A Performance Study of Java file I/O

Nathan Burnett, Deepak Jindal
University of Wisconsin-Madison
{ncb, jindal}@cs.wisc.edu

1 Abstract

Java's performance for applications that are compute bound has been widely studied and the affects of the JVM on such performance are well understood. However, very little has been done to study how Java's performance for I/O bound applications compares to non-Java implementations. In this paper, we attempt to first compare Java's I/O performance to non-Java programs performing similar tasks. We then attempt to determine whether the performance difference is simply due to the same factors affecting Java's compute-bound performance or whether there is additional overhead introduced by Java when performing I/O operations.

2 Introduction

In this paper we look at the performance of I/O bound programs written in Java, running under Sun Java Virtual Machine. This work is motivated by Java's increasing acceptance in both industry and academia, as well as the increasingly I/O bound nature of user applications.

Java's write-once, run-anywhere paradigm and its suitability for object-oriented software design are making Java increasingly popular. Java is gradually supplanting C and C++ as the language of choice for undergraduate education. Furthermore, the pervasiveness of the Internet is making cross-platform compatibility more important than ever. Java makes running on more than one platform nearly trivial. This is leading to Java's increasing use in Internet-centric applications. Thus, it is important to understand the tradeoffs in choosing between Java and traditional programming languages. The performance issues surrounding compute bound Java applications are well understood. However, little work has been done for I/O bound applications. Here we examine the tradeoffs that one encounters in writing I/O bound applications.

Some work has been done to discover techniques that the application programmer can employ to improve Java's I/O performance [2]. In this study, we aim to discover ways in which Java's I/O library classes could be modified or replaced in order to improve performance. Figure 1 shows how the user-time of a sequential read of a file differs between a C and a Java implementation. Both benchmarks read the file from beginning to end. The Java benchmark used the BufferedReader class provided by the Java API with a 4K buffer size, the C benchmark made read() calls, reading 4KB per call. This plot clearly shows a large performance difference between the C and Java implementations. The goal of our study is to discover ways to significantly improve the performance of the Java implementation.

All of the Java benchmarks described herein were compiled and run using Sun's JDK 1.2.2. C programs and Native methods were compiled with the GNU C Compiler (gcc) version XXXXX. Most benchmarks were run on a XXXXX running RedHat Linux 7.0, Linux kernel 2.X.X. Random read benchmarks were run on a 600 Mhz Pentium III running RadHat Linux 6.2, Linux kernel 2.2.16. For the benchmarks that used buffering, the buffering was done within the application and not in the OS.

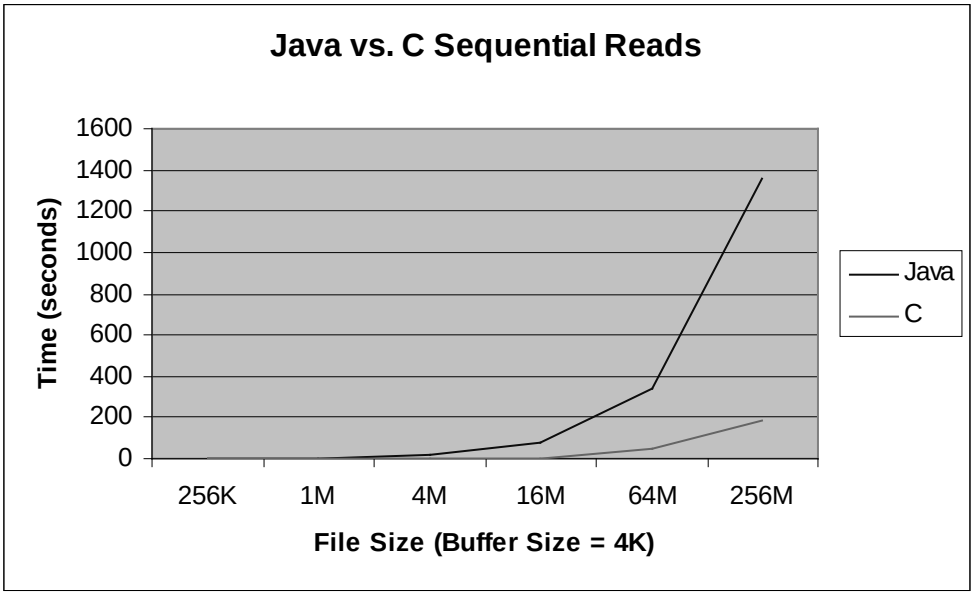

Figure 1 sequential reads in Java vs. the same in C.

