An In-Depth Examination of Java I/O Performance and Possible Tuning Strategies

Kai Xu xuk@cs.wisc.edu
Hongfei Guo guo@cs.wisc.edu

Abstract

There is a growing interest in using Java for high-performance computing because of the many advantages that Java offers as a programming language. To be useful as a language for high-performance computing however, Java must not only have good support for computation, but must also be able to provide high-performance file I/O. In this paper, we first examine possible strategies for doing Java I/O. Then we design and conduct a series of performance experiments accordingly using C/C++ as a comparison group. Based on the experimental results and analysis, we reach our conclusions: Java raw I/O is slower than C/C++, since system calls in Java are more expensive; buffering improves Java I/O performance, for it reduces system calls, yet there is no big gain for larger buffer size; direct buffering is better than the Java-provided buffered I/O classes, since the user can tailor it for his own needs; increasing the operation size helps I/O performance without overheads; I/O-related system calls implemented within Java native methods are cheap, while the overhead of calling Java native methods is rather high. When the number of JNI calls is reduced properly, a performance comparable to C/C++ can be achieved.

1. Introduction

There is a growing interest in using Java for high-performance computing because of the many advantages that Java offers as a programming language. To be useful as a language for high-performance computing however, Java must not only have good support for computation, but must also be able to provide high-performance file I/O, as many scientific applications have significant I/O requirements. However, while much work has been done in evaluating Java performance as a programming language, little has been done in a satisfying evaluation of Java I/O performance. In this paper, we investigate in depth the I/O capabilities of Java, and examine how and how well different possible tuning strategies work compared to C/C++.

1.1 Contribution of This Paper

The contributions of this paper are threefold. First we explored possible strategies one can utilize to get high performance in Java I/O. Secondly, we designed and conducted a series of experiments that examine the performance of each individual strategy accordingly, in comparison to C/C++. Finally, experiment results are thoroughly analyzed and conclusions are reached.

1.2 Related Work

There are already some papers discussing Java I/O performance. Our work is different from those in that we summarize possible I/O strategies in Java and give a thorough Java I/O performance evaluation and analysis in comparison to C/C++. [1] describes in detail possible strategies in improving Java I/O. However, no convincing experiments have been given to show how well those strategies work, neither has it studied Java I/O in comparison to that of C/C++. [2] compares Java I/O to that of C/C++ and proposes bulk I/O extensions. However, this paper mainly focuses on parallel Java I/O for specific applications instead of examining Java I/O in general.

1.3 Organization

The rest of this paper is organized as follows. In Section 2 we describe the basic I/O mechanisms defined in Java. In Section 3 we discuss our test methodology and experiments design. Then we give out the corresponding experiment results and analysis in Section 4. Conclusions and ideas for future work are presented in Section 5.

2. Java I/O Overview

To understand the issues associated with performing I/O in Java, it is necessary to briefly review the Java I/O model.

When discussing Java I/O, it is worth noting that the Java programming language assumes two distinct types of disk file organization. One is based on streams of bytes, the other on character sequences. Byte-oriented I/O includes bytes, integers, floats, doubles and so forth; text-oriented I/O includes characters and text. In the Java language a character is represented using two bytes, instead of the one byte representation in C/C++. Because of this, some translation is required to handle characters in file I/O. In this project, since our major concern is to compare Java I/O to that of C/C++, we will focus on the byte-oriented I/O.

In Java, byte-oriented I/O is handled by input streams and output streams, where a stream is an ordered sequence of bytes of unknown length. Java provides a rich set of classes and methods for operating on byte input and output streams. These classes are hierarchical, and at the base of this hierarchy are the abstract classes InputStream and OutputStream. It is useful to briefly discuss this class hierarchy in order to clarify the reason why we are interested in FileInputStream/FileOutputStream, BufferedInputStream/ BufferedOutputStream, and RandomAccessFile in our test cases. Figure 2.1 provides a graphical representation of this I/O hierarchy. Note that we have not included every class that deals with byte-oriented I/O but only those classes that are pertinent to our discussion.

