RC 21715 2/3/2000
Computer Science/Mathematics

IBM Research Report

An Evaluation of Java System Services with Microbenchmarks

Eugene Gluzberg **Stephen Fink**

IBM Research Division
Thomas J. Watson Research Center
PO Box 218
Yorktown Heights, NY 10598

Limited Distribution Notice
This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/CyberDig.nsf/home. Copies may requested from IBM T.J. Watson Research Center, 16-220, P.O. Box 218, Yorktown Heights, NY 10598 or send email to reports@us.ibm.com.
IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

An Evaluation of Java System Services with Microbenchmarks

Eugene Gluzberg Stephen Fink
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{gluzberg,sjfink}@us.ibm.com

Abstract

The Java programming language and standard libraries provide a portable interface to traditional operating system services. We present a set of microbenchmarks to help evaluate the performance of these services. Building on previous work on operating system microbenchmarks, we present an "apples-to-apples" comparison of Java and C performance of memory bandwidth, file system, thread, and network performance tests. We present experimental results for several Java runtime systems on Windows NT, which show both strengths and weaknesses of current Java implementations.

1 Introduction

The Java programming language, with its "write-once, run anywhere" philosophy, has provided a new target for writing portable applications. While Java's portability is a key selling point for the language, it comes at the cost of potential performance degradation. As Java matures as a platform, the Java community needs to understand the sources of the degradation, and apply new technology to fix problems where possible.

To promote portability, the Java standard library provides a portable Application Programmer Interface to traditional operating system services. Many resource-intensive server applications depend heavily on operating system (OS) services, such as file systems, network protocols, and remote procedure call; the performance of these services constrains the performance of the application. To understand server application performance, we must understand the performance of the Java interface to system services.

Several previous studies [12, 3, 14] have presented microbenchmarks as tools to help analyze performance of operating system services. In this paper, we apply this methodology to Java, and report the results. Specifically, we describe a suite of Java microbenchmarks from *The jMocha Microbenchmark Framework and Suite for Java*, and provide experimental results comparing performance on these microbenchmarks to corresponding tests written in C using native OS system calls. We have endeavored to provide a close "apples-to-apples" comparison between native and Java system services, which to our knowledge, has not yet appeared in the literature.

We present microbenchmark results that assess memory bandwidth, file system, thread services, and various flavors of network performance. The results show that with current technology, the best Java platforms currently approach C performance initializing and reading arrays. Java performance is currently on a par with native system calls for unformatted I/O, but Java's formatted I/O suffers large performance degradation. The results show that thread services and TCP performance in the best Java platforms is close to native performance with C. However, Java UDP and especially RMI performance lags far below native UDP and RPC performance, respectively.

The remainder of this paper proceeds as follows. Section 2 reviews related work on operating system and Java benchmarks. Section 3 presents our experimental methodology, and Section 4 presents experimental results. Section 5 summarizes our results, and suggests directions for future work.

2 Related Work

Prior to Java, several projects developed microbenchmark suites to evaluate OS performance. Ousterhoust [14] developed a set of microbenchmarks to help evaluate the Sprite operating system. His work exposed memory system performance, disk cache structure, and network protocols as crucial factors in OS performance. McVoy and Staelin [12] developed lmbench, a portable set of operating system benchmarks focusing on issues including memory system, IPC, cached I/O, system call, signal handling, network, and process/thread performance. Brown and Seltzer [3] built further on this work with hbench:OS, which improved some methodological and practical issues of lmbench. We build directly on this line of research, comparing benchmarks similar to those of hbench:OS with comparable Java versions.

Other related work, pre-Java, includes microbenchmarks presented by Saavedra and Smith [15]. This work presents microbenchmarks to evaluate memory system performance, exposing performance at the various levels of a computer system's memory hierarchy.

A number of Java benchmark suites have appeared. The Jmark suite [16] provides compil-

er/CPU "processor tests", as well as a set of tests of AWT graphics performance. The SPECjvm98 suite [5], perhaps the most-quoted Java benchmark, also provides tests of compiler and CPU performance. Both of these approaches target client codes, and shed little light on server and OS performance issues.