3 The Java I/O path

Before we start any performance analysis of the situation depicted in Figure 1, it is important to understand the path of a read in Java. At the top is the JVM that is responsible for bytecode interpretation. In Java each class is compiled in a platform independent JVM instruction set, known as bytecode. To achieve portability, JVM bytecode does not include instructions to interact with the operating system. Consequently, each operating system call is wrapped in a Java method which is implemented natively, that is, in a platform dependent way. These native methods need to access objects and variables from JVM. Different JVMs can have different internal representation of objects. To maintain portability between JVM and the native implementations, Sun [3] has defined a standard Java Native Interface (JNI). JNI allows one to write native methods that can be used by any JVM. We explain JNI later in this report. Once the control of execution is transferred to native method, the native method uses standard OS interface to perform any system activity.

3.1 OS interface

All of our benchmarks used the Linux read(), write() and lseek() calls respectively. We do not explore this interface as much work has been done on both the interface to operating system kernels and the inner workings of the kernel itself. All of our work was done on top of the Linux second extended filesystem, which we also do not explore in this paper.

3.2 Java Native Interface

At this point there are two paths we could have chosen toward our goal of improved Java I/O performance. One would require us to modify the Java Virtual Machine in order to improve performance. The other path, and the one we chose, was to implement our own file I/O class using the Java Native Interface, thus bypassing most of the above described layers of function calls.

Figure 2 shows the results of our initial attempt at using the Java Native Interface to implement a file I/O class. This benchmark randomly seeks to a position in the file, reads a single byte and then randomly seeks again. Our initial implementation was clearly somewhat naïve since it performed worse than the implementation provided as part of the Java API. To understand why this is so, we look at the basic usage of the Java Native Interface.

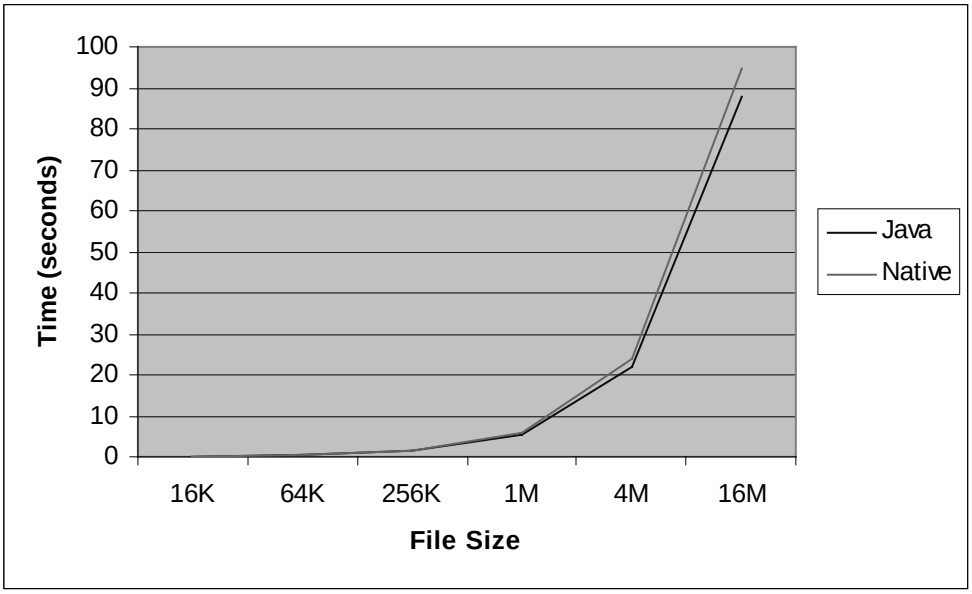

Figure 2 Random reads using a naïve native read implementation compared to the standard Java implementation

```
Java_MyRead_read(JNIenv *env, jobject obj) {
     cls = env->Get0bjectClass(obj);
     fid = env->GetFieldID(cls, "fd", "I");
     fd = env->GetIntField(obj, fid);
     . . .
     read(fd, &buf, buflen);
```

```
        return buf;
}
```

