technology thought,the Operational Programme COMPETITIVENES, measure 8.3.1 (PENED), and is co-financed by the European Social Funds (75%) and by national sources (25%) and partially by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software" (SQO-OSS).

8. REFERENCES

[1] G. Ammons, J. Appavoo, M. Butrico, D. D. Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. V. Hensbergen, and R. W. Wisniewski. Libra: a library operating system for a JVM in a virtualized execution environment. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 44–54, New York, NY, USA, 2007. ACM Press.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 11–20, New York, NY, USA, 1988. ACM Press.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[4] S. Beloglavec, M. Hericko, Matjaz, B. Juric, and I. Rozman. Analysis of the limitations of multiple client handling in a java server environment. *SIGPLAN Not.*, 40(4):20–28, 2005.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement*

and modeling of computer systems, pages 25–36. ACM Press, 2004.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM Press.

[7] D. Bonachea. Bulk file I/O extensions to Java. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 16–25, New York, NY, USA, 2000. ACM Press.

[8] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 15–28. USENIX, 2004.

[9] J. Dahlstedt. Bare metal - speeding up Java technology in a virtualized environment. Online, 2006.

[10] P. M. Dickens and R. Thakur. An evaluation of Java's I/O capabilities for high-performance computing. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 26–35, New York, NY, USA, 2000. ACM Press.

[11] L. Dykstra, W. Srisa-an, and J. Chang. An analysis of the garbage collection performance in Sun's Hotspot Java virtual machine. In *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference*, pages 335–339, Phoenix, AZ, USA, April 2002.

[12] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.

[13] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–46, June 1974.

[14] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The jx operating system. In *Proceedings of the USENIX 2002 Annual Technical Conference*, pages 45–48, June 2002.

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

[16] G. Gousios. The JikesXen Java server platform. In *Companion to the 23rd OOPSLA (Doctoral Symposium)*, Oct 21-24 2007. (to appear).

[17] G. Gousios, V. Karakoidas, and D. Spinellis. Tuning Java's memory manager for high performance server applications. In I. A. Zavras, editor, *Proceedings of the 5th International System Administration and Network Engineering Conference SANE 06*, pages 69–83. NLUUG, Stichting SANE, May 2006.

[18] Y. Gu, B. S. Lee, and W. Cai. Evaluation of Java thread performance on two different multithreaded kernels. *SIGOPS Oper. Syst. Rev.*, 33(1):34–46, 1999.

[19] C. Hawblitzel and T. von Eicken. Luna: a flexible Java protection system. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 391–401, New York, NY, USA, 2002. ACM Press.

[20] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 313–326, New York, NY, USA, 2005. ACM Press.

[21] M. Hertz, Y. Feng, and E. 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.

[22] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fa?hndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Microsoft Research Technical Report MSR-TR-2005-135 MSR-TR-2005-135, Microsoft Research, 2005.

[23] S. T. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *Proceedings of the General Track:2003 USENIX Annual Technical Conference*, pages 71–84, San Antonio, Texas, USA, June 2003. USENIX.

[24] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, first edition, Jun 1999.

[25] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, New York, NY, USA, 2005. ACM Press.

[26] K. Palacz. JSR 121: Application isolation API specification. Java Community Process, Jun 2006.

[27] K. Palacz, J. Vitek, G. Czajkowski, and

L. Daynas. Incommunicado: efficient communication for isolates. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 262–274, New York, NY, USA, 2002. ACM Press.

[28] M. Reinhold. JSR 51: New I/O APIs for the Java platform. Java Specification Request, May 2002.

[29] T. Skotiniotis and J. en Morris Chang. Estimating internal memory fragmentation for Java programs. *Journal of Systems and Software*, 64(3):235–246, December 2002.

[30] J. E. Smith and R. Nair. The architecture of virtual machines. *IEEE Computer*, 38(5):32–38, May 2005.

[31] D. Spoonhower, G. Blelloch, and R. Harper. Using page residency to balance tradeoffs in tracing garbage collection. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 57–67, New York, NY, USA, 2005. ACM Press.

[32] VMware. Vmware esx server 2: Architecture and performance implications. Vmware white paper, VMware, 2005.

[33] M. Welsh and D. Culler. Jaguar: enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, Aug 2000.

[34] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

[35] Q. Yang, W. Srisa-an, T. Skotiniotis, and J. Chang. Java virtual machine timing probes: a study of object life span and garbage collection. In *Conference Proceedings of the 2002 IEEE International Performance, Computing, and Communications Conference*, pages 73–80, Phoenix, AZ, USA, April 2002.

[36] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: taking real memory into account. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 61–72, New York, NY, USA, 2004. ACM Press.