Haydon and Najork [9] looked at performance limitations of the Java core libraries. Carpenter et al. [4] looked at serialization performance of Java. Mathew, Coddington and Hawick [11], and Bull et al. [7] compared the performance of Java on different Virtual machines and platforms in 6 "Java Grande" benchmark packages: arithmetic, assignment, casting, garbage collection, math, and method call. Nester et al. [13] evaluates the performance of Java's RMI and proposes an alternative implementation. Although all of these works evaluate the performance of different Java operations, none of them compare Java to that of native code.

Judd et al. [6] looked at the cost of communication in a Java implementation of MPI as compared to the native version.

A few benchmarks target server issues, including database, web server, and network performance. Baylor et al. [8] present a set of such benchmarks developed at IBM, as well as a brief discussion of microbenchmarks (including a preliminary version of the work presented here.) The VolanoMark [10] benchmark tests the performance of a chat server application, focusing on thread and network performance issues.

3 Methodology

Java presents the illusion of a portable operating system, which adds another layer of software to the OS services, To evaluate the performance of OS system services in Java, we build on the line of research culminating in hbench:OS. As part of jMocha, we have developed a set of Java versions of relevant hbench:OS tests, relying on calls to the standard Java APIs where needed. We compare the performance of the native hbench:OS tests, written in C with direct system calls, to the Java versions.

For Windows NT, we ported the hbench:OS tests, which rely on UNIX system calls, to use the Win32 API. In running both jMocha and hbench:OS benchmarks we utilize hardware cycle counters on the PowerPC and Pentium chips to obtain accurate timing information. In this work, we present only timing information; the infrastructure also supports instrumentation of various hardware events as collected by the respective CPUs.

In this paper, we restrict our attention to fine-grained single-threaded benchmarks, and present results measuring memory bandwidth, cached file read and write bandwidth, thread creation/destruction latency, TCP and UDP network performance, and RPC/RMI performance. Results are only presented for Windows NT. Specific details about each test appear in Section 4.

4 Results

We present microbenchmark results for an IBM Intellistation Z–Pro machine (6899–120), with an Intel 200 MHz Pentium Pro processor, 256 MB of RAM, running NT version 4.0, service pack 4. On this platform we compare results using a IBM Developer Kit for Windows(R), Java(TM) Technology Edition Version 1.1.8 (IBM DK) (release 07/28/99 with IBM Just In Time compiler (JIT) version 3.5), Sun Java(TM) Development Kit 1.1.8 release M and Sun Java 2 SDK, Standard Edition, v 1.2.2 release W (both using the Symantec JIT), Microsoft Software Development Kit (SDK) version 5.0, and C tests compiled with Microsoft Visual C++ version 3.1 using the following compiler flags: `/O2 /Oa /G6 /GD /GM /MT -DNO_PORTMAPPER -D_WIN32_WINN=0x0400` .

In each trial, we repeated each test for a number of iterations chosen so that the total running time exceeded one second. Each trial included a warm–up stage, not timed, so initial paging and JIT activity do not influence the reported results. The results presented are the 10% trimmed mean of ten trials for each data point. Standard deviations for all results were negligible ($< 5\%$ of mean). Of the Java platforms, we could examine the JIT-produced code for the IBM DK, but not for the other JITs.

We present results from a representative subset of the full jMocha suite. The following subsections evaluate Java performance for array copy, cached file system, network, and remote procedure call. The C implementations of these tests, derived from hbench:OS, are described in [3]. We describe the corresponding Java implementations in the following subsections.

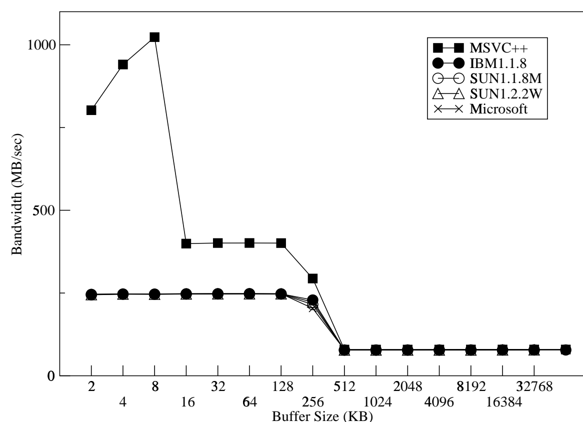
4.1 Memory Bandwidth

Previous research has demonstrated that memory bandwidth is a crucial factor in many aspects of operating system performance [14, 12]. We present results from a several tests to measure bandwidth when reading and writing memory. The first test initializes every element of an integer array. Figure 1 shows the Java kernel of this microbenchmark. Figure 2 shows the results of this test.

