# Exploring Storage Bottlenecks in Linux-based Embedded Systems

Russell Joyce
Real-Time Systems Research Group
Department of Computer Science
University of York, UK
russell.joyce@york.ac.uk

Neil Audsley
Real-Time Systems Research Group
Department of Computer Science
University of York, UK
neil.audsley@york.ac.uk

## ABSTRACT

With recent advances in non-volatile memory technologies and embedded hardware, large, high-speed persistent-storage devices can now realistically be used in embedded systems. Traditional models of storage systems, including the implementation in the Linux kernel, assume the performance of storage devices to be far slower than CPU and system memory speeds, encouraging extensive caching and buffering over direct access to storage hardware. In an embedded system, however, processing and memory resources are limited while storage hardware can still operate at full speed, causing this balance to shift, and leading to the observation of performance bottlenecks caused by the operating system rather than the speed of storage devices themselves.

In this paper, we present performance and profiling results from high-speed storage devices attached to a Linux-based embedded system, showing that the kernel's standard file I/O operations are inadequate for such a set-up, and that 'direct I/O' may be preferable for certain situations. Examination of the results identifies areas where potential improvements may be made in order to reduce CPU load and increase maximum storage throughput.

## Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

## General Terms

Design, Measurement, Performance

## Keywords

Linux, storage

## 1. INTRODUCTION

Traditionally, access to persistent storage has been orders of magnitude slower than volatile system memory, especially when performing random data accesses, due to the high latency and low bandwidth associated with the mechanical operation of hard disk drives, as well as the constant increase in CPU and memory speeds over time. Despite the deceleration of single-core CPU scaling in recent years, the main bottleneck associated with accessing non-volatile storage in a general-purpose system is still typically the storage device itself.

Linux (along with many other operating systems) uses a number of methods to reduce the impact that slow storage devices cause on overall system performance. Firstly, main memory is heavily used to cache data between block device accesses, avoiding unnecessary repeated reads of the same data from disk. This also helps in the efficient operation of file systems, as structures describing the position of files on a disk can be cached for fast retrieval. Secondly, buffers are provided for data flowing to and from persistent storage, which allow applications to spend less time waiting on disk operations, as these can be performed asynchronously by the operating system without the application necessarily waiting for their completion. Finally, sophisticated scheduling and data layout algorithms can be used to optimise the data that is written to a device, taking advantage of idle CPU time caused by the system waiting for I/O operations to complete.

For a general-purpose Linux system, these techniques can have a large positive effect on the efficient use of storage – memory and the CPU often far outperform the speed of a hard disk drive, so any use of them to reduce disk accesses is desirable. However, this relationship between CPU, memory and storage speeds does not hold in all situations, and therefore these techniques may not always provide a benefit to the performance of a system.

The limited resources of a typical embedded system can skew the balance between storage and CPU speed, which can cause issues for a number of embedded applications that require fast and reliable access to storage. Examples of these include applications that receive streaming data over a high-speed interface that must be stored in real-time, such as data being sent from sensors or video feeds, perhaps with intermediate processing being performed using hardware accelerators.

This paper considers effects that the limited CPU and memory speeds of an embedded system can have on a fast storage device – due to the change in balance between relative speeds, the system cannot be expected to perform in the same way as a typical computer, with certain performance bottlenecks shifting away from storage hardware limitations

*EWiLi'15,* October 8th, 2015, Amsterdam, The Netherlands.

and into software operations.

Results are presented in section 3 from basic testing of storage devices in an embedded system, showing that sequential storage operations experience bottlenecks caused by CPU limitations rather than the speed of the storage hardware if standard Linux file operations are used. Removing reliance on the page cache (through direct I/O) is shown to improve performance for large block sizes, especially on a fast SSD, due to the reduction in the number of times data is copied in main memory.