In the above code the first three lines are the important ones. In these lines we're retrieving the value of a data member store in the Java object of which this function (method) is a member. The first line fetches the class of the current object from the JVM environment. The second line fetches the ID of the field (data member) and the third line fetches the value we're interested in, in this case the Unix file descriptor of the file being read. Since this particular value will be the same every time this method is called from this object, there is no reason to spend the time to fetch the value on every call to read. The class type and field ID will also be the same for every call. We discovered that the retrieval of the field ID is particularly expensive, probably due to passing in the field name and type as strings. To solve this problem, we made the constructor of our file I/O class a native method and fetch these three values and store them in C global variables when the object is constructed. Figure 3 shows the results for sequential reads when we cache the cls, fid and fd values rather than fetching them on every call to read(). This gains a significant performance improvement over the Java implementation and over our initial native implementation, however the performance is still significantly worse than the implementation in C. The remainder of our study is focused on locating the source of this slowdown.

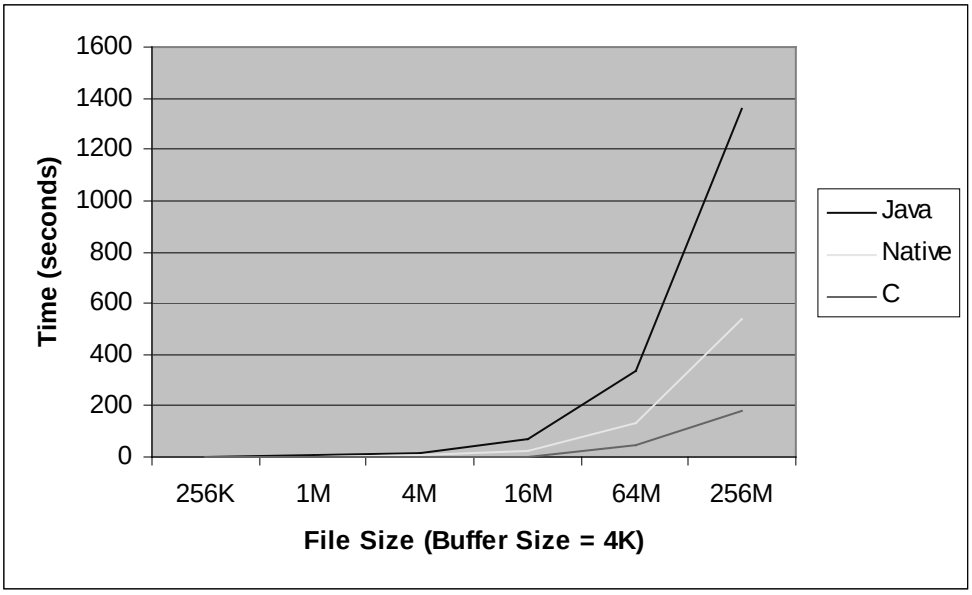

Figure 3 Sequential reads performed by a Java, C and non naïve native implementation.

3.3 Native method invocation and JVM

This is the last part of the Java I/O path we consider in this report. This part is implementation specific and would hold for any application regardless of whether it is I/O bound or not. We do not consider effects of different I/O interfaces provided by Java

standard library (java.io), as much work has already been done on evaluating this interface [2].

4 Results and Analysis

In this section we explain our results and optimizations. First, we identify the performance bottlenecks present in the Java I/O path explained earlier. Our methodology for identifying these performance bottlenecks is to compare Java and C implementation of the same benchmark. We assume that C implementation has the best performance we could hope to achieve. The result of this assumption is that we need not consider bottlenecks that have same performance overhead for both C and Java.