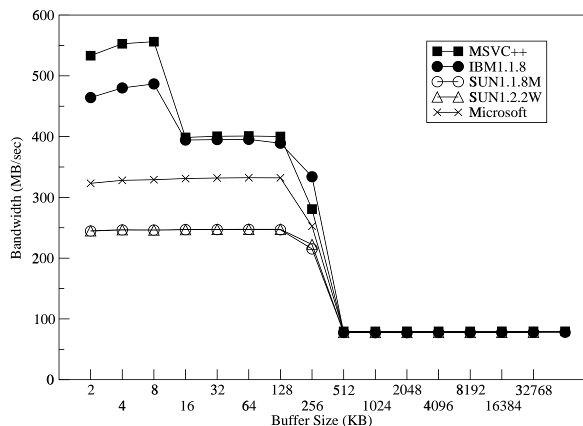
```
int [] mem = new int[bufferSize];
for (int i=numIter; i>0; i--) {
 for (p=0; p<bufferSize; p++) {
   mem[p] = 1;
 }
}
```

(a) simple loop

Figure 1: Array Initialization Kernel (simple)



(a) simple loop



(b) unrolled loop

Figure 2: Results for Array Initialization Test

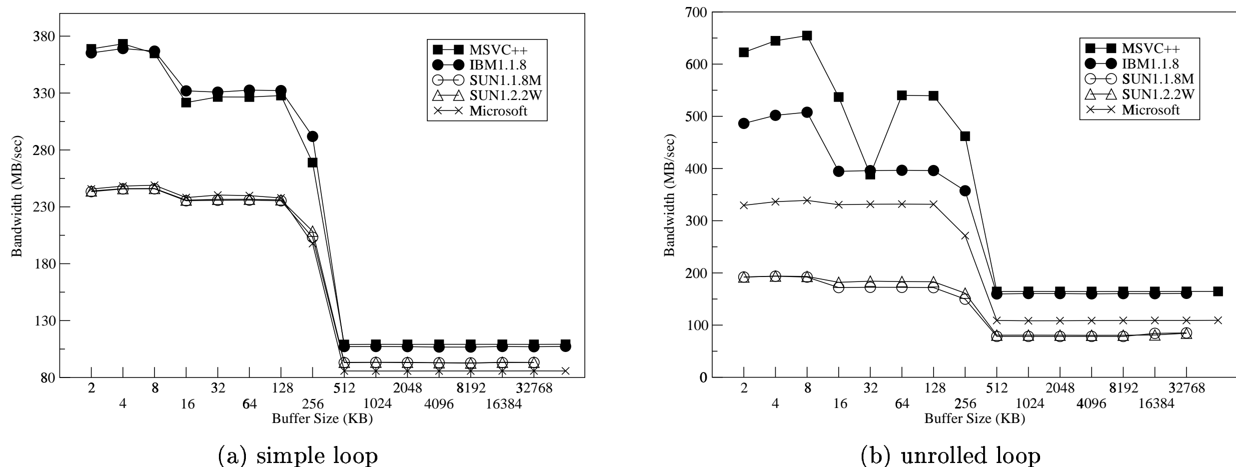
In the simple version of the test, the C results clearly show three levels of the memory hierarchy, corresponding to L1 cache, L2 cache, and main memory. All the Java platforms significantly lag behind the C performance in L1 cache. We note that the C compiler replaces the entire loop of

Figure 1 with a single x86 string move instruction. The IBM JIT does not recognize this kernel opportunity. Note that Java requires array bounds checks for the loop. The IBM JIT, and most likely other compilers, optimizes away these array bounds checks, using versioning to create a safe, check-free loop. Note that when the array no longer fits in L2 cache, the high latency of memory access dominates all other factors, and C and all Java platforms perform equivalently.

