Journal 4, Performance test

1 Objectives

The performance of the created set of drivers and applications for both Windows and Linux needs to be tested. This will be done on both platforms under various environments.

2 Problem analysis

The theoretical maximum transfer speed across the PCI Express x1 interface is 250 MB/s in each direction (data will only be transmitted in one way at a time during this test). It is however not expected that effective transfer speeds of this magnitude can be reached with programmed I/O. As the x86 architecture of the host system is limited to transfer 1 DWORD per read or write command, there will be quite a bit of overhead. Just for the transaction layer, a read command consists of a total of 7 DWORDs being transmitted (3 for the request and 4 for the completion), while a write request consists of 4 DWORDs (4 for the request, no completion is sent as memory write is a posted transaction). On top of this comes additional overhead from the datalink and physical layers. Also, as both Windows and Linux use multitasking, it is not to be expected that a driver will be allowed to transmit and receive data uninterrupted.
To test the performance, four different tests in various environments are performed. The tests are:

- *Application-controlled write:* A register on the Spartan3 board is repeatedly written 1000000 times using a for-loop in the application, which calls the write-register driver function for each run through the loop. The execution time of the for-loop is then timed.
- *Application-controlled read:* Similarly, this performs 1000000 reads, using the same technique as described above.
- *Driver-controlled write:* The two above tests will result in extra overhead as the driver is called once for each of the 1000000 reads or writes. This test instead places the for-loop inside the driver, so that only one request is send from the application. This requires a new function and control code to be added to the drivers, including functionality to time the for-loops, and return the time taken to the application.
- *Driver-controlled read:* This works in a similar way to the driver-controlled write, but is just performing reads instead.

3 Software and FPGA design

The drivers and applications designed and created in journal 3 are used, with the mentioned functionality added. The code for driver and applications can be found on CD1 in the folder "*Source\Windows_Device_Drivers_and_Applications\Performancetest*" for the Windows version, and in "*Source\Windows_Device_Drivers_and_Applications\Performancetest*" for the Linux version. The *IOControlDemo* FPGA design is used again. Further info can be found in the "Performance test" applicationguide.
Below is described how the timing is performed.

3.1 Timing in Windows, application-control

For the application-controlled test in Windows, the *GetTickCount()* function is used. This returns the number of milliseconds elapsed since the system was booted. By calling this function right before and right after the reads/writes, the number of milliseconds used can be found by taking the difference between the end and start times. Roll-over is not considered an issue, as this only happens every ~50 days.

3.2 Timing in Windows, driver-control

GetTickCount() is only available from user-mode applications, so instead the kernel-mode function *KeQueryTickCount()* is used. This returns the number of ticks since system boot. The amount of ticks it takes to get through the loop is returned to the application as the first value in *pBuffer*. The duration of a tick varies from system to system, so this needs to be send back to the application also. The tick duration is retrieved through *KeQueryTimeIncrement()*, and is stored as the second value in *pBuffer*.

3.3 Timing in Linux, application-control

Under Linux, the function *GetTimeOfDay()* is used for user mode timing. It returns the current time as the number of seconds and microseconds since 1/1 1970.

3.4 Timing in Linux, driver-control

Similarly to *KeQueryTickCount()*, the amount of ticks after system boot are used here. In Linux, this value is stored in the system-wide *jiffies* variable. The tick duration can be derived from the system-wide *HZ* variable, which specifies the number of ticks per second. A separate driver-call is created for returning the *HZ* value to the application.

4 The test

The tests will be performed in the following environments:

- Windows, checked build, HT enabled
- Windows, checked build, HT disabled
- Windows, free build, HT enabled
- Windows, free build, HT disabled
- Linux, console, HT enabled
- Linux, console, HT disabled
- Linux, kde, HT enabled
- Linux, kde, HT disabled

The reason for testing with HyperThreading (HT) both enabled and disabled is that it might affect the number of necessary context switches, and thereby impact performance. HT can be enabled / disabled through the system bios at boot.
It should be noted that due to differences in the time measurement functions, the resolution of the timing varies between Windows and Linux, and between user- and kernel-mode.
All tests are run three times, and the average time is used. The complete set of results can be seen in the folder "*Data\Performance test*" on CD1. The effective transfer speeds (the amount of actual data transferred) can be seen in figure 4.a.