We only use the sequential read micro benchmark for studying bottlenecks. The advantage of using a simple micro benchmark is that we get to analyze each interface conclusively. Once we have identified the bottlenecks, we look at the different kinds of benchmarks and report our results in the light of developed understanding of different interfaces.

Our approach in understanding the performance implication of this system is bottom up. We start looking at performance bottlenecks in OS interface and traverse our path upwards to the JVM.

4.1 Analyzing the Java I/O path

To analyze the Java I/O path, we use a sequential read micro-benchmark. It reads a file sequentially from beginning to end. It implements a byte-oriented read interface in which the main loop reads one byte at a time. However, the `read()` system call reads the number of bytes specified by the buffersize, an argument to the benchmark. We use three implementations of this benchmark – BufferedRead, NativeRead and C_Read. BufferedRead and NativeRead are implemented in Java whereas C_Read is implemented in C. BufferedRead uses BufferedInputStream defined in the standard Java package java.io. BufferedInputStream provides a byte-oriented read interface but internally it uses a buffer to read a number of bytes from the file. NativeRead uses our own implementation of BufferedInputStream. Since we implement our own native read function, this implementation gives us a handle to analyze the performance bottlenecks in the native implementation. Finally C_Read implements the same functionality in C. Appendix I contains the complete code of these implementations.

Having explained our benchmark, we are now ready to explore the Java I/O path.

4.1.1 OS Interface

First we study a possible performance bottleneck of using the Unix read system call. To study the overhead we compare user time of two implementations – NativeRead as described before and DummyNativeRead. DummyNativeRead is same as NativeRead except the read system call is replaced by a dummy function call. Since read system call

is executed in kernel, we expected no improvement in user time when using DummyNativeRead. Our initial experiments were confined to use read system call for each byte (i.e. set the buffer size to 1). Contrary to our expectations, we saw a big improvement in user time while not using the read system call. For 1 GB file there was around 9.7 times improvement in user time. This raised a number of hypothesizes –

- Java uses green threads, which are threads handled by the JVM. The JVM might be scheduling these threads in non-optimal fashion.
- There might be a problem with the Unix time utility that was being used to measure kernel and system time.
- This difference can be due to the processor cache state. In the case when a system call is used in every loop, the user process incurs a context switch during each iteration and hence user process can never run in cached state. Both instruction and data caches can improve performance dramatically.

We tried to disprove each hypothesis one by one. The first hypothesis can be disproved if we use a JVM implementation that can use native threads or green threads and the results are similar under both threads. Unfortunately, we couldn't find any JVM implementation that has both native and green threads. Finally we decided to do the same experiment in C. We used C_Read and Dummy_C_Read (which replaces read system call by a dummy function call). Interestingly, we saw an improvement of over 13 times in C rejecting our first hypothesis.

To disprove the second hypothesis we used time utility of different Unix implementations. All of them agreed with the current observation rejecting our hypothesis.

The third hypothesis can be disproved if we let the user process run in cached state for a while before using read system call and the results remain the same. We did this by changing the buffer size while making the read system call. Our hope was that at some particular buffer size, we would get all the benefits of caching and the user times for both implementations (with or without read system call) would converge. Figure 4 shows the result of this experiment. It met our expectations to some extent. The user time of NativeRead did drop initially but once the buffer size reaches 64, there is virtually no improvement. There still seems to be some kind of overhead for the read system call. To contrast it with C, we repeated the same experiment with C implementations. Figure 5 shows the results for C implementations. As in the case of Java, User time drops significantly till buffer size increases to 16. Afterwards, increase in buffer size results in a small increase in user time. We hypothesize that this happens because large buffers are not laid out in cache conscious manner. In case of Java, caching contents of buffer is not as important as the state of JVM itself.

To differentiate the read system call overhead in C and Java, we plot the gap between user times for both C and Java implementations. Figure 6 shows the resulting graph.