Figure 2(b) shows performance on a version of the array initialization test with the loop unrolled 16 times. Compared to the simple test, the unrolled C performance suffers because the compiler cannot find the opportunity to turn the initialization into one instruction. The Java performance improves on the IBM and the Microsoft Java implementations. Examining the code from the IBM JIT, we observed that the compiler improves performance by eliminating address arithmetic and using better x86 instruction selection than in the simple loop case.

To measure performance reading from memory, the next test sums an array of values into a local variable. The simple version of the test is identical to Figure 1, replacing the line that initializes a memory location `mem[p] = 1` with `a += mem[p]` . We also consider a version that unrolls the loop 16 times.

Figure 3 shows the results. As in the array initialization test, the C results clearly show the three levels of cache. The IBM JIT matches the performance of the C compiler on the simple loop. In this case, the C compiler cannot reduce the loop to a single instruction. The other Java platforms do not perform as well. The unrolled loop nearly doubles performance for the C compiler. The performance on the Java platforms varies, with the best result (IBM) lagging behind C by roughly 15% in L1 cache, and the differential between C and Java decreases in L2 cache and main memory.



(a) simple loop

(b) unrolled loop

Figure 3: Results for Array Read Bandwidth

4.2 File I/O Bandwidth

Memory bandwidth plays a large role in performance of OS services that predominantly move data, such as file system performance. We now evaluate Java performance for reading and writing a file in the file cache. Figure 4 shows a code fragment for a jMocha test that reads from a cached file using a FileInputStream. We have a similar test for file write bandwidth using a FileOutputStream.

We choose fileSize = 8MB for read tests, and fileSize = 4MB for write tests, which on our platform, assures that the file fits in the file buffer cache, and avoids disk access. For these benchmarks we read or write the entire file only once for each trial.

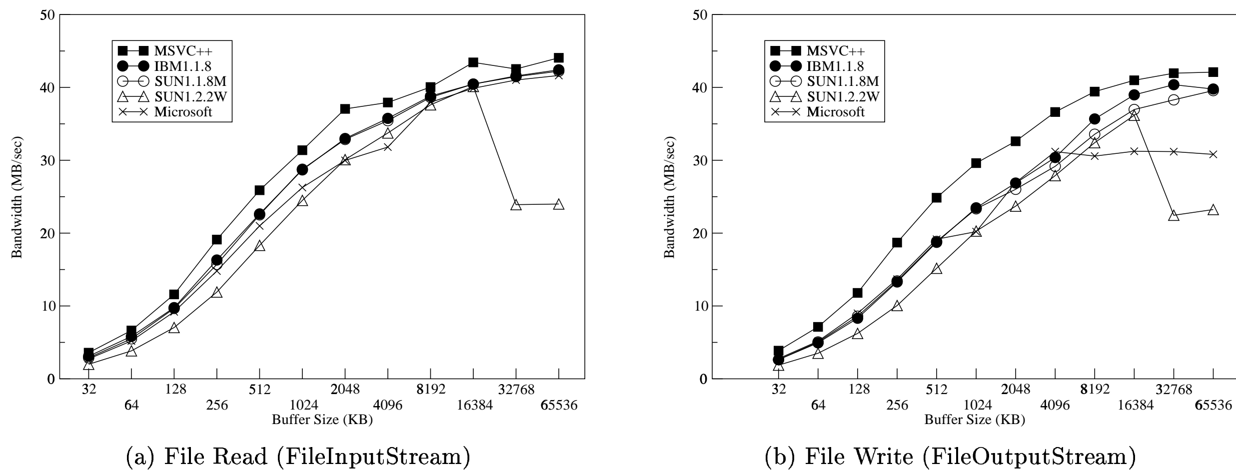
Figure 5 shows the results of these tests. Java performance on these benchmarks is compet-

```
FileInputStream f = new FileInputStream(fileName);
byte [ ] buffer = new byte[fileSize];
int size = 0;
while (size < fileSize) {
  f.read(buffer,size,blockSize);
  size += blocksize;
}
```

Figure 4: Java File Read Bandwidth Kernel (FileInputStream)

itive with C for all Java platforms considered. We conclude that the FileInput/OutputStream implementation is a thin layer over the native OS service, and does not greatly hinder performance.