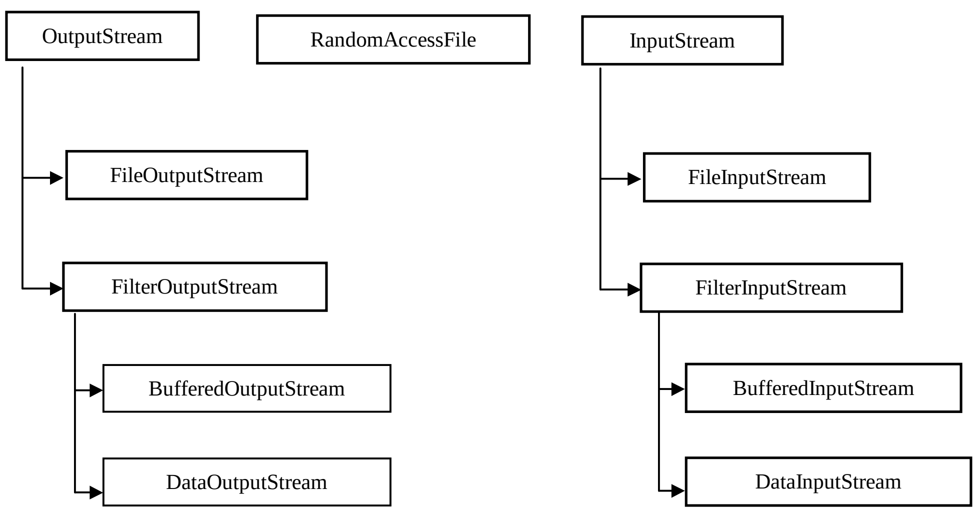

Figure 2.1 Pertinent Java I/O classes hierarchy

2.1 InputStream and OutputStream Classes

The abstract classes InputStream and OutputStream are the foundation for all input and output streams. They define methods for reading/writing raw bytes from/to streams. For example, the InputStream class provides methods for reading a single byte, a byte array, or reading the available data into a particular region of a byte array. The OutputStream class provides methods for writing that are analogous to those of InputStream.

2.2 File Input and Output Streams

The FileInputStream and FileOutputStream classes are concrete subclasses of InputStream and OutputStream respectively, which provide a mechanism to read from and write to files sequentially. Both classes provide all the methods of their superclasses. These two classes are the lowest file I/O classes provided to users.

2.3 Filter Streams

Filter streams provide methods to chain streams together to build composite streams. For example, a BufferedOutputStream can be chained to a FileOutputStream to reduce the number of calls to the file system. The FilterInputStream and FilterOutputStream classes also define a number of subclasses that manipulate the data of an underlying stream.

2.4 Buffered Input and Output Streams

Two important subclasses of filter streams are pertinent to this investigation – BufferedInputStream and BufferedOutputStream. These classes provide buffering for an underlying stream, where the stream to be buffered is passed as an argument to the constructor. The buffering is provided with an internal system buffer whose size can (optionally) be specified by the user.

2.5 Higher Level I/O Classes

All the classes discussed so far manipulate raw byte data only. Applications however may want to deal with higher-level data types, such as integers, floats, doubles, and so forth. Java defines two interfaces, DataInput and DataOutput, which define methods to treat raw byte streams as these higher-level Java data types. Together, these interfaces define methods for reading and writing all Java data types. The DataInputStream and DataOutputStream classes provide default implementations for these interfaces. These classes are outside the scope of this paper.

2.6 Random Access Files

All the classes mentioned above deal with sequential access I/O, RandomAccessFile is the only class provided by Java for random access I/O (at the byte level) on files. This class provides the seek method, similar to the one found in C/C++, to move the file pointer to an arbitrary location, from which point bytes can then be read or written. The RandomAccessFile class sits alone in the I/O hierarchy and duplicates methods from the stream I/O hierarchy. In particular, RandomAccessFile duplicates the read and write methods defined by the InputStream and OutputStream classes and implements the DataInput and DataOutput interfaces that are implemented by the data stream classes.

3. Test Methodology and Experiment Design

In this section we discuss our test methodology and experiment designs corresponding to different Java I/O strategies.

3.1 Methodology

In order to get a thorough examination of Java I/O performance in comparison to that of C/C++, we design a series of experiments corresponding to different Java I/O strategies. Throughout our experiments, we mainly examine two aspects of each strategy:

(1) How well it improves I/O in Java, and
(2) How good it is compared to its C/C++ counterpart.

Furthermore, we try to explain how and why each strategy impact Java I/O.

Note that for the sake of reliability and precision, we ran each test case five times, and took the average of the results.

3.1.1 Access Pattern and Benchmark

In our experiments, we implemented a collection of small benchmark programs using two typical file access patterns: sequential access and random access. For sequential access benchmarks, we first write the whole file sequentially, and then read all bytes back in the same order. Following is the psudo code of our benchmarks:

```
// Sequentially write a file
seqWrite() {
   open file;
   for (int I=0; I<FILESIZE/OPERATIONSIZE, I++)
   {
     WRITE(buf, OPERATIONSIZE, fileDesc);
     // Here different I/O strategies provide different write methods
   }
   close file;
}

// Sequentially read a file
seqRead() {
   open file;
   for (int I=0; I<FILESIZE/OPERATIONSIZE, I++)
   {
     READ(buf, OPERATIONSIZE, fileDesc);
     // Here different I/O strategies provide different read methods
   }
   close file;
}
```

For random access benchmarks, we randomly write as many bytes as the file size into a file, and randomly read back the same number of bytes from the file:

```
//Randomly write into a file
ranWrite() {
   open file;
   for (int I=0; I<FILESIZE/OPERATIONSIZE, I++)
   {
      position = rand() % (FILESIZE/OPERATIONSIZE);
       WRITE(buf, OPERATIONSIZE, position, fileDesc);
     // Here different I/O strategies provide different write methods
   }
   close file;
}

//Randomly read from a file
ranRead() {
   open file;
   for (int I=0; I<FILESIZE/OPERATIONSIZE, I++)
   {
      position = rand() % (FILESIZE/OPERATIONSIZE);
      READ(buf, OPERATIONSIZE, position, fileDesc);
      // Here different I/O strategies provide different read methods
   }
   close file;
}

Note: FILESIZE         – the file size;
      OPERATIONSIZE    - how many bytes are written/read in each operation
```

3.1.2 Comparison Group

Throughout our experiments, we use C/C++ as a comparison group, because it is the most commonly accepted programming language in the industry. We believe it is beneficial for people to have a good understanding of what I/O performance to expect compared to C/C++ when using different Java I/O strategies in their applications.