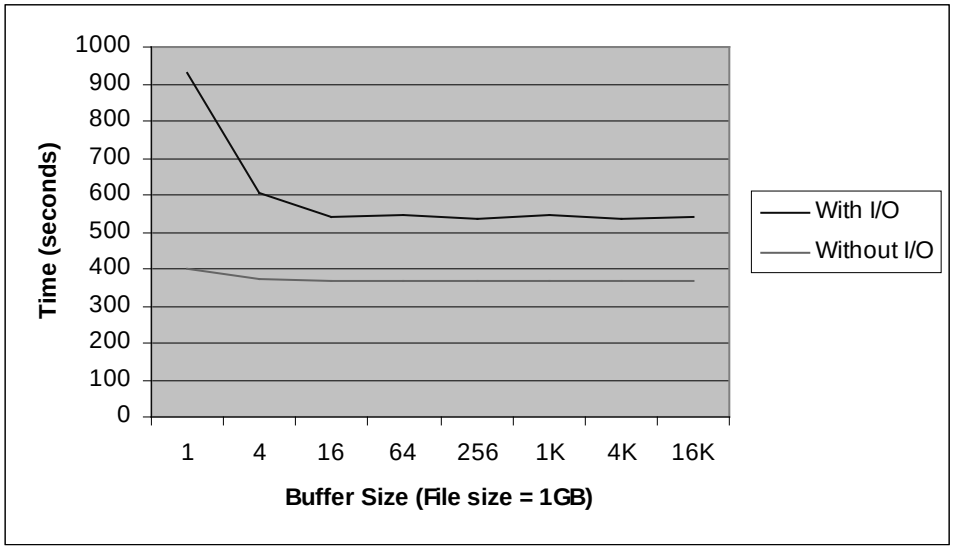

Figure 4: User time for the two Java implementations – NativeRead (with I/O) and DummyNativeRead (w/o I/O).

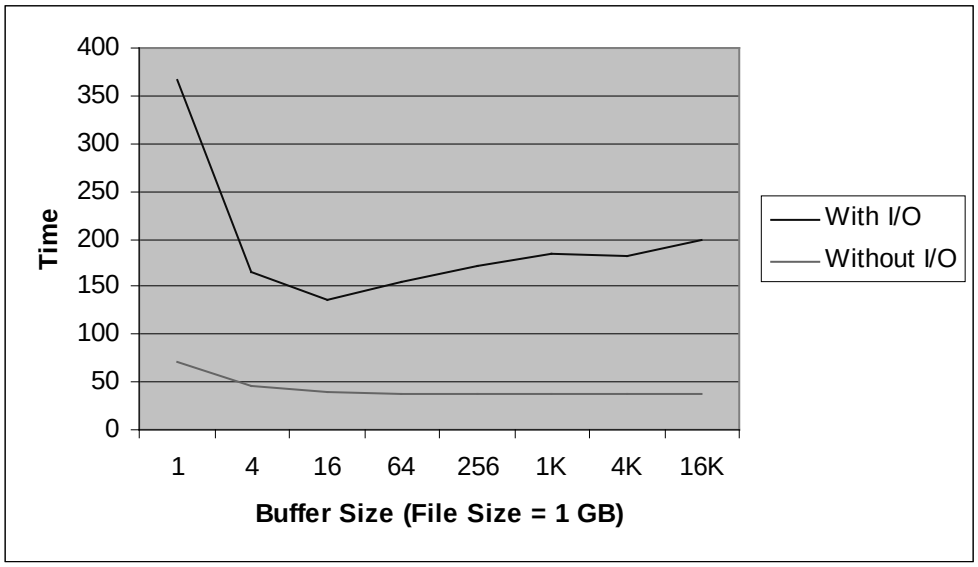

Figure 5: User time for the two C implementations – C_Read (with I/O) and C_Dummy_Read (w/o I/O).