Java also provides classes to perform formatted file I/O. Figure 6 shows a version of the file read benchmark, using a FileReader class, which reads a file as a sequence of characters, forcing the system to interpret the input data. We also consider a similar test using a FileWriter.



(a) File Read (FileInputStream)

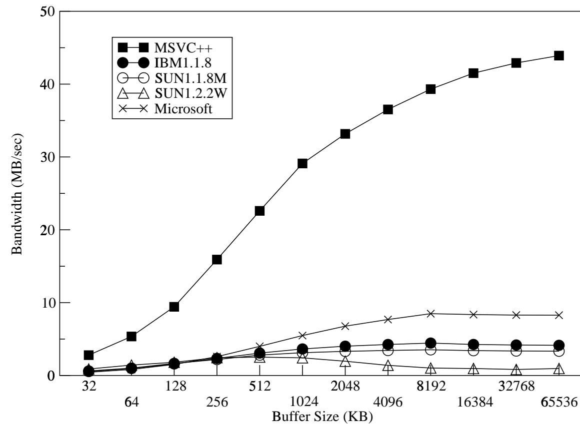
(b) File Write (FileOutputStream)

Figure 5: Results for File Bandwidth (file streams)

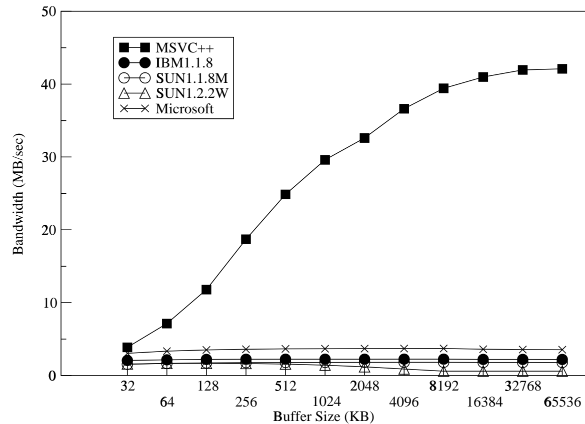
Figure 7 shows the results of these tests, compared to the native unformatted C I/O. For all Java platforms, performance suffers drastically when using the formatted I/O. Every file read (using the Reader classes), and file write (using the Writer classes), must convert the native byte-size characters to the Java two-byte unicode representation or vice versa. The performance of the Java byte-to-unicode and unicode-to-byte conversion routines dominate this test. For Java formatted I/O to compete with native services, Java vendors must address this problem.

```
FileReader f = new FileReader(fileName);
byte [ ] buffer = new byte[fileSize];
int size = 0;
while (size < fileSize) {
  f.read(buffer,size,blockSize);
  size += blocksize;
}
```

Figure 6: Java File Read Bandwidth Kernel (FileReader)



(a) File Read (FileReader)



(b) File Write (FileWriter)

Figure 7: Results for File Bandwidth(Readers/Writers)

4.3 Thread Creation/Destruction

Some server codes such as chat servers [10], transaction processing codes [8], and web servers [2] create a large number of threads. We consider a benchmark that compares the Java performance creating and destroying threads to the equivalent C code using native services. The benchmark repeatedly creates and destroys a thread to determine the combined overhead. Figure 8 shows a fragment of this test.

```
Thread t;
for (int i = niter; i > 0; i--) {
  (t = new emptyThread()).start();
  t.join();
}
```

Figure 8: Java Thread Spawn Kernel

Figure 9 shows the results for this benchmark. The results show that for all Java platforms, the cost of this thread exercise is significantly slower than the cost using native NT services. We speculate that all these Java platforms implement Java threads as native threads, and that the Java run-time system imposes non-trivial overhead in managing the underlying OS threads. We speculate that a Jvm with a custom lightweight thread system, such as Jalapeno [1], will reduce this type of thread overhead, which may be important for codes that create many threads dynamically

on demand.