Potential solutions briefly presented in section 5 suggest that restructuring the storage stack to favour device accesses over memory and CPU usage in this type of system, as well as more radical changes such as the introduction of hardware accelerators, may reduce the negative effects of CPU limitations on storage speeds.

## 2. PROBLEM SUMMARY

Recent advances in flash memory technology have caused the widespread adoption of Solid-State Drives (SSDs), which offer far faster storage access compared to mechanical hard drives, along with other benefits such as lower energy consumption and more-uniform access times. It is anticipated that over the next several years, further advances in non-volatile memory technologies will accelerate the increasing trend in storage device speeds, potentially allowing for large, non-volatile memory devices that operate with similar performance to volatile RAM. At a certain point, fast storage speeds, relative to CPU and system memory speeds, will cause a critical change in the balance of a system, requiring a significant reconsideration of an operating system's approach to storage access [10].

At present, this shift in the balance of system performance is beginning to affect the embedded world, where processing and memory speeds are typically low due to constraints such as energy usage, size and cost, but where fast solid-state storage still has the potential to run at the same speed as in a more powerful system. For example, an embedded system consisting of a slow, low-core-count CPU and slowly-clocked memory connected to a high-end, desktop-grade SSD has a far different balance between storage, memory and CPU than is expected by the operating system design. While such a system may run Linux perfectly adequately for many tasks, it will not be able to take advantage of the full speed of the SSD using traditional methods of storage access, due to bottlenecks elsewhere in the system.

Before fast solid-state storage was common, non-volatile storage in an embedded system would often consist of slow flash memory, due to the high energy consumption and low durability of faster mechanical media, meaning the potential increase in secondary storage speeds provided by SSDs is even greater in embedded systems than many general-purpose systems. An increase in the general storage requirements and expectations for systems, driven by fields such as multimedia and 'big data' processing, have also accelerated the adoption of fast solid-state storage in embedded systems.

### 2.1 Buffered vs Direct I/O

The Linux storage model relies heavily on the buffering and caching of data in system memory, typically requiring data to be copied multiple times before it reaches its ultimate destination. The kernel provides the 'direct I/O' file access method to reduce the amount of memory activity

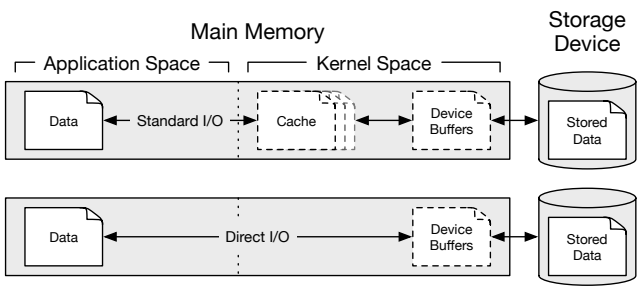


**Figure 1: The operation of direct I/O in the Linux kernel, bypassing the page cache**

involved in reading and writing data from a block device, allowing data to be copied directly to and from an application's memory space without being transferred via the page cache. While this allows applications more-direct access to storage devices, it can also create restrictions and have a severe negative impact on storage speeds if used incorrectly. In the past, there has been some resistance to the direct I/O functionality of Linux [11], partly due to the benefits of utilising the page cache that are removed with direct I/O, and the large disparity between CPU/memory and storage speeds meaning there were rarely any situations where the overhead of additional memory copies was significant enough to cause a slowdown. However, when storage is fast and the speed of copying data around memory is slow, using direct I/O can have a significant performance improvement if certain criteria are met.

Figure 1 shows the basic principles of standard and direct I/O, with direct I/O bypassing the page cache and removing the need to copy data from one area of memory to another between storage devices and applications.

