Nadeau software consulting

Java tip: How to read files quickly

February 24, 2008
Topics: Java
Technologies: Java 5+

Java has several classes for reading files, with and without buffering, random access, thread safety, and memory mapping. Some of these are much faster than the others. **This article benchmarks 13 ways to read bytes from a file and shows which ways are the fastest.**

Table of Contents

A quick review of file reading classes

Let's quickly run through several ways to open and read a file of bytes in Java. To give us a way to compare them, let's compute a file checksum.

FileInputStream with byte reads

`FileInputStream` opens a file by name or **`File`** object. It's **`read()`** method reads byte after byte from the file.

```
FileInputStream f = new FileInputStream( name );
int b;
long checkSum = 0L;
```

FileInputStream uses synchronization to make it thread-safe.

```
while ( (b=f.read()) != -1 )
    checkSum += b;
```

FileInputStream with byte array reads

FileInputStream does an I/O operation on every read and it synchronizes on all method calls to make it thread-safe. To reduce this overhead, read multiple bytes at once into a buffer array of bytes

```
FileInputStream f = new FileInputStream( name );
byte[] barray = new byte[SIZE];
long checkSum = 0L;
int nRead;
while ( (nRead=f.read( barray, 0, SIZE )) != -1 )
    for ( int i=0; i<nRead; i++ )
        checkSum += barray[i];
```

BufferedInputStream with byte reads

BufferedInputStream handles **FileInputStream** buffering for you. It wraps an input stream, creates a large internal byte array (usually 8 kilobytes), and fills it in large reads from the stream. It's **read()** method gets the next byte from the buffer.

```
BufferedInputStream f = new BufferedInputStream(
    new FileInputStream( name ) );
int b;
long checkSum = 0L;
while ( (b=f.read()) != -1 )
    checkSum += b;
```

BufferedInputStream uses synchronization to be thread-safe.

BufferedInputStream with byte array reads

BufferedInputStream synchronizes on all method calls to make it thread-safe. To reduce synchronization and method call overhead, make fewer **read()** calls by reading multiple bytes at a time.

```
BufferedInputStream f = new BufferedInputStream(
    new FileInputStream( name ) );
byte[] barray = new byte[SIZE];
long checkSum = 0L;
int nRead;
while ( (nRead=f.read( barray, 0, SIZE )) != -1 )
    for ( int i=0; i<nRead; i++ )
        checkSum += barray[i];
```

RandomAccessFile with byte reads

RandomAccessFile opens a file by name or **File** object. It can read, write, or read and write at your choice of position within the file. Its **read()** method reads the next byte from the current file position.

```
RandomAccessFile f = new RandomAccessFile( name );
int b;
long checkSum = 0L;
while ( (b=f.read()) != -1 )
    checkSum += b;
```

RandomAccessFile is thread-safe.

RandomAccessFile with byte array reads

Like a **FileInputStream**, **RandomAccessFile** issues an I/O operation on every read and synchronizes on all method calls to make it thread-safe. To reduce this overhead, make fewer method calls by reading into an array of bytes.

```
RandomAccessFile f = new RandomAccessFile( name );
byte[] barray = new byte[SIZE];
long checkSum = 0L;
int nRead;
while ( (nRead=f.read( barray, 0, SIZE )) != -1 )
    for ( int i=0; i<nRead; i++ )
        checkSum += barray[i];
```

FileChannel with ByteBuffer and byte gets

FileInputStream and **RandomAccessFile** both can return a **FileChannel** for low-level I/O. **FileChannel**'s **read()** method fills a **ByteBuffer** created using the **allocate()** method on **ByteBuffer**. **ByteBuffer**'s **get()** method gets the next byte from the buffer.

FileChannel and **ByteBuffer** are *not* thread-safe.

```
FileInputStream f = new FileInputStream( name );
FileChannel ch = f.getChannel( );
ByteBuffer bb = ByteBuffer.allocate( SIZE );
long checkSum = 0L;
int nRead;
while ( (nRead=ch.read( bb )) != -1 )
{
    if ( nRead == 0 )
        continue;
    bb.position( 0 );
    bb.limit( nRead );
    while ( bb.hasRemaining() )
        checkSum += bb.get( );
    bb.clear( );
}
```