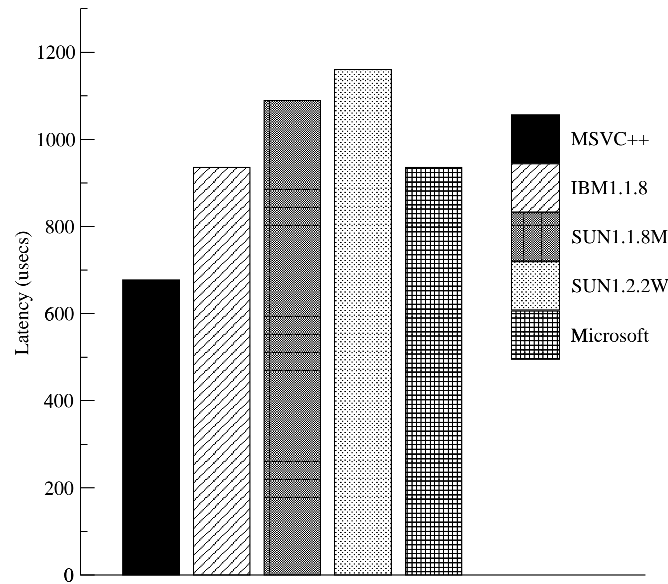


Figure 9: Results for Thread Spawn Latency

4.4 Network Performance

Several jMocha tests evaluate the performance of Java's network functions. We present results to evaluate three areas of network performance: TCP bandwidth, TCP latency, and UDP latency. In this paper, we present only loopback results, where both the client and server reside on the same physical machine.

The TCP bandwidth benchmark tests one way write bandwidth. Two Jvms are created on the machine; the client Jvm connects to the server Jvm and writes data through. Code fragments for this test appear in Figure 10.

```
OutputStream os =
 socket.getOutputStream();
while (nbytes > 0) {
 os.write(buffer,0,msgSize);
 nbytes -= msgSize;
}
```

(a) Client

```
InputStream is =
 socket.getInputStream();
while (nbytes > 0) {
 actual=is.read(buffer,0,msgSize);
 nbytes -= actual;
}
```

(b) Server

Figure 10: Java TCP Bandwidth Kernel

The current Jvms implement the socket read and write methods as native methods, which interact with the native operating system interface. The implementation and tuning of these native methods dictate the Jvm performance on these tests. Figure 11 shows the results of this test on the two platforms.

Surprisingly, the results show that two Java platforms outperform the C implementation on the TCP bandwidth test. We do not currently have an explanation for this phenomenon.

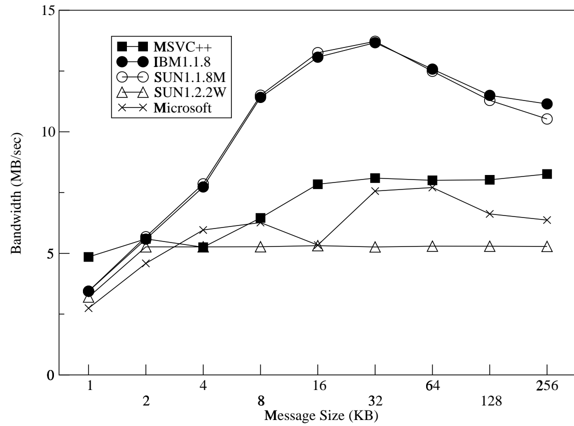


Figure 11: Results for TCP Bandwidth

We next present a TCP latency benchmark that times the latency of a round trip of one byte. As with the TCP bandwidth benchmark, this test involves a client and server Jvm. We present loopback results; Figure 12 shows code fragments for the client and server.

```
OutputStream os =
 socket.getOutputStream();
InputStream is =
 socket.getInputStream();
dataout[0] = 1;
for(i=0; i < num_iterl i++) {
 os.write(dataout,0,1);
 is.read(datain,0,1);
}
```

(a) Client

```
OutputStream os =
 socket.getOutputStream();
InputStream is =
 socket.getInputStream();
is.read( datain, 0, 1);
while (datain[0] > 0) {
 os.write( dataout,0,1);
 is.read(datain,0,1);
}
```