One of the main issues with direct I/O is the large overhead caused when dealing with data in small block sizes. Even when using a fast storage device, reading and writing small amounts of data is far slower per byte than larger sizes, due to constant overheads in communication and processing that do not scale with block size. Without kernel buffers in place to help optimise disk accesses, applications that use small block sizes will suffer greatly in storage speed when using direct I/O, compared to when utilising the kernel's data caching mechanisms, which will queue requests to more efficiently access hardware. The performance of accessing large block sizes on a storage device does not suffer from this issue, however, so applications that either inherently use large block sizes, or use their own caching mechanisms to emulate large block accesses, can use direct I/O effectively where required.

A further issue with the implementation of direct I/O in the Linux kernel is that it is not standardised, and is not part of the POSIX specification, so its behaviour and safety cannot necessarily be guaranteed for all situations. The formal definition of the `O_DIRECT` flag for the `open()` system call is simply to "try to minimize cache effects of the I/O to and from this file" [1], which may be interpreted differently (or not at all) by various file systems and kernel versions.

### 2.2 Real-time Storage Implications

Many high-performance storage applications require consideration of real-time constraints, due to external producers

or consumers of data running independently of the system storing it – if a storage system cannot save or provide data at the required speed then critical information may be lost or the system may malfunction. While solid-state storage devices have far more consistent access times than mechanical storage, making them more suitable for time-critical applications, if the CPU of a system is proving to be a bottleneck in storage access times, the ability to maintain a consistent speed of data access relies heavily on CPU utilisation.

If the CPU can be removed as far as possible from the operation of copying data to storage, the impact of other processes on this will be reduced, increasing the predictability of storage operations and making real-time guarantees more possible. This could be achieved through methods such as hardware acceleration, as well as simplification of the software storage stack.

## 2.3 Motivation

A number of examples exist where fast and reliable access to storage is required by an embedded system, which may be limited by CPU or memory resources when standard Linux file system operations are used.

Embedded accelerators are increasingly being investigated for use in high-performance computing environments, due to their energy efficiency when compared to traditional server hardware [8, 7]. CPU usage when performing storage operations has also been identified as an issue in server situations, using large amounts of energy compared to storage devices themselves, and motivating research into how storage systems can be made to be more efficient [4, 9].

Standalone embedded systems that use storage devices also create motivation for efficient access to storage, for applications such as logging sensor data and recording high-bandwidth video streams [6]. Often, external data sources will have constraints on the speeds required for their storage, for example, with the number of frames of video that must be stored each second, so any methods that can help to meet these requirements while keeping energy usage at a minimum are desirable.

Consider a basic Linux application that reads a stream of data from a network interface and writes it to a continuous file on secondary storage using standard file operations. Disregarding any other system activity and additional operations performed by the file system, data will be copied a minimum of six times on its path from network to disk:

1. From the network device to a buffer in the device driver
2. From the driver's buffer to a general network-layer kernel buffer
3. From the kernel buffer to the application's memory space
4. From the application's memory space to a kernel file buffer
5. From the file buffer to the storage device driver
6. From the driver's buffer to the storage device itself

This process has little impact on overall throughput if either storage or network speed is slow relative to main memory and CPU, however as soon as this balance changes, any additional memory copying can have a severe impact. Techniques such as DMA can help to reduce the CPU load related to copying data from one memory location to another, however this relies on hardware and driver support, and does not fully tackle the inefficiencies of unnecessary memory copies.

One advantage of the kernel using its page cache to store a copy of data is the ability to access that data at a later time without having to load it from secondary storage, however this will have no benefit if data is solely being written to or read from a disk as part of a streaming application, because by the time the data is needed a second time it is likely that it has already been purged from the cache.

# 3. EXPERIMENTAL WORK

In order to examine the effects that a slow system can have on the performance of storage devices, and to identify the potential bottlenecks present in the Linux storage stack, we performed a number of experiments with storage operations while collecting profiling and system performance information.

## 3.1 Experimental Set-up

The experimental set-up consisted of a an Avnet ZedBoard Mini-ITX development board connected to storage devices using its PCI Express Gen2 x4 connector. The ZedBoard Mini-ITX provides a Xilinx Zynq-7000 system-on-chip, which combines a dual-core ARM Cortex-A9 processor (clocked at 666MHz) with a large amount of FPGA fabric, alongside 1GiB of DDR3 RAM and many other on-board peripherals.

