Efficient data transfer through zero copy

Zero copy, zero overhead

by Sathish Palaniappan, Pramod Nagaraja | Published September 2, 2008

Java

Many Web applications serve a significant amount of static content, which amounts to reading data off of a disk and writing the exact same data back to the response socket. This activity might appear to require relatively little CPU activity, but it's somewhat inefficient: the kernel reads the data off of disk and pushes it across the kernel-user boundary to the application, and then the application pushes it back across the kernel-user boundary to be written out to the socket. In effect, the application serves as an inefficient intermediary that gets the data from the disk file to the socket.

Each time data traverses the user-kernel boundary, it must be copied, which consumes CPU cycles and memory bandwidth. Fortunately, you can eliminate these copies through a technique called — appropriately enough —*zero copy*. Applications that use zero copy request that the kernel copy the data directly from the disk file to the socket, without going through the application. Zero copy greatly improves application performance and reduces the number of context switches between kernel and user mode.

The Java class libraries support zero copy on Linux and UNIX systems through the `transferTo()` method in `java.nio.channels.FileChannel`. You can use the `transferTo()` method to transfer bytes directly from the channel on which it is invoked to another writable byte channel, without requiring data to flow through the application. This article first demonstrates the overhead incurred by simple file transfer done through traditional copy semantics, then shows how the zero-copy technique using `transferTo()` achieves better performance.

Date transfer: The traditional approach

Consider the scenario of reading from a file and transferring the data to another program over the network. (This scenario describes the behavior of many server applications, including Web applications serving static content, FTP servers, mail servers, and so on.) The core of the operation is in the two calls in Listing 1 (download the complete sample code):

```
File.read(fileDesc, buf, len);
Socket.send(socket, buf, len);
```

Although Listing 1 is conceptually simple, internally, the copy operation requires four context switches between user mode and kernel mode, and the data is copied four times before the operation is complete. Figure 1 shows how data is moved internally from the file to the socket:

Figure 1. Traditional data copying approach

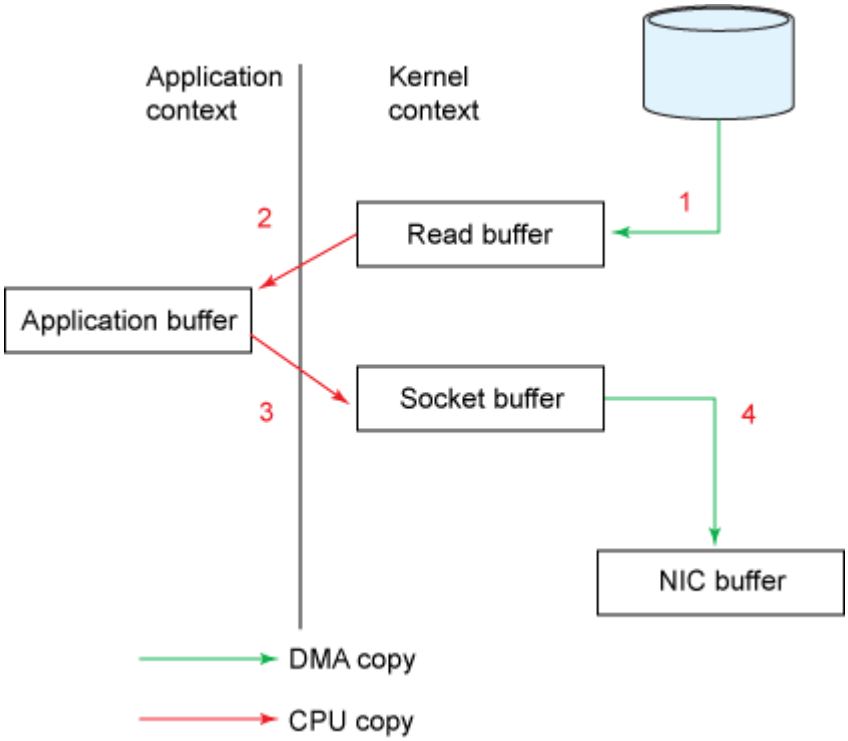
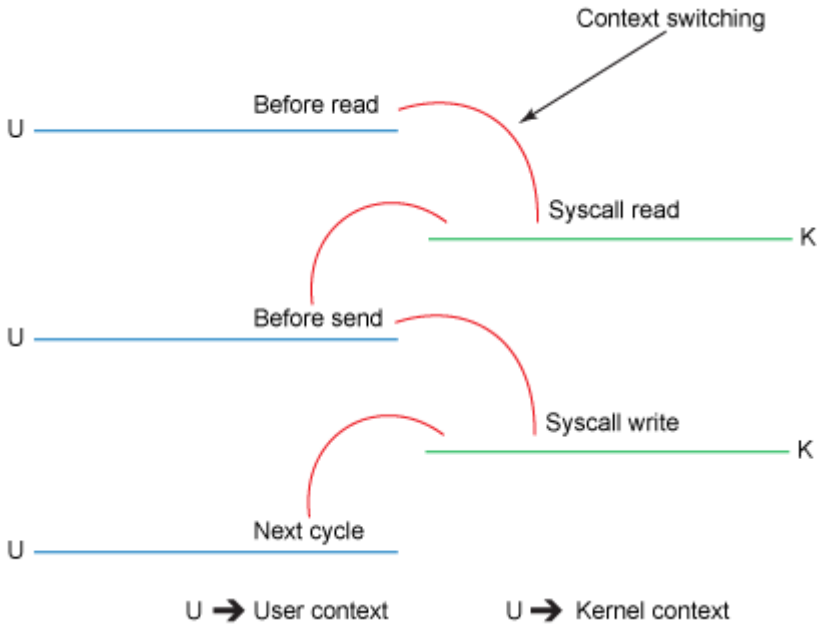