(b) Server

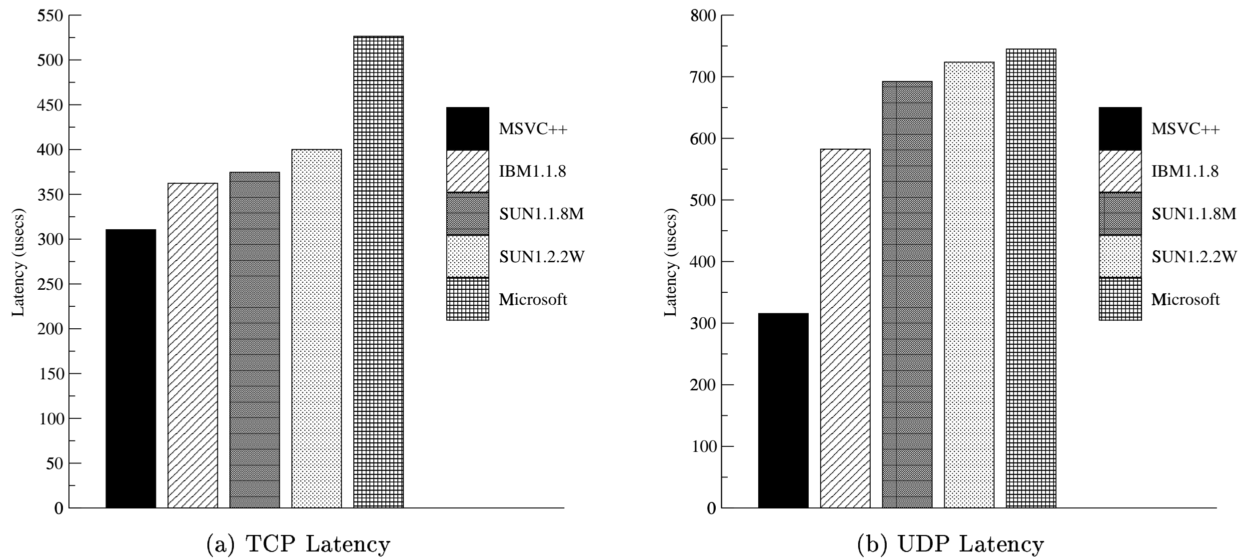
Figure 12: Java TCP Latency Kernel

As with the TCP bandwidth benchmark, the results for this test depend on the implementation of the native send method. Figure 13(a) shows the results.

The best Jvms perform within 20% of the native C implementation. We hypothesize that native method call overhead contributes to the difference.

Finally, we present a test to measure the round trip latency of a UDP packet. In this test, we send a sequence number with every packet, and verify that the packets arrive in order. For every UDP message, this test creates a new DatagramPacket object. We store the integer sequence number in the packet. Since the DatagramPacket accepts a byte array, we must convert the integer to a String, then the String to a byte array. The receiver performs the reverse conversion. (Code not shown). The C version of this benchmark avoids these conversion, using pointers to bypass the language type system. Figure 13(b) shows the results.

The Figure shows that Java lags behind C by approximately a factor of 2. The Java overheads of memory management for short-lived DatagramPacket objects, and the type conversions, exact a heavy price on UDP performance. In fact, these language overheads dominate the overheads of the network protocol. Although the programmer can reduce the overhead by reusing the DatagramPacket objects from one network transmission to the next, we believe using new packet objects for each transmission is a more natural Java coding style.



(a) TCP Latency

(b) UDP Latency

Figure 13: Results for Network Latency

4.5 RMI performance

We compare the performance of a simple Java Remote Method Invocation (RMI) call to an equivalent Remote Procedure Call (RPC). The Java kernel for this microbenchmark appears in Figure 14. The Figure shows code for a client that invokes a method xact on a server via RMI. The method simply returns the integer value 123.

```
for (i=num_iter;i>0;i--) {
  result = server.xact();
  if ( result != 123 )
    throw new
     LatRMIException("Invalid Data: " + result );
}
```

Figure 14: Java RMI Latency Kernel

During the benchmark runs, the RMI call always succeeded, and the client never threw the LatRMIException. Figure 15 show the performance of this program, and the corresponding RPC program. Note that the Microsoft SDK does not support RMI, so we do not present numbers for the Microsoft SDK.

Figure 15 shows four C results, corresponding to four RPC transport mechanisms provided by the operating system: local, named pipes, UDP, and TCP. The C results show that the choice of transport mechanism dictates the RPC latency. For these loopback tests, the native local transport mechanism reduces latency by a factor of 9 compared to native TCP. The three Java RMI results show significant performance degradation, with the best RMI latency a factor of 4 slower than the worst RPC latency.