3.1.3 Data of Interest and Profiling Tools

There are two kinds of data we are interested in in our experiments. One is the elapsed time, which is the commonly used measure for I/O performance; the other is the CPU breakdown, which can help explain where the CPU time is spent.

We use time profiler to get the elapsed time for both Java and C, PerfAnal [7] profiler to get the CPU breakdown of Java, and gprof profiler to get the CPU breakdown of C/C++.

PerfAnal is a GUI-based profiler for analyzing the performance of Java applications. It can analyze the CPU usage of each called method *inclusive* or *exclusive* subroutine calls.

3.2 Java I/O Strategies and Test Cases

There are several ways to tune I/O in Java. As a means of starting the discussion, here are some general rules on how to speed up I/O:

- Avoid accessing the disk.
- Avoid accessing the underlying operating system.
- Avoid method calls.
- Avoid processing bytes and characters individually.

Bearing those rules in mind, we address different Java I/O strategies one by one.

3.2.1 The Lowest Level I/O

In UNIX/C(C++), the lowest level way to read/write a byte from/to a file is to use the *read()/write()* system calls. As we mentioned above, the equivalent in Java are the *read()/write()* methods of the FileInputStream/FileOutputStream classes.

In this test, we simply use *read()/write()* of FileInputStream/FileOutputStream to operate the file with one byte a time. For the random access, we use *read(), write()* and *seek()* provided by the RandomAccessFile class.

We call this strategy raw Java I/O, which serves as a baseline in our discussion.

3.2.2 Buffered Input / Output

For sequential file access, the Java API provides a buffered input/output stream, which keeps an internal buffer for read/write operations. Intuitively, buffering will improve I/O performance. In this test case, we compare the performance of a buffered input/output stream to that of raw I/O, and to direct buffering, which will be explained shortly.

Note that the internal buffer size of the buffered stream is adjustable. Here we use its default value, which is 1KB.

3.2.3 Direct Buffering

If buffering does improve I/O performance, then which buffer size should we use? Is it true that the larger, the better? In this test case, we address those questions with a user-specified application-level buffer which is maintained to keep most of the read and write operations within memory and reduce I/O system calls. We call this method direct buffering so as to differentiate it from the buffering provided by the Java API.

3.2.4 Operation Size

The impact of the operation size on Java I/O is examined in this experiment. Here again we will use the lowest level FileInputStream/FileOutputStream and RandomAccessFile for sequential access and random access respectively. The reason we do not use any buffering is that we want to separate the impact of the operation size from that of the others.

3.2.5 Using JNI

A system call implemented within Java native methods is cheap, but the overhead (for example, parameter transformation) of making a native call is high. In this comprehensive test case, we examine the possible performance gain if we implement buffering and file access system calls using the Java Native Interface. We try to reduce the JNI call overhead by increasing the operation size.

4. Test Results and Analysis

In this section we present the results and analysis of our experiments with the various strategies described above. We first describe the hardware and software configurations for our experiments.

4.1 Experimental Setup

We conducted our experiments on a personal computer with one Pentium III 667 MHz CPU, 128 MB of memory, and a 10 GB IDE disk (1216 cylinders, 255 heads and 512 bytes per sector).

The operating system we used is RedHat Linux 6.2. We choose the JDK 1.2.2 for Linux as our Java Virtual Machine. In order to analyze the CPU usage of the benchmarks we used some profiling tools to obtain function/method timing and other information pertinent to performance. PerfAnal is a GUI-based profiler for analyzing the performance of Java applications. It can analyze the CPU usage of each called method *inclusive* or *exclusive* subroutine calls. For the C++ benchmark, we used gprof and time profilers under Linux.

4.2 The Lowest Level Java I/O

The first experiment we conducted is to measure the raw Java I/O performance. Both sequential and random benchmarks are tested with different file sizes. In all test cases we didn't use any buffer and the read/write operation size is set to 1 byte in order to get the raw Java I/O performance compared to C/C++.

Figure 4.1 shows the result for sequential read/write tests, where file sizes from 1 MB to 100 MB are tested. As it is clear from the figure, for both Java and C, the I/O operation rates are stable regardless of the increasing file size. And for all test cases, the Java sequential access performance is as slow as two times that of C. In Figure 4.2 we show the CPU time breakdown result, which was generated by the profiling tools mentioned above, for the 100 MB file size test case. We observed that for both Java and C, the read and write system calls cost most of the CPU time (98% for C, 90% for Java). In addition, Java's I/O-related system calls are more expensive than the corresponding C calls (224% for read, 158% for write). That is why Java has a poor performance for these sequential access tests. We also noted that since most of the CPU time is spent on the I/O-related system calls, to improve the Java I/O performance, we should focus on reducing this part of CPU cost.