FileChannel with ByteBuffer and byte array gets

To reduce method call overhead, get an array of bytes at a time. The array and **ByteBuffer** can be different sizes.

```
FileInputStream f = new FileInputStream( name );
FileChannel ch = f.getChannel( );
ByteBuffer bb = ByteBuffer.allocate( BIGSIZE );
byte[] barray = new byte[SIZE];
long checkSum = 0L;
int nRead, nGet;
while ( (nRead=ch.read( bb )) != -1 )
{
    if ( nRead == 0 )
        continue;
    bb.position( 0 );
    bb.limit( nRead );
    while( bb.hasRemaining( ) )
    {
        nGet = Math.min( bb.remaining( ), SIZE );
        bb.get( barray, 0, nGet );
        for ( int i=0; i<nGet; i++ )
            checkSum += barray[i];
    }
    bb.clear( );
}
```

FileChannel with array ByteBuffer and byte array access

A **ByteBuffer** created with the **allocate()** method above uses hidden internal storage. Instead, call **wrap()** to wrap a **ByteBuffer** around your own byte array. This lets you access your array directly after each read, reducing method call overhead and data copying.

```
FileInputStream f = new FileInputStream( name );
FileChannel ch = f.getChannel( );
byte[] barray = new byte[SIZE];
ByteBuffer bb = ByteBuffer.wrap( barray );
long checkSum = 0L;
int nRead;
while ( (nRead=ch.read( bb )) != -1 )
{
    for ( int i=0; i<nRead; i++ )
        checkSum += barray[i];
    bb.clear( );
}
```

FileChannel with direct ByteBuffer and byte gets

A **ByteBuffer** created with the **allocateDirect()** method may directly use storage deeper in the JVM or OS. This can reduce copying of data upward into your application's array, saving some overhead.

```
FileInputStream f = new FileInputStream( name );
FileChannel ch = f.getChannel( );
ByteBuffer bb = ByteBuffer.allocateDirect( SIZE );
long checkSum = 0L;
int nRead;
while ( (nRead=ch.read( bb )) != -1 )
{
    bb.position( 0 );
    bb.limit( nRead );
    while ( bb.hasRemaining() )
        checkSum += bb.get( );
    bb.clear( );
}
```

FileChannel with direct ByteBuffer and byte array gets

And of course, you can get arrays of bytes to save on method call overhead. The buffer and array sizes can differ.

```
FileInputStream f = new FileInputStream( name );
FileChannel ch = f.getChannel( );
ByteBuffer bb = ByteBuffer.allocateDirect( BIGSIZE );
byte[] barray = new byte[SIZE];
long checkSum = 0L;
int nRead, nGet;
while ( (nRead=ch.read( bb )) != -1 )
{
    if ( nRead == 0 )
        continue;
    bb.position( 0 );
    bb.limit( nRead );
    while( bb.hasRemaining( ) )
    {
        nGet = Math.min( bb.remaining( ), SIZE );
        bb.get( barray, 0, nGet );
        for ( int i=0; i<nGet; i++ )
            checkSum += barray[i];
    }
    bb.clear( );
}
```

FileChannel with MappedByteBuffer and byte gets

`FileChannel`'s map method returns a **`MappedByteBuffer`** that memory maps part or all of the file into the address space of the application. This gives more direct access to the file without an intermediate buffer. Call the **`get()`** method on **`MappedByteBuffer`** to get the next byte.

```
FileInputStream f = new FileInputStream( name );
FileChannel ch = f.getChannel( );
MappedByteBuffer mb = ch.map( ch.MapMode.READ_ONLY,
    0L, ch.size( ) );
long checkSum = 0L;
while ( mb.hasRemaining( ) )
    checkSum += mb.get( );
```

FileChannel with MappedByteBuffer and byte array gets

And getting arrays of bytes saves method call overhead.

```
FileInputStream f = new FileInputStream( name );
FileChannel ch = f.getChannel( );
MappedByteBuffer mb = ch.map( ch.MapMode.READ_ONLY,
    0L, ch.size( ) );
byte[] barray = new byte[SIZE];
long checkSum = 0L;
int nGet;
while( mb.hasRemaining( ) )
{
    nGet = Math.min( mb.remaining( ), SIZE );
    mb.get( barray, 0, nGet );
    for ( int i=0; i<nGet; i++ )
        checkSum += barray[i];
}
```

FileReader and BufferedReader

What about **`FileReader`** and **`BufferedReader`**? Both of these classes read *characters*, not bytes, so they are not included in these comparisons.

To handle characters, **`FileReader`** reads bytes from an internal **`FileInputStream`** and decodes them as UTF-8 multi-byte characters. **`BufferedReader`** does the same thing by wrapping a **`Reader`** and adding an internal buffer. Character decoding takes time, giving worse performance than simpler byte-reading classes.