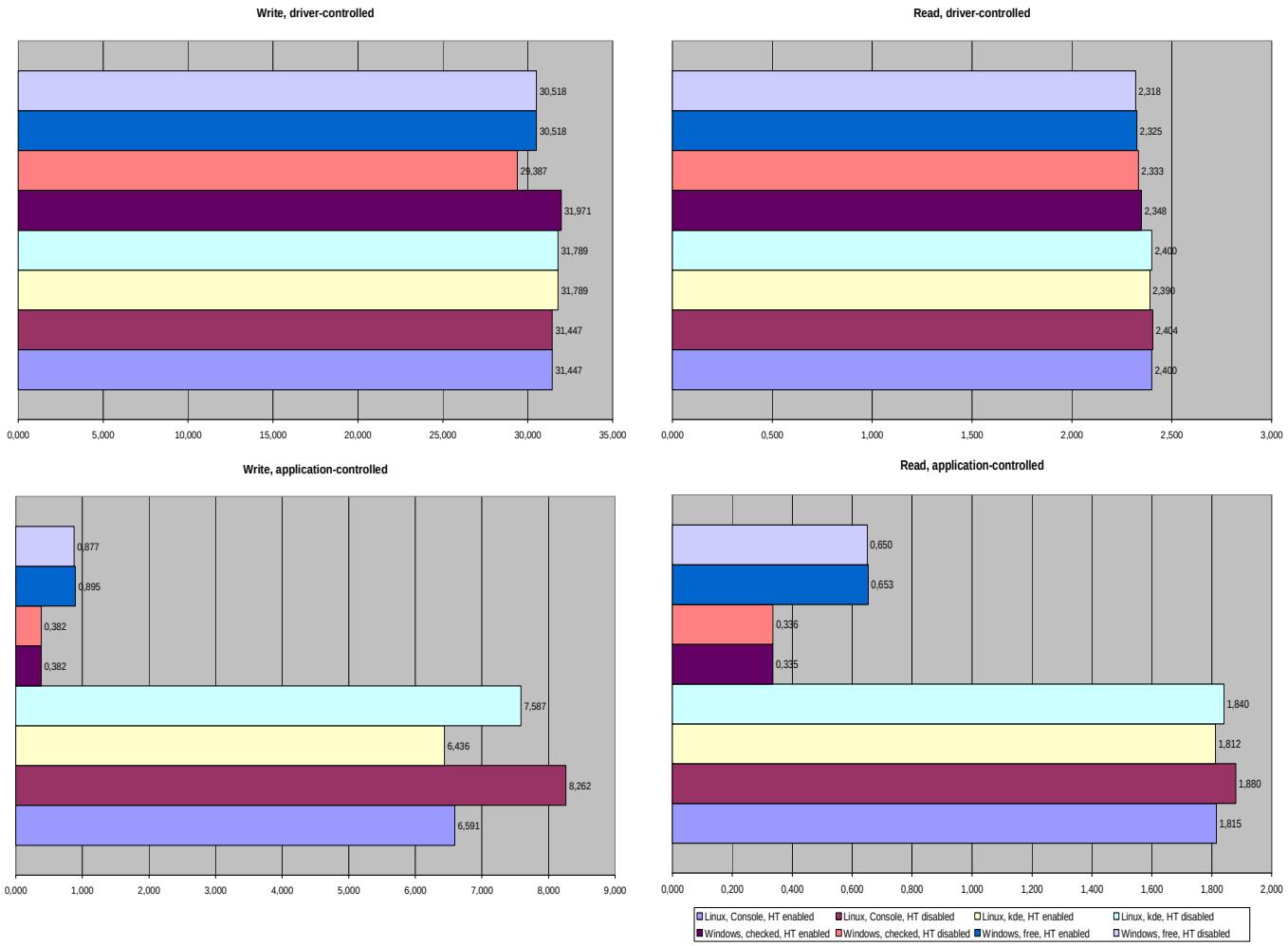

Figure 4.a: The results of the test. All graphs show the effective transfer speeds, values are in MB/s.

The first thing to notice is that the transfer speeds, as expected, are far away from the theoretical maximum of 250 MB/s. A maximum effective write speed of about 30 MB/s, and a maximum effective read speed of about 2.5 MB/s are achieved, when using the driver-controlled approach. With application-control, and thus a substantial increase in the amount of overhead, about 7 MB/s for writes and 2 MB/s for reads is reached with Linux and less than 1 MB/s either way under Windows. The reason for the larger performance hit with application-control on Windows than on Linux, is probably due to differences in the way device drivers are accessed, and/or the internal scheduling of processes within the kernels. This is also evident as the tests with driver-controlled writes and reads, for which the kernel-mode code on Linux and Windows is nearly identical, result in very similar performance for both Windows and Linux.

Recall that, considering the overhead, the actual transfer speed for the write operations is roughly 4 times higher than the effective speed, while it is about 7 times higher for the read operations. This is not enough to explain the large difference between read and write speeds though. But the difference could be due to the read operations being non-posted, resulting in the OS switching to another process while waiting for a response to the read request. The write operations however, are posted transactions, and can thus be performed without needing to wait.

The difference between checked and free builds under Windows is also clear. With free builds, all debug information (including test output to the debug console) is stripped from the driver. This does not have much of an impact with driver-control, as the for-loops do not include any debug messages. The single-register versions of the write and read functions used with application-control do however write a line to the debug-console each time they are called, resulting in the lowered performance.

It is also interesting to see that while not having much impact in most of the tests, enabling HT actually noticeably decreases performance under Linux with application-control. The reason for this is unknown, as HT is notorious for giving performance boosts under certain conditions, while in other cases slowing things down.

5 Conclusion

The performance of the created drivers and applications has now been tested. As there is a substantial amount of overhead present, in both the host OS and in the PCI Express transfer itself, the performance is far away from the theoretical maximum of 250 MB/s. By using the driver-controlled approach and thus minimizing the overhead in the host OS, a best-case transfer speed of ~30 MB/s when writing to and ~2.5 MB/s when reading from the board has been achieved on both platforms. This only works for large transfers where all data is transferred to/from the driver in one request. When transferring single DWORDS, the additional overhead of calling the driver has a higher influence, and drops the transfer speeds to well below 1 MB/s in both directions on Windows, and to around 7 MB/s writing and 1.8 MB/s reading on Linux.
Sadly this is far away from the desired transfer speed of 180 MB/s. It should therefore be investigated if it is possible to improve on the current performance.