The object-based RMI protocol entails more overhead than the function-based RPC protocol, which helps account for the Java performance penalty. In future work, we will extend the jMocha suite in an attempt to pinpoint bottlenecks in the RMI protocol. Furthermore, we will also compare RMI to CORBA, a more similar object-based communication protocol.

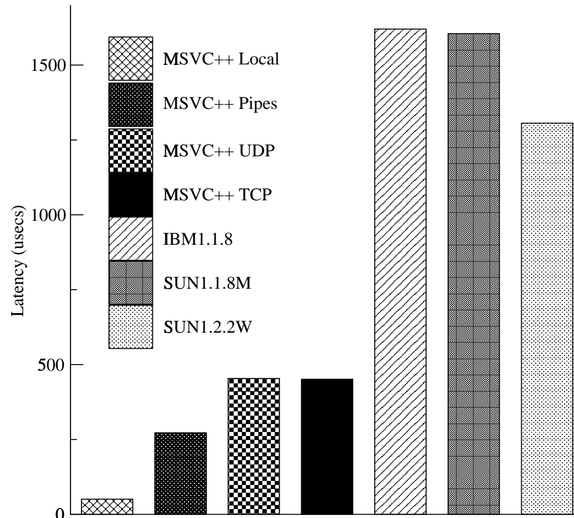


Figure 15: Results for RMI Latency

5 Conclusion

We have described jMocha, a set of microbenchmarks to evaluate performance of typical operating system services written in Java. The results show that with current Java implementations for NT,

- the best Java platforms currently approach C performance initializing and reading arrays,
- Java performance rivals native system calls for unformatted I/O,
- Java's formatted I/O imposes large overheads,
- simple Java thread services impose a small overhead compared to native thread services,
- TCP performance in the best Java platforms is close to native performance, and
- Java UDP and especially RMI performance lags far below native UDP and RPC performance, respectively.

Microbenchmarks such as we present can serve a limited but important role in improving Java performance. Microbenchmarks help to identify and isolate system problems, and good microbenchmarks give JVM implementers a small, fixed target to optimize for. However, microbenchmarks only give a small picture of total system performance. In future work, we will examine real codes, which expose interactions and issues that do not appear in small fixed codes. We will evaluate whether the microbenchmarks we have presented correlate with performance in real codes, and expand our set of microbenchmarks to more closely model system issues that appear.

For the final version of this paper, we expect to announce public availability of the jMocha microbenchmark suite.

References

[1] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *OOPSLA*, 1999.

[2] E. Bayeh. The WebSphere Application Server archiecture and programming model. *IBM Systems Journal*, 37(3):336–364, 1998.

[3] Aaron B. Brown and Margo I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. *Performance Evaluation Review*, 25(1):214–224, June 1997.

[4] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 66–71, San Francisco, CA, June 1999. ACM Press.

[5] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[6] Glenn Judd et al. Design issues for efficient implemintation of MPI in Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 152–159, San Francisco, CA, June 1999. ACM Press.

[7] J. M. Bull et al. A methodology for benchmarking Java Grande applications. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 81–88, San Francisco, CA, June 1999. ACM Press.

[8] Sandra J. Baylor et al. Java Server Benchmarks. *IBM Systems Journal*, 39(1), Febuary 2000. Special issue on Java Performance.

[9] Allan Heydon and Mark Najork. Performance limitations of the Java core libraries. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 35–41, San Francisco, CA, June 1999. ACM Press.

[10] Volano LLC. VolanoMark 2.0. http://www.volano.com/benchmarks.html.

[11] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 72–80, San Francisco, CA, June 1999. ACM Press.

[12] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of USENIX 1996*, pages 279–294, June 1996.

[13] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient rmi for java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 58–65, San Francisco, CA, June 1999. ACM Press.

[14] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware. In *Proceedings of USENIX Summer Conference*, pages 247–256, June 1990.

[15] Rafael H. Saavedra and Alan J. Smith. Measuring cache and TLB performance and their effect on benchmark run times. *IEEE Transactional Computing*, 44(10), Oct 1995.

[16] ZDNet. JMark 2.0. http://www.zdnet.com/zdbop/jmark/jmark.html, Jan 1999.