Figure 4.3 and 4.4 present the I/O operation rate and CPU time breakdown results for random read/write tests. From these figures we also found a slower operation rate for Java (~400%). And all I/O-related system calls (read, write and seek) are expensive for Java in random access.

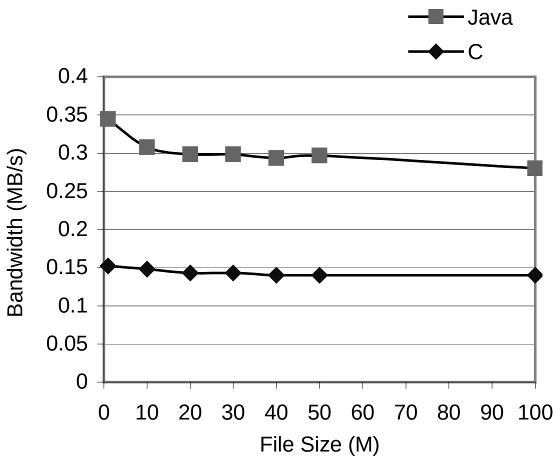

Figure 4.1: Sequential raw I/O performance with different file sizes

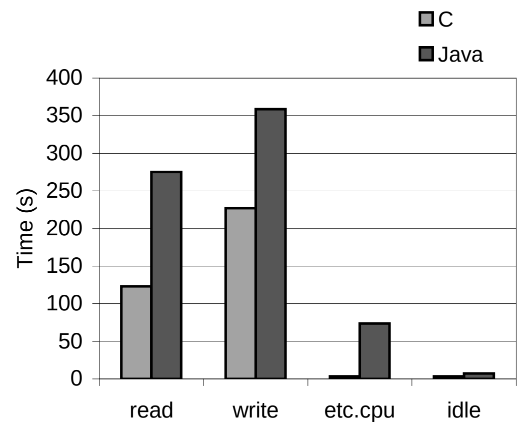

Figure 4.2: CPU breakdown for 100 MB sequential access

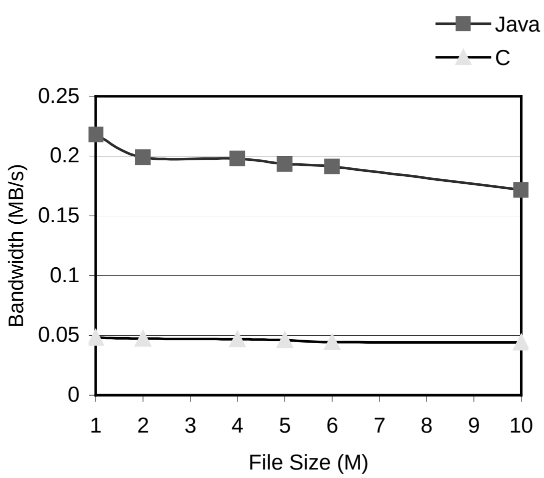

Figure 4.3: Random raw I/O performance with different file sizes

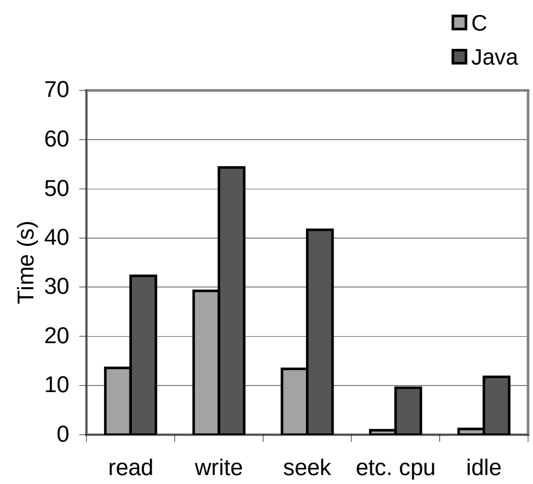

Figure 4.4: CPU breakdown result for 10 MB random access

4.3 Buffered I/O in Java

The above lowest level Java I/O performance experiment demonstrates that I/O-related system calls are expensive and account for a high percentage of CPU time. A basic rule to speed up I/O is to use buffers in order to reduce the accessing of underlying system calls. In this experiment we measured the performance of BufferedInputStream and BufferedOutputStream, which are the buffered I/O classes provided by the Java I/O package. Furthermore, we compared these general purpose buffered streams with direct buffering, where a user-specified application-level buffer is provided and maintained to keep most of the read and write operations within memory and reduce I/O system calls. Since the Java I/O package doesn't provide buffered streams for random access, in this experiment we only test the sequential benchmarks. The file size of the test benchmarks is set to 100MB. And the buffer size is specified to 1KB, which is the default buffer size of the BufferedInputStream and BufferedOutputStream classes.