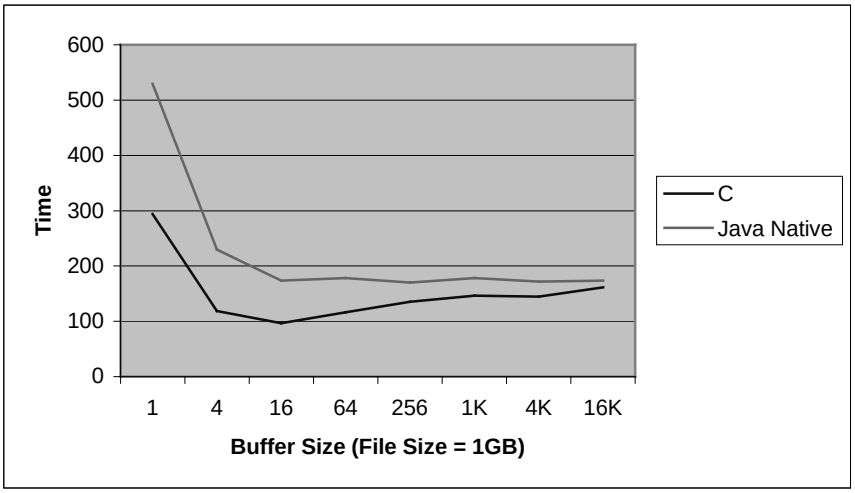

Figure 6: Comparing overhead of read system call in C and Java

It is clear from the figure that this overhead of read system call is same for C and Java. We hypothesize that this is because of the code generated by the compiler for a read system call. Since it is same for both C and Java, we do not explore this further.

4.1.2 Native Method Invocation and JVM

Our final experiment was to measure the overhead of native method invocation. We measure performance of two programs – EmptyNative and PureJava. Both have a main loop that calls a dummy function, which just increments a counter. EmptyNative uses native implementation of that function whereas PureJava uses a java implementation of the same method. Figure 7 shows the resulting bar graph. PureJava is slower than EmptyNative proving that the overhead of native method invocation is similar to a pure java method invocation.

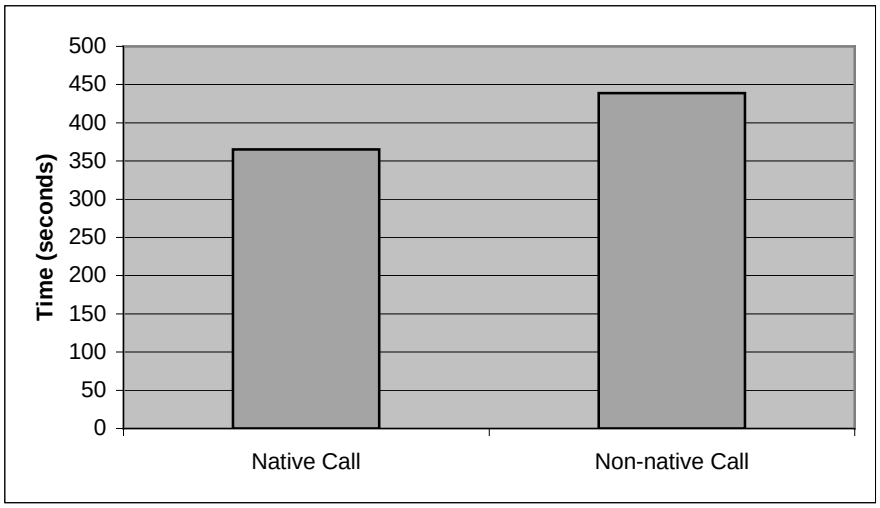

Figure 7: Native call overhead

4.2 Overall performance

The previous analysis indicated that in the Java I/O path, the bytecode interpretation is the main performance bottleneck. The lack of system calls in bytecode does not degrade the I/O performance. Before we conclude that Java I/O is significantly slower than C I/O, we examine a few more benchmarks. The problem with our previous read benchmark is that it assumes that applications always use I/O in byte-oriented fashion. Consequently, there is a significant bytecode overhead for each byte of I/O. In this section, we examine performance of the benchmarks that have low bytecode overhead per byte of I/O performed. These benchmarks represent the I/O bound applications more accurately. We also study the result of random I/O in which total time is dominated by disk seeks rather than CPU. Figure 8 and 9 show the results of these benchmarks.

The results of random I/O show the difference in user time for C and Java implementation. Total execution of a program had three components – user time, kernel time and disk I/O time. In the case of random I/O the disk seek would dominate the total running time making this difference in user time insignificant. We wanted to measure the effect of seeks on the real time, but once the file size increases beyond 16MB, the program does not finish in 32 hours.