The system uses Linux 3.18 (based on the Xilinx 2015.2 branch) running on the ARM cores, while an AXI-to-PCIe bridge design is programmed on the FPGA, to provide an interface between the processor and PCI Express devices.

To provide a range of results, two storage devices were tested with the system: a Western Digital Blue 500GB SATA III hard disk drive, connected though a Startech SATA III RAID card; and an Intel SSD 750 400GB. While both devices use the same PCI Express interface for their physical connection to the board, the RAID card uses AHCI for its logical storage interface, whereas the SSD uses the more efficient NVMe interface.

Due to limitations of the high-speed serial transceiver hardware on the Zynq SoC, the speed of the SSD interface is limited to PCI Express Gen2 x4 (from its native Gen3 x4), reducing the maximum four-lane bandwidth from 3940MB/s to 2000MB/s. While this is still far faster than the 600MB/s maximum of the SATA-III interface used by the HDD, it means the SSD will never achieve its advertised maximum capable speed of 2200MB/s in this hardware set-up.

## 3.2 Data Copy Tests

To determine an indication of the operating speeds of the storage devices at various block sizes with minimal external overhead, we performed basic testing using the Linux `dd` utility. For write tests, `/dev/zero` was used as a source file, and for read tests, `/dev/null` was used as a destination. Both storage devices were freshly formatted with an *ext4* file system before each test.

For each block size and storage device, four tests were performed: reading from a file on the device, writing to a file on the device, and reading and writing with direct I/O enabled (using the `iflag=direct` and `oflag=direct` operands of `dd` respectively). Additionally, read and write tests were performed with a 512MiB *tmpfs* RAM disk (`/dev/shm`) in order to determine possible maximum speeds when no external storage devices or low-level drivers were involved. Each test was performed with and without the capture of system

resource usage and collection of profiling data, so results could be gathered without any additional overheads caused by these measurements.

With the secondary storage devices, data was recorded for 20GiB sequential transfers, and with the RAM disk, 256MiB transfers were used due to the lower available space. Sequential transfers are used as they represent the type of problem that is likely to be encountered when requiring high-speed storage in an embedded system – reading and writing streams of contiguous data – as well as being simple to implement and test. Storage devices, especially mechanical hard disks, generally perform faster with sequential transfers than with random accesses, and operating and file system overheads are also likely to be greater for non-sequential access patterns, so further experimentation will be necessary to determine whether the same effects are present when using different I/O patterns.

## 3.3 System Resource Usage and Profiling

To collect information about system resource usage during each test, we used the `dstat` utility [2] to capture memory usage, CPU usage and storage device transfer speeds each second.

Additionally, to determine the amount of execution time that is spent in each relevant function within the user application, kernel and associated libraries during the tests, we ran `dd` within the full-system profiler, `operf`, part of the *OProfile* suite of tools [3]. The impact on performance caused by profiling is kept to a minimum through support from the CPU hardware and the kernel performance events subsystem, however slight overheads are likely while the profiler is running, potentially causing slower speeds and slight differences in observed data.

## 3.4 Results

The following results were gathered using the system and methods described above, in order to investigate various aspects of storage operations.

### 3.4.1 Read and Write Speeds

Figure 2 shows the average read and write speeds for a number of block sizes when transferring data to or from the storage devices and RAM disk.

The standard read and write speeds for both storage devices are very similar, with the SSD only performing slightly faster than the HDD for all block sizes tested. This suggests that bottlenecks exist outside of the storage devices in the test system, either caused by the CPU or system memory bandwidth, as it is expected that the SSD should perform significantly faster than the HDD in both read and write speed.