Figure 4.5 plots the elapsed time for different buffering strategies in Java for our test cases. Three categories are shown in this figure, total elapsed time, sequential read elapsed time and sequential write elapsed time. For all these categories, we found that the Java package-provided buffered stream classes could improve Java I/O performance by 25% compared to no buffer for sequential access. Further, using our direct buffering could reduce the elasped time by an additional 40%. We show the CPU breakdown analysis in Figure 4.6. The costs of read and write system calls have been greatly reduced because of the buffering (they only occupy 0.36% and 0.5% of the CPU time respectively). However, in order to maintain the input/output buffers a large percentage of the CPU time has been spent on memory copy system calls for both buffered streams (28.83%) and direct buffering (43.64%). The difference in CPU usage for user-level buffer read/write method calls explains why direct buffering could show better I/O performance than general buffered streams provided by the Java package. Because direct buffering could maintain the buffer more efficiently according to the user's special request, the unnecessary buffer maintenance jobs are avoided.

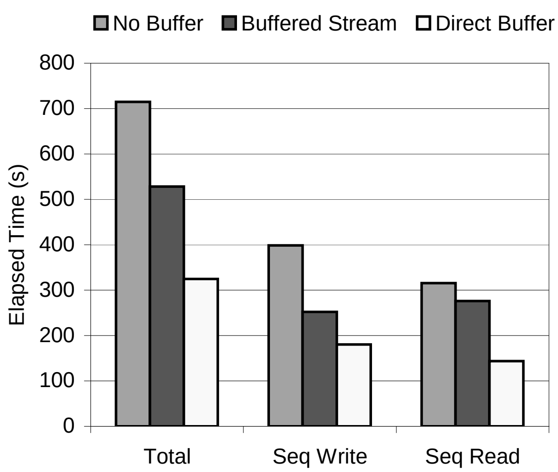

Figure 4.5: Elapsed time for different Java buffer strategies

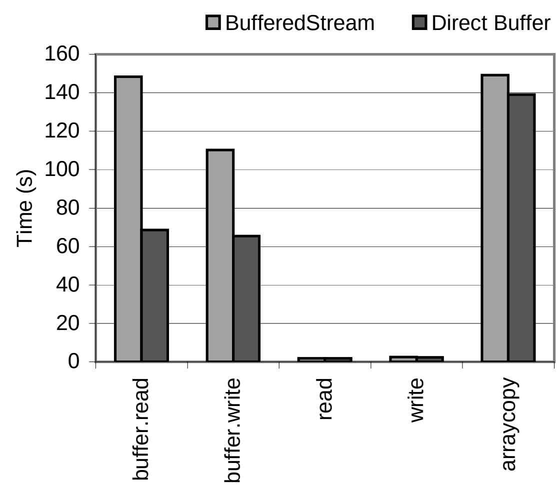

Figure 4.6: CPU breakdown for different Java buffer strategies

4.4 Direct Buffering

We have shown that the direct buffering technique can improve the Java I/O performance better than other buffer techniques in the previous experiment. In this experiment we studied this technique in-depth by employing it on both sequential and random accesses with different buffer sizes ranging from 1 byte to 4KB. In these test cases, we still keep 1 byte per read/write operation in order to get the pure effect of changing the buffer size.

4.4.1 Sequential Buffering

We depicted the sequential test result in Figure 4.7, where the file size is set to 100 MB and the elapsed time for each buffer size test case is plotted. There is no surprise that increasing the user-level buffer size reduces the elapsed time steadily. And we noticed that with the direct buffering technique, Java I/O performance is still 3 times slower than C performance. In Figure 4.7 we also provide the answer to an obvious question: will making the buffer larger speed up I/O? The experimental result shows that when the buffer size is larger than 512B or 1KB, increasing the buffer size may help speed up I/O, but only by a few percent, say 1% to 5%. We show the reason for that in the CPU breakdown analysis in Figure 4.8 and 4.9.

Figure 4.8 presents the CPU breakdown for Java sequential access tests. For each buffer size case, we break the elapsed time into several parts -- write and read CPU time, memory copy CPU time for maintaining the buffer, other CPU time and CPU idle time. Close examination of this figure reveals that when the buffer size is increased, the CPU time on I/O-related system calls is consistently reduced. At the same time, the other parts of CPU time do not change much. This gave us a clear picture that the buffering reduces the I/O system calls, and thus improves the I/O performance. But, even though we may increase the buffer size to a large scale to reduce the I/O-related CPU time to its limit, we cannot reduce the other parts of CPU time with buffering. According to Amdahl's law, if parts of a program run at two different speeds, the slower speed will dominate. Here this unchanged part of CPU time decides that using a larger buffer size can only improve Java I/O performance with a small percentage.

Another interesting point in Figure 4.8 is that for the two large buffer size test cases (1MB and 10MB), the elapsed time, especially the CPU idle time, increased a little. Our explanation for this is that for these test cases, the user-level buffer is already larger than the underlying system I/O buffer size. Under such circumstances, the read-ahead and write-behind buffer maintained by the underlying runtime system can not satisfy the large scale user-level read and write operations. The extended I/O waiting time increased the CPU idle time and total elapsed time of the benchmarks.

The CPU breakdown for C sequential tests is shown in Figure 4.9, where we got similar results as for the Java test cases.

4.4.2 Random Buffering

After examining the direct buffering for sequential access, in this section we study how the direct buffering will affect the random access. All the test cases use a file size of 10 MB.