Benchmarks

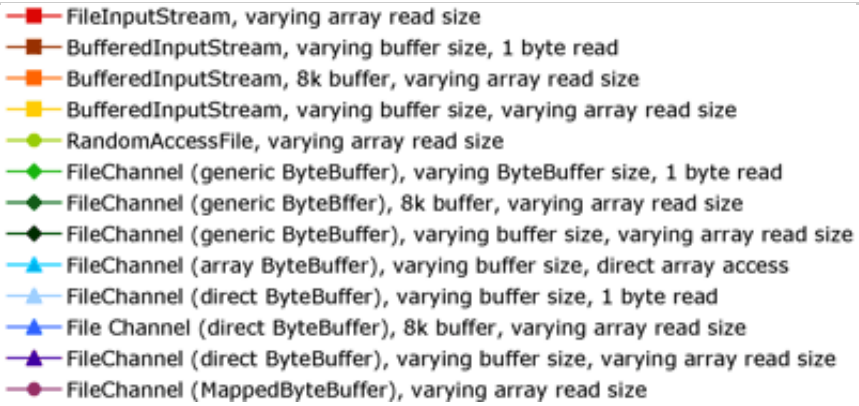
All of these approaches were benchmarked reading a 100 Mbyte file from a local disk. To make sure benchmarks included the cost of issuing I/O calls and their OS overhead, I measured wall clock time for the task, not CPU or user time. The results plotted below are from an unloaded Mac with JVM 5, but similar results were found on a Windows PC and JVM 6. The JVM was run with the following options:

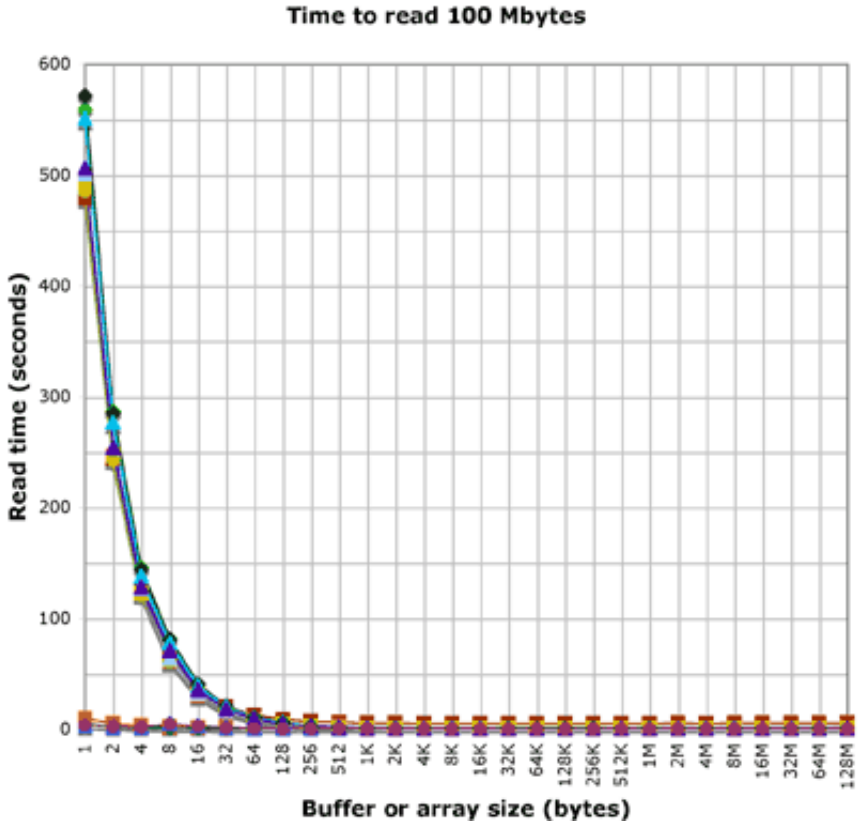
```
java -server -XX:CompileThreshold=2 -XX:+AggressiveOpts -
XX:+UseFastAccessorMethods -Xmx1000m Test
```

The **-server** option uses larger memory defaults and a parallel garbage collector to improve performance. The **-XX:CompileThreshold=2** option forced the JVM to compile methods after just two calls, insuring that compilation occurred early, and not in the middle of the benchmarks. The **-XX:+AggressiveOpts** option enables fancier optimizations, and **-XX:+UseFastAccessorMethods** inlines some **get()** methods. The **-Xmx1000m** option set a 1Gbyte memory limit so that the largest tests could run easily. Each of these options were tested independently and they all made a difference.