BigRead is a micro-benchmark that is intended to track the performance of applications that use a buffered interface to read files. In this case instead of reading a byte at a time, the application reads a buffer at a time. As shown in the resulting graph, the overall time goes down as the buffer size is increased. Once the buffer size reaches 64, only the disk I/O determines the total running time.

5 Conclusion and Future Work

From this study we conclude that while I/O bound applications written in Java do have poorer performance than their C counterparts, that slowdown is for the same reasons that compute bound Java Applications run slower than their C counterparts. The applications where I/O is significantly more than CPU activity, Java does not perform very poorly. In addition to the application level techniques described in [2], implementing I/O functionality natively. Doing this, of course, gives up some of the platform independence that makes Java attractive. We believe that by implementing classes with native methods that present the same interface to the application that the standard Java API presents, it is reasonable to implement applications that can use the Java API classes when high-performance native versions are not available.

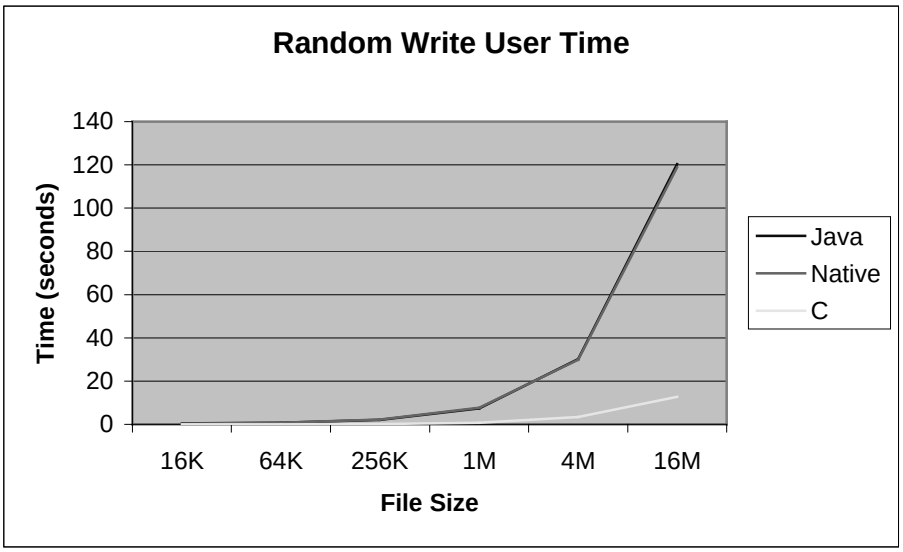

Figure 8: Comparing User time in C, Java and our implementation of native read method