The elapsed time for different buffer sizes is shown in Figure 4.10. According to this figure, for random access, when buffer size is increased the Java and C I/O performances degrade. The CPU breakdown results in Figure 4.11 (for Java) and Figure 4.12 (for C) give us a clear explanation. With the growing of the buffer size, the CPU time of I/O-related system calls (read, write, seek) increased. This is caused by the low read/write hit ratio to the buffer because of the arbitrary moving of the file pointer. In our random access test cases, the buffer hit ratio is lower than 1%. Under such circumstances, read/write system calls cannot be reduced and more data are read in/written out of the buffer with the growth of buffer size to maintain the buffer.

A closer examination of Figure 4.11 and 4.12 shows that when the buffer size increases, the CPU time of read system calls increases, while the CPU time of write system calls stays stable. The reason is again related to the underlying system I/O buffers. Because of the random access pattern, the hit ratio of the system I/O buffers is also very low. When a page-fault happens, for the read system call, the system will spend more time waiting for the input buffer to be filled. While for the write system call, after copying the data from the user-level buffer to the system output buffer, the system call can return without waiting for the real disk write operation to be finished.

In this experiment we also tested the extreme case of buffering the whole file. That means a 10 MB buffer is used, which is equal to the whole file size. The rightmost column on Figure 4.11 and 4.12 presents the result for this test. The performance improvement is obvious, since all the I/O operations are performed within the user-level buffers.

4.5 Operation Size

In previous experiments we have tested the effects of different file and buffer sizes. In this experiment we tested another I/O parameter -- the operation size. Both sequential and random benchmarks are tested with different operation sizes ranging from 1 byte to 4KB. Again, in order to obtain the pure effect of changing the operation size, no buffer is used in all test cases.