Figure 2 shows the context switching:

Figure 2. Traditional context switches



The steps involved are:

1. The `read()` call causes a context switch (see Figure 2) from user mode to kernel mode. Internally a `sys_read()` (or equivalent) is issued to read the data from the file. The first copy (see Figure 1) is performed by the direct memory access (DMA) engine, which reads file contents from the disk and stores them into a kernel address space buffer.
2. The requested amount of data is copied from the read buffer into the user buffer, and the `read()` call returns. The return from the call causes another context switch from kernel back to user mode. Now the data is stored in the user address space buffer.

3. The `send()` socket call causes a context switch from user mode to kernel mode. A third copy is performed to put the data into a kernel address space buffer again. This time, though, the data is put into a different buffer, one that is associated with the destination socket.
4. The `send()` system call returns, creating the fourth context switch. Independently and asynchronously, a fourth copy happens as the DMA engine passes the data from the kernel buffer to the protocol engine.

Use of the intermediate kernel buffer (rather than a direct transfer of the data into the user buffer) might seem inefficient. But intermediate kernel buffers were introduced into the process to improve performance. Using the intermediate buffer on the read side allows the kernel buffer to act as a "readahead cache" when the application hasn't asked for as much data as the kernel buffer holds. This significantly improves performance when the requested data amount is less than the kernel buffer size. The intermediate buffer on the write side allows the write to complete asynchronously.

Unfortunately, this approach itself can become a performance bottleneck if the size of the data requested is considerably larger than the kernel buffer size. The data gets copied multiple times among the disk, kernel buffer, and user buffer before it is finally delivered to the application.

Zero copy improves performance by eliminating these redundant data copies.