For the 512B block size, speeds are slower on both devices, however there is little difference in speed once block sizes increase above this. This slow speed could be due to the significant number of extra context switches at a low block size being a bottleneck, rather than the factors limiting storage operations at 4KiB and above.

The consistently slightly higher speeds seen with the SSD are likely to be caused by it using an NVMe logical interface to communicate with the operating system, compared to the less-efficient AHCI interface used by the SATA HDD. If the storage operations are indeed experiencing a CPU bottleneck, then the more-efficient low-level drivers of NVMe

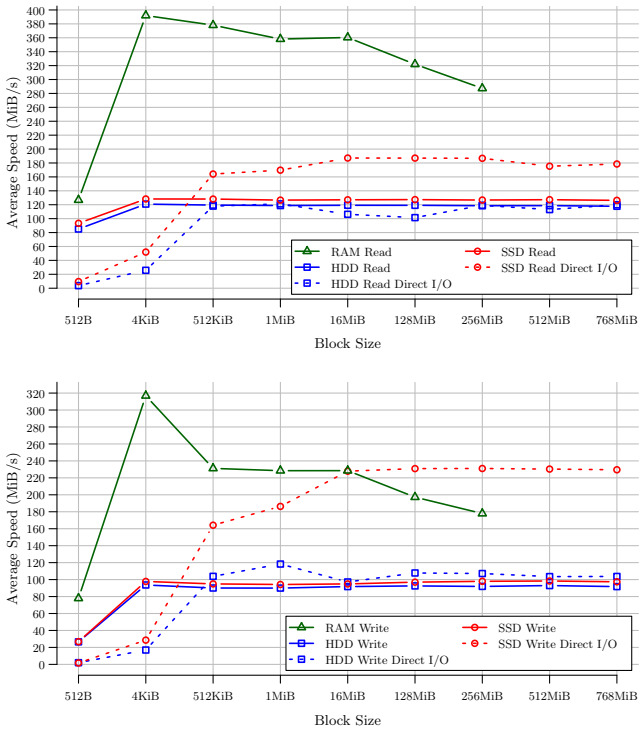


**Figure 2: Plots of average read and write speeds for each device and block size tested**

would allow for this higher speed. This could be confirmed by repeating the tests with a SATA SSD connected to the same RAID card as the HDD, instead of using a separate NVMe device.

When both reading and writing using standard I/O, RAM disk performance is far higher than both non-volatile storage devices. This was expected even when bottlenecks exist outside of the storage devices themselves, as the kernel optimises accesses to *tmpfs* file systems by avoiding the page cache, thus requiring fewer memory copy operations.

### 3.4.2 Impact of Direct I/O

When performing the same tests with direct I/O enabled, speeds to the storage devices are generally higher when the block size is sufficiently large to overcome the overheads involved, such as increased communication with hardware. 512B and 4KiB block sizes are slower than the standard write tests, as the kernel cannot cache data and write it to the device in larger blocks, but larger block sizes are faster.

It appears that the HDD is limited by other factors when block sizes of 512KiB and above are used, which may be due to the inefficiencies of AHCI, or simply the speed limitations of the disk itself. This is reinforced by the HDD direct I/O read speeds being approximately equal to, or lower than standard I/O speeds to the device, rather than seeing the performance increases of the SSD.

For the SSD, maximum direct write speeds are over double those of standard I/O, and maximum direct read speeds also show a significant improvement, however these are both still far lower than the rated speeds of the device. A further bottleneck appears to be encountered between 1MiB and 16MiB direct I/O block sizes, suggesting that at this point