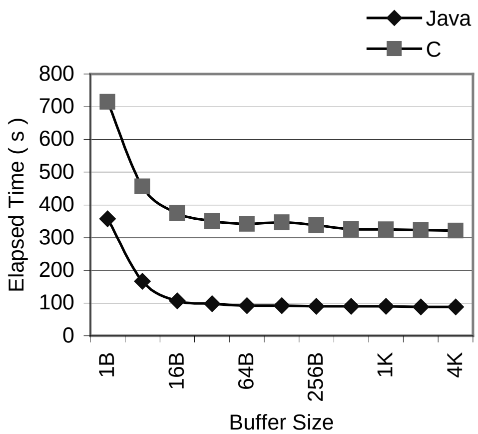

Figure 4.7: Sequential access for direct buffering

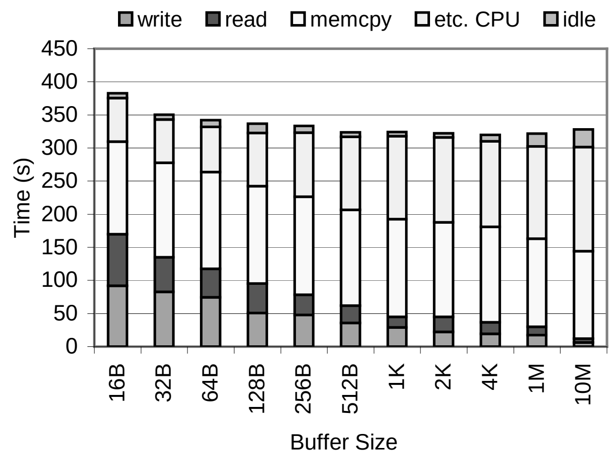

Figure 4.8: CPU breakdown for Java sequential access

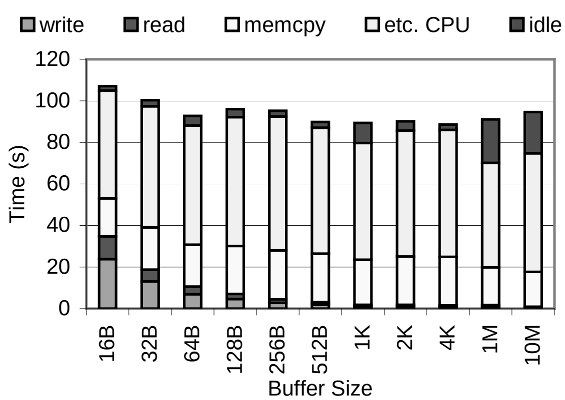

Figure 4.9: CPU breakdown for C sequential access

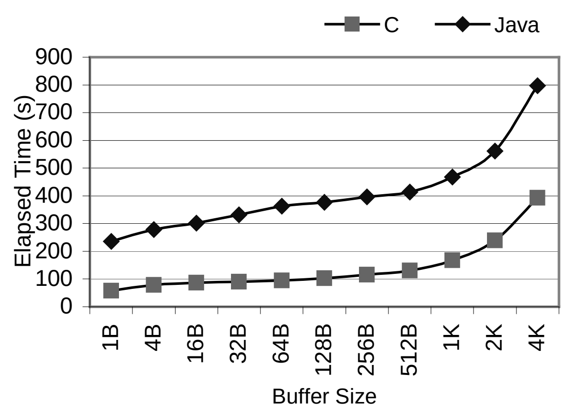

Figure 4.10: Random access for direct buffering

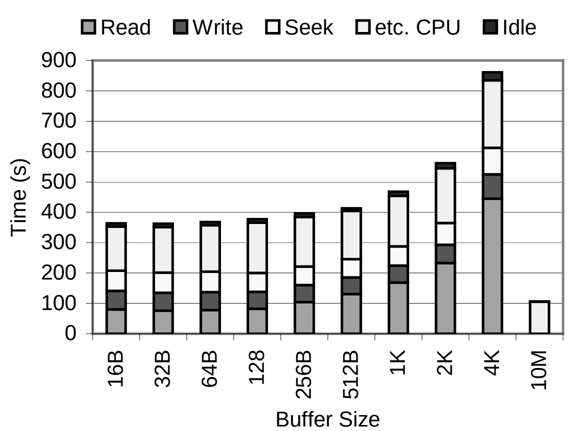

Figure 4.11: CPU breakdown for Java random access

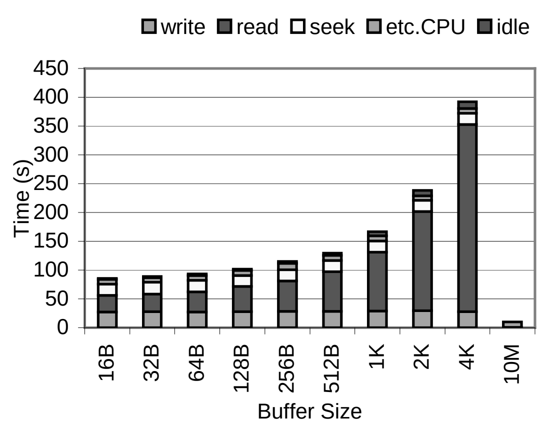

Figure 4.12: CPU breakdown for C random access

The sequential access result for a 100 MB file is shown in Figure 4.13. As we expected, the I/O performance improves as the operation size is increased, because the I/O-related system calls have been reduced. For the large operation test cases for Java, we observed an 85% performance improvement compared to 1 byte per read/write. Also, we noted that the Java I/O performance could be comparable to C performance when a large operation size is used, since increasing the operation size imposes no extra overheads.

Like the direct buffering experiments, when the operation size is larger than 512 bytes or 1 KB, there is no big gain of performance to increase the operation size further. The reason is that enlarging the operation size still cannot affect the non-I/O parts of CPU utilization.

Figure 4.14 presents the result for random access with a file size of 10 MB. There is no surprise that increasing the operation size in random access has a similar effect as the sequential access tests.

4.6 Using JNI

The Java Native Interface (JNI) is a native programming interface [9]. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. One of the cases for using Java native methods is to fulfill certain highly time-critical operations. In this experiment we implemented two native methods in C to deal with the I/O-related functionality including maintaining the user-level direct buffers. Performance results for the JNI implementation are compared with the Java version direct buffering and with C performance. The experiment setting is a 100MB file size, 4KB fixed buffer size, 1B to 1KB operation size, and only sequential access is tested.

Figure 4.15 presents the elapsed time result as a function of the operation size for three different implementations. As the previous experiment suggested, increasing the operation size improved the performances of all three implementations. Furthermore, we found that when the operation size is small (less than 4B), the JNI performance is comparable to Java direct buffering; while when the operation size becomes large (bigger than 64B), the JNI performance can reach the C level. Our explanation for this observation is that a native method itself is cheap because of its implementation in C; however, calling the native methods from JVM is expensive since there must be some overhead imposed. Further explanation of our test case is shown in Figure 4.16, which plots the CPU breakdown for the JNI implementation.

Since the Java profiler, PerfAnal, cannot trace into the native methods, on Figure 4.16 we only gave the CPU utilization of our two I/O native methods, jniread and jniwrite, and all the other CPU times. As we expected above, for the small operation size, these native methods cost a large amount of CPU time, since they are called frequently and the overhead for each call is expensive. When the operation size increases, less native methods are called, and the CPU time goes down. Although we cannot present the CPU time of the real I/O system calls in this figure, from Figure 4.15 we can deduce that the cost of I/O system calls in the JNI implementation is the same as in a C implementation. In the large operation size cases, when the calling overhead of native methods is reduced to a small value, the JNI implementation has similar performance to a C implementation.