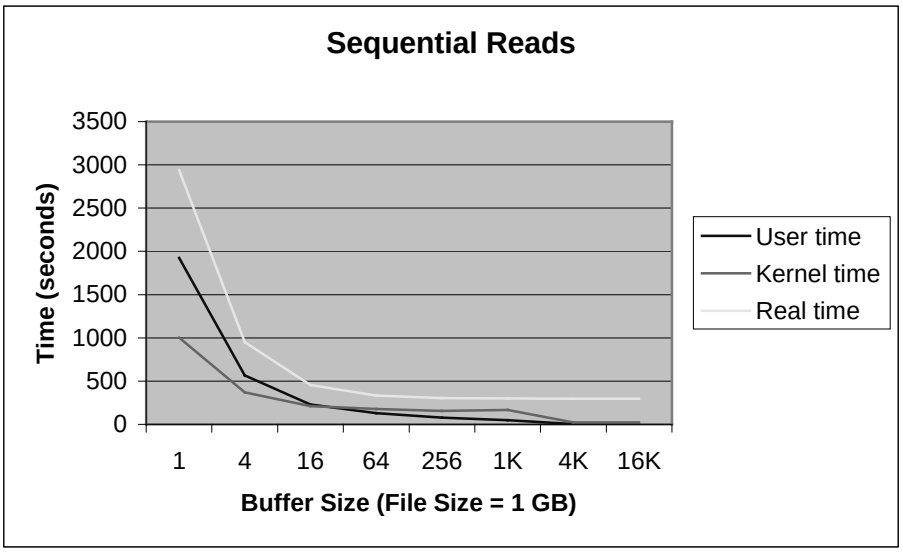

Figure 9: Measuring performance of BigRead

6 Acknowledgments

We wish to thank the Program Chair, Publicity Chair, Steering Committee, Chief of Staff, and Commander in Chief, Professor Remzi H. Arpaci-Dusseau. We also wish to thank the University of Wisconsin Computer Systems Lab for providing the equipment for this study. We would also like to appeal to the Commander in Chief that he not count the appendix toward the limit of 12 pages since one does not really have to read it in order to understand our work, rather it is provided to satisfy the curiosity of the reader.

7 Bibliography

[1] Bonachea, D., "Bulk File I/O Extensions to Java" *Java 2000,* San Francisco, CA

[2] Dickens, P. And Thakur, R., "An Evaluation of Java's I/O Capabilities for High-Performance Computing" *Java 2000,* San Francisco, CA

[3] JNI™ Java Native Interface http://java.sun.com/products/jdk/1.2/docs/guide/jni/

8 Appendix I

Here we list the code used in our sequential read micro-benchmarks.

8.1 BufferedRead.java

It uses the BufferedInputStream class from java.io package.

```
import java.io.*;

public class BufferedRead{
  public static void main(String args[]) {
   if (args.length != 2) {
     System.err.println(" <file-name> <buffer-size>");
     System.exit(1);
   }
   int buffer_size = (new Integer(args[1])).intValue();
   try {
     FileInputStream fis =
         new FileInputStream(args[0]);
     BufferedInputStream bis =
         new BufferedInputStream(fis, buffer_size);
     int cnt = 0;
     int b;
     while ((b = bis.read()) != -1) {
       if (b == '\n')
         cnt++;
```

```
        }
      bis.close();
      System.out.println(cnt);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

8.2 NativeRead.java

It is similar to BufferedRead.java except it uses our own implementation to BufferedInputStream.

The main function is implemented as before. Here we list our implementation of the read method of BufferedInputStream.

```
JNIEXPORT jint JNICALL
Java_NativeRead_read(JNIEnv *env, jobject obj){
    jclass cls;
    jfieldID fid;
    int fd;

    // Get the class of obj (this)
    cls = env->GetObjectClass(obj);

    // Get the field ID from this class which is of type
    // Int and is named "fd"
    fid = env->GetFieldID(global_cls, "fd", "I");

    // Get the value of this field
    fd =  env->GetIntField(obj, global_fid);

    // buf_length and buf_pos are statically defined
    // variables
    if(buf_length != 0){
        buf_length--;
        return buf[buf_pos++];
    }

    // Making the read system call
    // buf_size is a global varible set once in the open
    // method
    buf_length = read(global_fd, buf, buf_size);

    if(buf_length == 0) return -1;

    buf_pos = 0;
    buf_length--;
```

```
    return buf[buf_pos++];

}
```

8.3 C_Read.c

It implements the same functionality in C.

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>


static int buf_size;
static int buf_length = 0;
static int buf_pos;
static char * buf;

int my_read(int fp, char * c){

    if(buf_length != 0){
        buf_length--;
        *c = buf[buf_pos++];
        return 1;
    }

    buf_length = read(fp, buf, buf_size);

    if(buf_length == 0) return 0;

    buf_pos = 0;
    buf_length--;
    *c = buf[buf_pos++];
    return 1;

}

main(int argc, char ** argv){
    int fp;
    char c;
    int cnt,ret;
    int count;

    if(argc < 3){
        printf("<file-name> <buffer-size>\n");
        return -1;
    }

    buf_size = atoi(argv[2]);
    buf = (char *) malloc(buf_size*sizeof(char));

    count = 0;
```

```
    cnt = 0;
    fp = open(argv[1],0_RDONLY,0);

    // The Main Loop
    while(my_read(fp, &c)!=0){
        if(c == '\n') count++;
    }

    printf("%d\n",count);
    close(fp);
}
```