There are three variables to test: the read approach, the internal buffer size (for **BufferedInputStream** and **ByteBuffer**), and the read array size. Buffer and array sizes were varied separately and together from 1 byte to 128 Mbytes in size.

Full plot



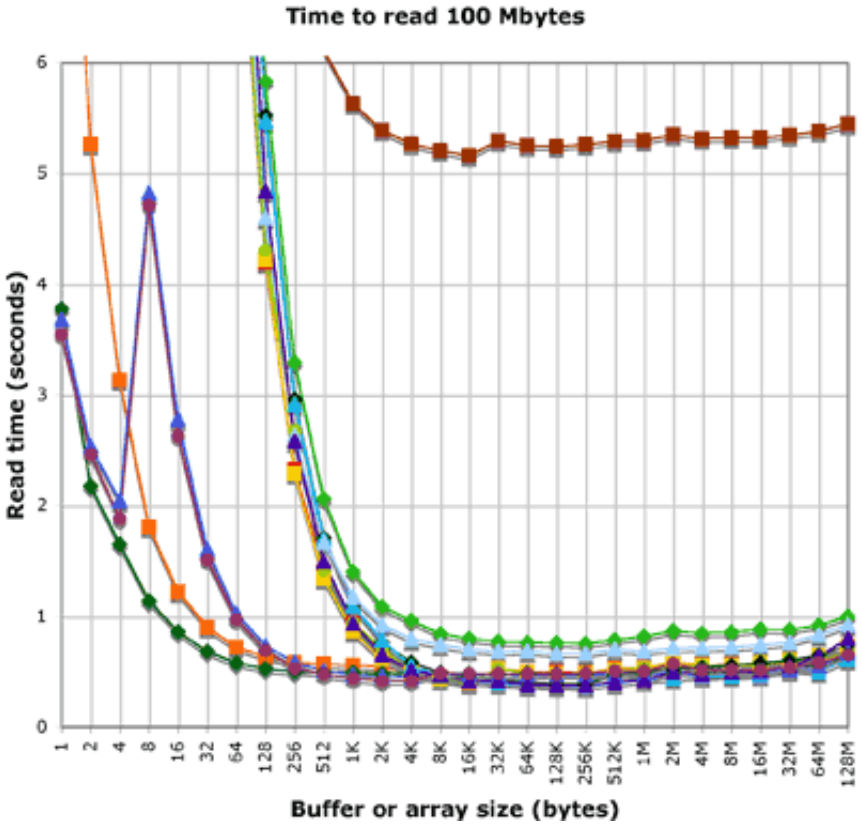


The cliff on the left is from *very slow* byte-by-byte reads. The peak is near 600 seconds, or 10 minutes! Over a 100 Mbyte file, that's a dismal 10 Mbytes *per minute*. For comparison, hard disks themselves can transfer data at better than 40 Mbytes per second.

This is what we expect to see. Issuing an I/O request is *very* expensive. To avoid this expense, make fewer I/O requests by reading an array of bytes at a time. Even a small array of just 64 bytes dramatically improves performance.

To see more, we need to zoom in to the bottom few seconds of the plot.

Zoomed plot



The curves drops down and level out as we increase the array sizes. By about 8Kbytes the curves flatten. Increasing array sizes further doesn't gain much. This is why **`BufferedInputStream`**'s default buffer size is 8Kbytes.

Performance levels out into about three groups:

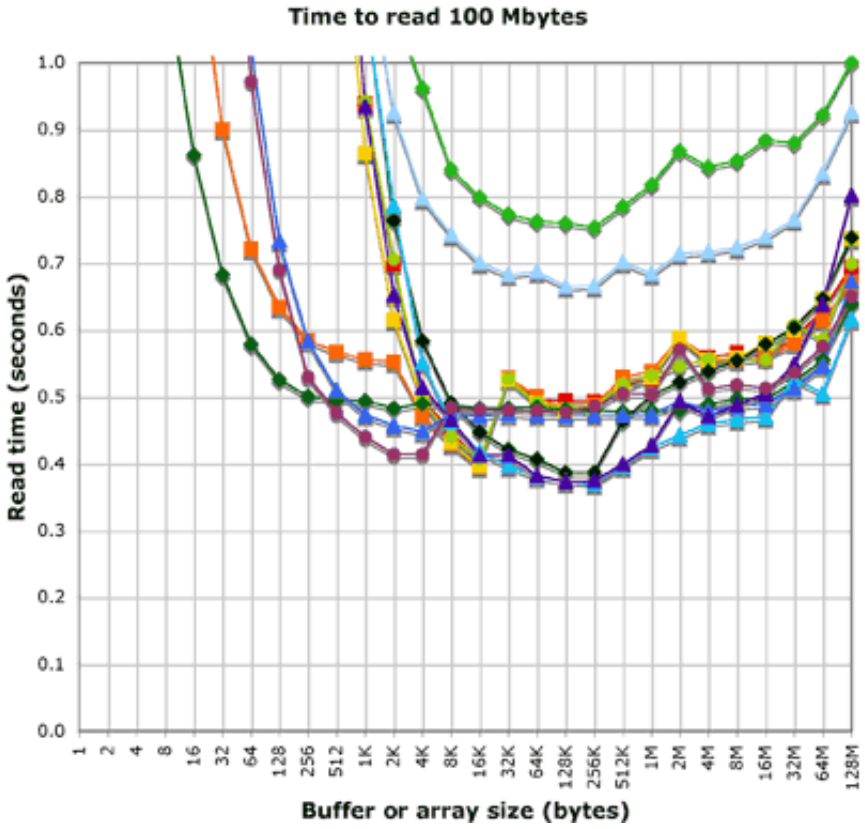
- At 5 seconds is **`BufferedInputStream`** and getting one byte at a time from an internal buffer. Each byte requires a method call, and each method call requires a thread synchronization lock, hurting read performance.
- At just under 1 second is getting one byte at a time from a **`FileChannel`** and a large **`ByteBuffer`**. **`FileChannel`** doesn't use synchronization locks, but a method call per byte still hurts performance.
- At about 1/2 second are all of the rest. All of these read arrays of bytes to minimize I/O operations, method calls, and thread synchronization, giving them better performance.

A few approaches stand out at the left side of the plot with curves that drop faster than the others:

- **`BufferedInputStream`** with an internal 8Kbyte buffer and **`FileChannel`** with an internal 8Kbyte **`ByteBuffer`** do better simply because there is that 8Kbyte buffer behind the reads. The left-shifted curves are really a plotting artifact. They should, perhaps, be plotted based upon the internal read buffer size, not the application's read array size.
- **`FileChannel`** with a **`MappedByteBuffer`** does well because there is effectively an internal buffer involved in memory mapping. That buffer is a multiple of the OS page size, which is typically 4Kbytes.

To be sure we've seen everything, let's zoom in even more.

Really zoomed plot



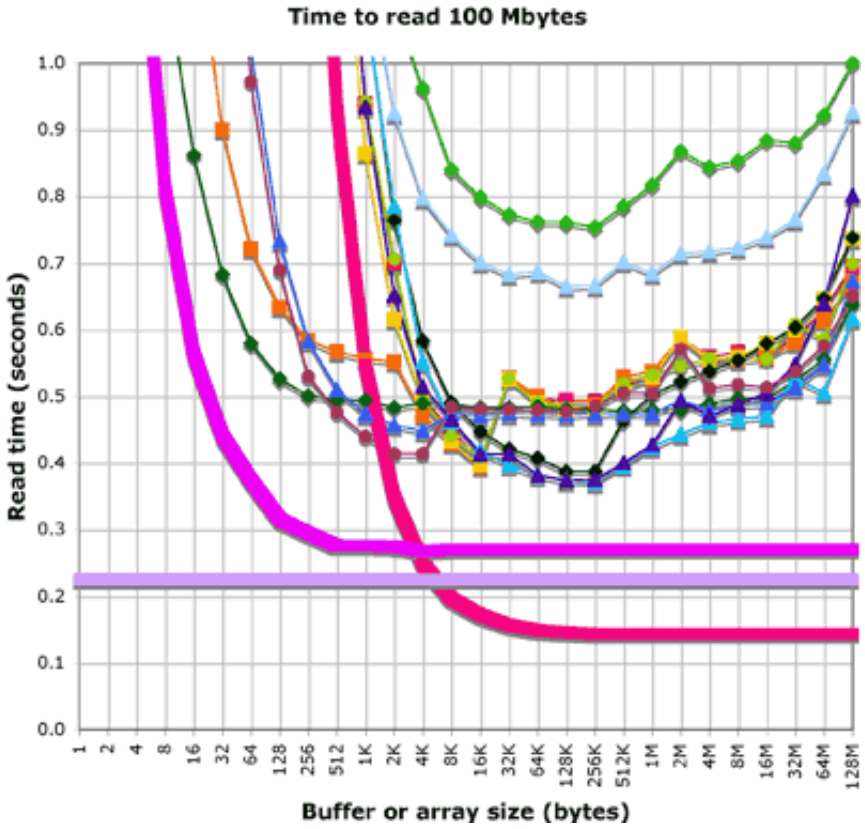
Zoomed in this far, the lines vary by about 200ms in the "flat" area above 8Kbytes. Monitoring the JVM's compilation and garbage collection found that neither of these were occurring during the tests. So these variations are due to other overhead within the JVM.