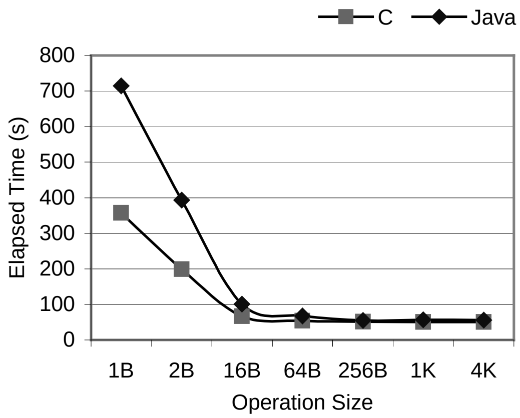

Figure 4.13: Sequential access for different operation sizes

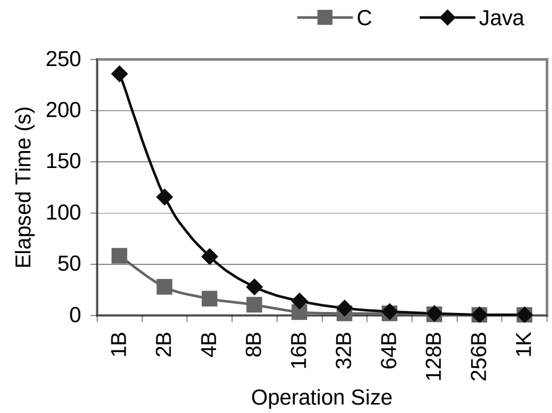

Figure 4.14: Random access for different operation sizes

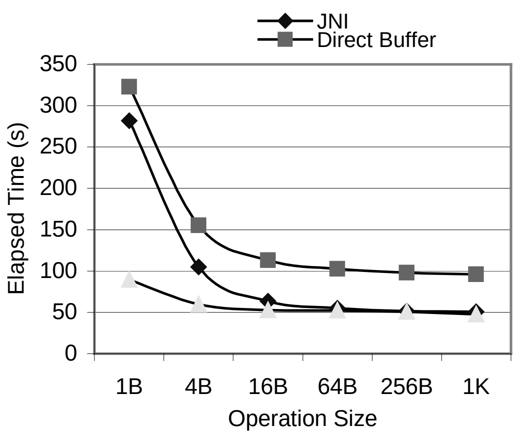

Figure 4.15: Elapsed time of three implementations

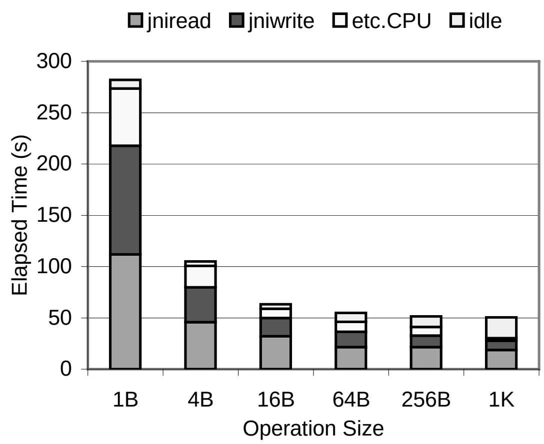

Figure 4.16: CPU breakdown for JNI

5. Conclusions and Future Work

In this paper, we investigate in depth the I/O capabilities of Java, and examine how and how well different possible tuning strategies work when compared to C/C++. A series of experiments was designed and conducted to examine the lowest level Java I/O performance and different strategies to enhance the performance. Based on the experiment results and analysis, we reached our conclusions: Java raw I/O is slower than C/C++, since system calls in Java are more expensive; buffering improves Java I/O performance, for it reduces system calls, yet there is no big gain for larger buffer size; direct buffering is better than the Java-provided buffered I/O classes, since the user can tailor it for his own needs; increasing the operation size helps I/O performance without overheads; and system calls are cheap in Java native methods, while the overhead of calling native methods is rather high. When the number of native calls is reduced properly, a performance comparable to C/C++ can be achieved.

In this project, we mainly focus on the low level Java I/O – byte-oriented I/O. As mentioned in section 2, Java needs to handle text-oriented I/O in a special way, for which a rich set of classes is provided such as FileReader/FileWriter and BufferedReader/BufferedWriter etc. It would be useful to further examine Java I/O performance and special tuning approaches within such a context.

References:

[1] Glen McCluskey. Tuning Java I/O Performance. March 1999. http://www.java.sun.com
[2] Dan Bonachea, Phillip Dickens, Rajeev Thakur. High-Performance File I/O in Java: Existing Approaches and Bulk I/O Extensions. August 2000. Submitted to Concurrency-Practice and Experience.
[3] Peter M Chen, David A. Patterson. A New Approach to I/O Performance Evaluation – Self-scaling I/O Benchmarks, Predicted I/O Performance. ACM 0734-2071/94/1100-0308, 1994.
[4] Rafael H. Saavedra, Alan J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. ACM 0734-207/96/1100-0344, 1996.
[5] Mendel Rosenblum, John K. Ousterhout. The Design and Implementation of a Log-Structured File System. Proceedings of the 13th ACM Symposium on Operating Systems Principles, 1992.
[6] Sameer Shende. Profiling and Tracing in Linux, 2000.
[7] Nathan Meyers. PerfAnal: A Performance Analysis Tool. 2000. http://www.java.sun.com
[8] Andy Vaught. Gprof, Bprof and Time Profilers.
[9] Sun Microsystems, Inc. Java Native Interface Specification. March 1997. http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html
[10] Nick Zhang. How to Improve Java's I/O Performance. October 2000. http://www.javaworld.com/javatips
[11] Gosling, J. and G. Steele. The Java Language Specification, Addison-Wesley, June 1996.
[12] Liang, S. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, 1999.