Data transfer: The zero-copy approach

If you re-examine the traditional scenario, you'll notice that the second and third data copies are not actually required. The application does nothing other than cache the data and transfer it back to the socket buffer. Instead, the data could be transferred directly from the read buffer to the socket buffer. The `transferTo()` method lets you do exactly this. Listing 2 shows the method signature of `transferTo()`:

Listing 2. The transferTo() method

```
public void transferTo(long position, long count, WritableByteChannel target);
```

The `transferTo()` method transfers data from the file channel to the given writable byte channel. Internally, it depends on the underlying operating system's support for zero copy; in UNIX and various flavors of Linux, this call is routed to the `sendfile()` system call, shown in Listing 3, which transfers data from one file descriptor to another:

Listing 3. The sendfile() system call

```
#include <sys/socket.h>
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

The action of the `file.read()` and `socket.send()` calls in Listing 1 can be replaced by a single `transferTo()` call, as shown in Listing 4:

Listing 4. Using transferTo() to copy data from a disk file to a socket

```
transferTo(position, count, writableChannel);
```

Figure 3 shows the data path when the `transferTo()` method is used:

Figure 3. Data copy with transferTo()

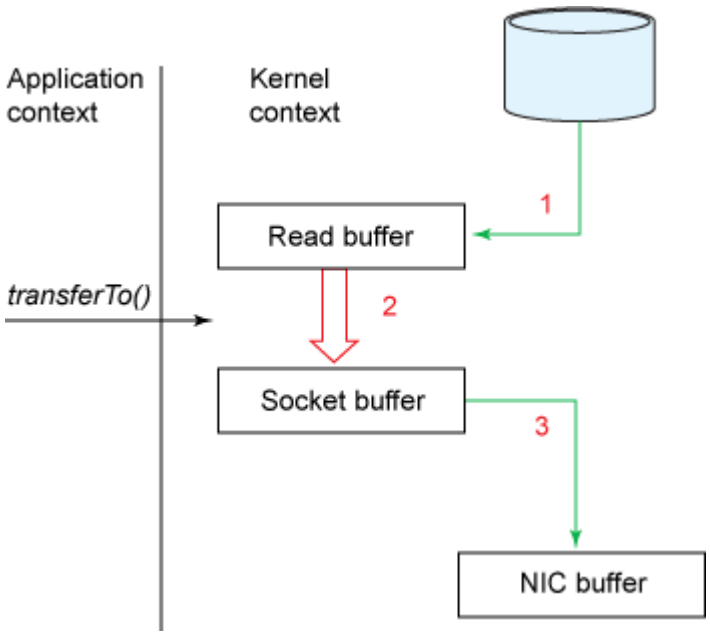
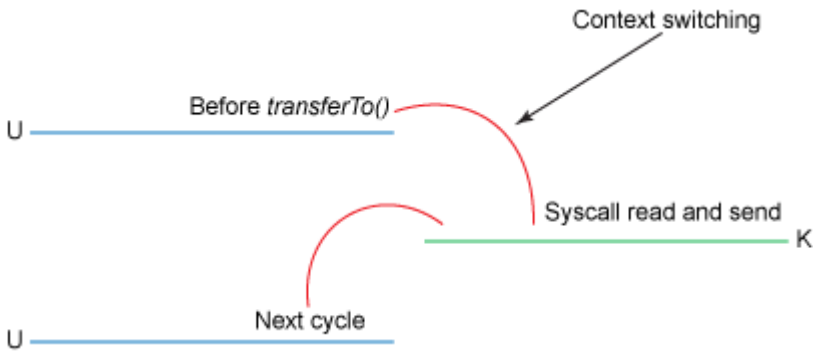


Figure 4 shows the context switches when the `transferTo()` method is used:

Figure 4. Context switching with transferTo()



The steps taken when you use `transferTo()` as in Listing 4 are:

1. The `transferTo()` method causes the file contents to be copied into a read buffer by the DMA engine. Then the data is copied by the kernel into the kernel buffer associated with the output socket.
2. The third copy happens as the DMA engine passes the data from the kernel socket buffers to the protocol engine.

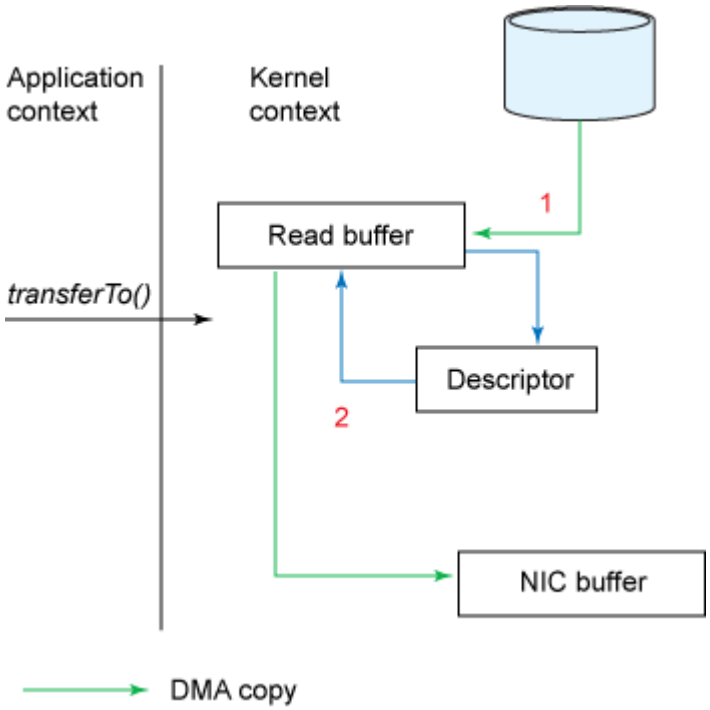
This is an improvement: we've reduced the number of context switches from four to two and reduced the number of data copies from four to three (only one of which involves the CPU). But this does not yet get us to our goal of zero copy. We can further reduce the data duplication done by the kernel if the underlying network interface card supports *gather operations*. In Linux kernels 2.4 and later, the socket buffer descriptor was modified to accommodate this requirement. This approach not only reduces multiple context switches but also eliminates the duplicated data copies that require CPU involvement. The user-side usage still remains the same, but the intrinsics have changed:

1. The `transferTo()` method causes the file contents to be copied into a kernel buffer by the DMA engine.

2. No data is copied into the socket buffer. Instead, only descriptors with information about the location and length of the data are appended to the socket buffer. The DMA engine passes data directly from the kernel buffer to the protocol engine, thus eliminating the remaining final CPU copy.

Figure 5 shows the data copies using `transferTo()` with the gather operation:

Figure 5. Data copies when transferTo() and gather operations are used



Building a file server

Now let's put zero copy into practice, using the same example of transferring a file between a client and a server (see Download for the sample code). `TraditionalClient.java` and `TraditionalServer.java` are based on the traditional copy semantics, using `File.read()` and `Socket.send()`. `TraditionalServer.java` is a server program that listens on a particular port for the client to connect, and then reads 4K bytes of data at a time from the socket. `TraditionalClient.java` connects to the server, reads (using `File.read()`) 4K bytes of data from a file, and sends (using `socket.send()`) the contents to the server via the socket.

Similarly, `TransferToServer.java` and `TransferToClient.java` perform the same function, but instead use the `transferTo()` method (and in turn the `sendfile()` system call) to transfer the file from server to client.

Performance comparison

We executed the sample programs on a Linux system running the 2.6 kernel and measured the run time in milliseconds for both the traditional approach and the `transferTo()` approach for various sizes. Table 1 shows the results:

Table 1. Performance comparison: Traditional approach vs. zero copy

File size	Normal file transfer (ms)	transferTo (ms)
7MB	156	45
21MB	337	128
63MB	843	387
98MB	1320	617
200MB	2124	1150
350MB	3631	1762
700MB	13498	4422
1GB	18399	8537

As you can see, the `transferTo()` API brings down the time approximately 65 percent compared to the traditional approach. This has the potential to increase performance significantly for applications that do a great deal of copying of data from one I/O channel to another, such as Web servers.

Summary

We have demonstrated the performance advantages of using `transferTo()` compared to reading from one channel and writing the same data to another. Intermediate buffer copies — even those hidden in the kernel — can have a measurable cost. In applications that do a great deal of copying of data between channels, the zero-copy technique can offer a significant performance improvement.