If we have to pick a fastest approach, it would be one of three:

- **`FileChannel`** with a **`MappedByteBuffer`** and array reads (the magenta line with round dots that hits a low at 2Kbytes to 4Kbytes).
- **`FileChannel`** with a direct **`ByteBuffer`** and array reads (the dark blue line with triangle dots that hits a low at 128Kbytes).
- **`FileChannel`** with a wrapped array **`ByteBuffer`** and direct array access (the light blue line with triangle dots that hits a low at 256Kbytes).

All three of these do better than the others by reducing the amount of data copying. They all enable the JVM to read new data into the application's own array without going through multiple intermediate buffers.

Comparison to C



How does Java I/O compare to C I/O? The upper thick line above shows C's buffered I/O (**`fopen()`** and **`fread()`**), the middle line shows memory mapping (**`mmap()`**), and the lower thick line shows unbuffered I/O (**`open()`** and **`read()`**). All of these C tests were compiled with gcc and the **`-fast`** option on a Mac.

C's buffered I/O uses an internal buffer filled by large reads from the file. Bytes from the buffer are copied to an application array on each **`fread()`** call. This is similar to **`FileChannel`** with a **`ByteBuffer`**.

C's unbuffered I/O reads bytes into the application's array without an internal buffer and an extra layer of byte copying. This gives better performance and is a similar approach to **`FileChannel`** with a wrapped array **`ByteBuffer`**.

C's memory mapping gives more direct access to the OS's paged-in chunks of the file. Memory mapping has more OS overhead, reducing its performance compared to unbuffered I/O. The approach is similar to **`FileChannel`** with a **`MappedByteBuffer`**.

C's performance is clearly better. But comparisons between C and Java are, perhaps, unfair. Java does array bounds checking, runtime type checking, thread synchronization, garbage collection, and so on. C does not. This lets C go faster, but with a higher risk of crashing if you've got a pointer error.

Conclusions

For the best Java read performance, there are four things to remember:

- **Minimize I/O operations by reading an array at a time, not a byte at a time.** An 8Kbyte array is a good size.
- **Minimize method calls by getting data an array at a time, not a byte at a time.** Use array indexing to get at bytes in the array.

- **Minimize thread synchronization locks if you don't need thread safety.** Either make fewer method calls to a thread-safe class, or use a non-thread-safe class like **`FileChannel`** and **`MappedByteBuffer`**.

- **Minimize data copying between the JVM/OS, internal buffers, and application arrays.** Use **`FileChannel`** with memory mapping, or a direct or wrapped array **`ByteBuffer`**.

Further reading

Tuning Java I/O Performance. This article at Sun's Developer Network discusses I/O approaches and also gives tips on text file handling, message printing, file attribute queries, and more.

Tweak your IO performance for faster runtime. This older article at JavaWorld discusses several I/O approaches and benchmarks them on five JVMs of the day. It's general points remain valid today, though newer JVMs are much more sophisticated and we now have low-level FileChannel I/O for better performance.

Comments

Very Helpful Article

Submitted by Anonymous (not verified) on December 5, 2008 - 8:51am.

Excellent work on this article. Was really helpful.

very thorough and

Submitted by Anonymous (not verified) on January 23, 2009 - 7:54am.

very thorough and informative analysis!

its very fundu article about

Submitted by jitendra (not verified) on January 28, 2009 - 1:27am.

its very fundu article about File I/o

Nice benchmark!

Submitted by Petri (not verified) on February 8, 2009 - 4:57am.

Very informative and useful article. Excellent research work. Great example codes.

Very interesting & helpful

Submitted by Chris (not verified) on June 26, 2009 - 12:22am.

really worth reading & trying!

Thx!

Great!!!...

Submitted by Anonymous (not verified) on August 3, 2009 - 7:09am.

Excellent article!!!.. congrats!

Excellent!!

Submitted by Sameer (not verified) on September 10, 2009 - 3:28am.

An extremely useful article. Thanks.

Amazing.Worth the time spent

Submitted by prashubk (not verified) on November 11, 2009 - 1:40pm.

Amazing.Worth the time spent

tnx

Submitted by kimi (not verified) on December 29, 2009 - 5:16am.

that was really cool!!!

reading this amaizing post, i was so excited that i couldn't wait to go and implement as concluded here.

i just wana thank you for your usefull research. you rock!!!

Nice Job

Submitted by Anonymous (not verified) on January 17, 2010 - 8:39pm.

Thanks for doing this work!

You got almost 500MB per sec

Submitted by Vlad (not verified) on January 28, 2010 - 2:21pm.

You got almost 500MB per sec sequential read speed on 40MB per sec interface. This is kind of achievement.

You actually have measured OS file cache read performance.

Re: You got almost 500MB per sec

Submitted by Dave_Nadeau on February 4, 2010 - 8:02pm.

Yup, and this was intentional. Each benchmark run started by loading the file into the OS cache in memory. The benchmark then ran repeatedly to access that file over and over using the Java methods being tested. Since the file was already in the OS cache, the benchmark reported only the cost of the Java methods, independent of disk speed, device drivers, etc. Using the fastest Java I/O, speeds should approach the limits of the OS cache and far surpass the speed of a disk. In the real world, though, disk speed is an issue and all of the Java I/O methods I tested will slow down. Still, the faster Java I/O methods will remain faster.

No, it isn't. Since all data

Submitted by cy6erGn0m (not verified) on October 2, 2012 - 8:43am.

No, it isn't. Since all data already in memory then we have no: data fragmentation effect, no I/O delays and blocking effects caused by read hardware bus. Also data copied from the cache to process user space but in case of direct buffer we can get data read from hardware to our user space directly. There are many differences and many nuances

that affects read performance.

So the only way to test it is using disk with disabled cache because in most cases we are interested in non-cached files read performance.

Re: No, it isn't. Since all data

Submitted by Dave_Nadeau on October 4, 2012 - 3:26pm.

Reading a file triggers actions to queue a disk I/O request, suspend the process until the I/O is done, wait for head and platter movement, read data from the disk to the disk's internal buffer, transfer that buffer across the disk and system buses into host memory, interrupt the processor, transfer data from kernel buffers to application buffers, schedule the process to run after it's been waiting on I/O, and finally transfer data from application buffers into whatever variable the programmer wants filled. Performance factors in all that include the latency and bandwidth of the disk, disk bus, system bus, memory (bus-to-memory and memory-to-CPU), and CPU cache, and CPU speed, cache sizes, etc.

When benchmarking, we have to reduce the number of variables to make sense of the result. While benchmarking a full system gets us a real-world number, it's very hard to connect cause and effect and deduce what specific bit of code or hardware is causing a slow result. Without connecting cause and effect we cannot determine what to change to improve performance. So, we have to simplify the testing.

The benchmarks reported in this article simplify to remove as many hardware variables as possible. Since disk I/O is very slow and depends a lot on system load, disk type, and where data is stored on disk, removing the disk I/O from the benchmark is essential to get stable results that focus on software effects, not hardware. This benchmark therefore intentionally ran with all file data in memory and no disk activity.

What we're left with is benchmarking the path from kernel disk caches to application variables by way of the Java VM and Java I/O classes. No matter what Java classes we use, that path is limited by the same memory and CPU cache latency and bandwidth costs. The only major differences are then due to the Java classes themselves and how they interact with the kernel. And that's exactly what we want to benchmark here. We want to know which Java classes for I/O incur the lowest overhead themselves, independent of the disk hardware.

Once we know which Java classes are the fastest to use, then we can shift to looking at other performance factors. How much faster is SSD vs. disk? How does the file system's block size affect speed? The OS's page size? Memory speed? Disk bus speed? And so on. Benchmarking any of these again requires simplifying to isolate the variables we're interested in and better connect cause with effect.

Complicating all that, we also need to consider our use cases. You suggest that we're mostly interested in non-cached file read performance. Sure, sometimes. If we're accessing a file infrequently, it won't be in the OS's disk cache and we'd like to know how much it costs to get it from disk. But there are also use cases where we access the same file very frequently... such as databases, system logs, shared libraries, directories, and so on. For those use cases the OS's disk cache is an essential tool to speed up operations. That's why it exists, after all.

So, the use cases matter. This article sought only to benchmark the overhead of Java I/O classes themselves. As you apply real use cases atop those classes, other performance factors emerge. This includes the OS's disk cache, but also whether you're accessing data sequentially or randomly, in big chunks or small, whether you're locking the file, whether you're writing to it too, and so on. There's always more to benchmark!

that was really

Submitted by Ragazze (not verified) on May 12, 2010 - 11:01am.

that was really cool!!!

reading this amaizing post, i was so excited that i couldn't wait to go and implement as concluded here.

i just wana thank you for your usefull research. you rock!!!

Good job! Thank you very

Submitted by Anonymous (not verified) on June 24, 2010 - 10:16pm.

Good job! Thank you very much.

this was great - I just

Submitted by Anonymous (not verified) on August 11, 2010 - 5:09am.

this was great - I just speeding a part of my app with 450%

Awesome

Submitted by Anonymous (not verified) on September 21, 2010 - 9:32pm.

Awesome

Simply ... excellent

Submitted by Anonymous (not verified) on September 26, 2010 - 12:55am.

"THE" reply to java I/O

Nice wok

Submitted by Deepak Agrawal (not verified) on July 19, 2012 - 12:13am.

thnx for dis post...

Excellent work!!!!!

Submitted by somu (not verified) on July 30, 2012 - 11:15pm.

such a excellent and great work... thanks!!!!!!!

good job

Submitted by Anonymous (not verified) on September 4, 2012 - 9:35pm.

very useful

Great Article

Submitted by Anonymous (not verified) on October 4, 2012 - 12:57pm.

Great job, the whole article was a neat from its start till the end. Can't wait to see more of your tips!.

Readling Line By Line using NIO

Submitted by Anonymous (not verified) on October 29, 2012 - 4:16am.

Hello,

Thanks for this very useful article. The problem I have is my file is very large and I want to be able to read it line by line. It seems that all NIO libraries do not support line by line and BufferedReader is a bit slow. Any idea?

Thanks

Re: Reading Line By Line using NIO

Submitted by Dave_Nadeau on November 1, 2012 - 5:47am.

At the lowest level, all I/O schemes read bytes. The I/O system doesn't know or care what those bytes are. To make a read operation stop on a specific byte (like end-of-line), code must use a loop and check each byte. Using a loop to read a byte, check a byte, and repeat, is very slow, so buffered code reads a big array of bytes first, then loops through the array to find the byte to stop on. The bytes up to that point are returned to the caller, while the remaining bytes are kept around for the next I/O call.

Looping through an array of bytes takes time. Since an "end-of-line" is undefined on bytes, but is defined on characters, classes that look for an end of line also convert bytes to characters, and that takes time. This makes the classes that read lines go slower, and it's unavoidable. If you wrote your own code to do this, it would run just as slow.

Reading a large file is an additional problem. Since lines in a text file don't have a fixed length, the only way to find the N-th line is to read all prior (N-1) lines and count end-of-line characters. That's very slow, and unavoidable. To make I/O on a big file cheaper, fast code indexes the file ahead of time and stores a table of byte offsets to the beginning of each line. Then getting the N-th line requires an index look-up, a seek into the file by the line's byte offset, and a read of the line's bytes. Better yet, store the line's length in the index so that you can read the correct number of bytes in one read operation. There are many more variations, such as binary files that store the index within the file itself, and multiple indexes to allow searching on line attributes. But all of these share the same goal: get to the N-th line without reading (N-1) lines first.

Thank you. what about writing?

Submitted by TachisAlopex (not verified) on December 29, 2012 - 7:17pm.

Thank you so much. I'm a beginner to buffers and array reads, so far everything I've written has been though Scanner and other simple classes. I rewrote my code using this information and my reads are around 20x faster! I love how you provided example code, because without it, I would be at a loss...

I do have one question though: what about writing? Is FileChannel/(Mapped)ByteBuffer still the king there?

Thanks,
Tach

Very helpful! Great Job!

Submitted by pbergn (not verified) on February 8, 2013 - 9:44am.

Great job! I am impressed... Very helpful too...

Little Doubt on the results

Submitted by Yizhou (not verified) on February 17, 2013 - 10:32am.

The **`ByteBuffer`**'s JavaDoc said:

an invocation of this method of the form src.get(dst, off, len) has exactly the same effect as the loop
for (int i = off; i < off + len; i++)
dst[i] = src.get();
except that it first checks that there are sufficient bytes in this buffer and it is potentially much more efficient.

So I doubt *array gets* for **`ByteBuffer`** won't be faster than *byte gets*. On contrary, I think it should be slower as you did redundant reading.

This is extremely helpful,

Submitted by Anonymous (not verified) on March 9, 2013 - 7:39am.

This is extremely helpful, thanks for taking the time and effort to put this together.

Great Job!!!. Nice article!!!

Submitted by Anonymous on September 6, 2013 - 6:43am.

Great Job!!!. Nice article!!!