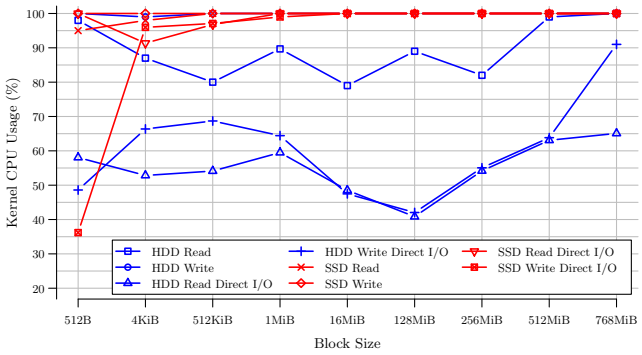


**Figure 3: Plot of average single-core kernel CPU usage for each block size tested**

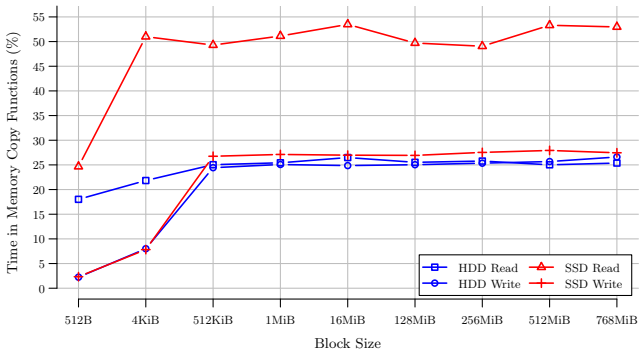


**Figure 4: Plot of proportion of time spent in memory copy functions for each block size tested**

the block size is large enough to overcome any communication and driver overheads and the earlier limitations experienced with non-direct I/O are once again affecting speeds. This speed limit (at around 230MiB/s) also matches the write speed limit of the RAM disk when using block sizes between 512KiB and 16 MiB, suggesting that both the SSD and RAM disk are experiencing the same bottleneck here.

### *3.4.3 CPU Usage*

Figure 3 shows the mean single-core kernel CPU usages across block sizes for each test, where single-core figures are calculated as the maximum of the two cores for each sample recorded. In general, it can be seen that a large amount of CPU time is spent in the kernel across the tests, with all but HDD direct I/O using an entire CPU core of processing for large block sizes, strongly suggesting that the bottlenecks implied by the speed results are caused by inadequate processing power.

The low system CPU usage of the HDD direct I/O tests suggests that the bottleneck may indeed be the disk itself, unlike the SSD tests, which show more clear, consistent limits in their transfer speeds.

For the SSD direct I/O write test, the 16MiB block size where the speed bottleneck begins corresponds to where CPU usage reaches 100%, further suggesting that the bottleneck is caused by processing on the CPU.

Further experimentation to test the direct impact that CPU speed has on the storage speeds could be carried out by repeating the tests while altering the clock speed of the ARM core, or limiting the number of CPU cores available to the operating system.

### *3.4.4 Profiling Results*

Results from profiling show that for both read and write tests, a large amount of CPU time is spent copying data between user and kernel areas of memory. Figure 4 shows the percentage of total execution time spent in the kernel functions `__copy_to_user` (for read) and `__copy_from_user` (for write), used for copying data to and from user space respectively. The direct I/O write tests spend no time in these functions, but instead a large amount of time is spent flushing the CPU data cache in the `v7_flush_kern_dcache_area` function.

Both read and direct I/O operations, which rely on more immediate access to storage devices, additionally spend a large amount of CPU time waiting for device locks to be released in the `_raw_spin_unlock_irq` function.

## 4. RELATED WORK

There are several areas of related work that suggest the current position of storage in a system architecture needs rethinking, due to the introduction of fast storage technologies, and due to inefficiencies in the software storage stack and file systems. Their focus is not entirely on embedded systems, but also on the increasing demand for efficient and fast storage in high-performance computing environments.

### *Refactor, reduce, recycle.*

The discussion in [10] advocates the necessary simplification of software storage stacks through refactoring and reduction, in order to make them able to fully utilise emerging high-speed non-volatile memory technologies, and lower the relative processor impact caused by fast I/O. It demonstrates that due to the large increase in storage speeds available with these devices, the traditional balance between slow storage and fast CPU speed is broken, and that improvements can be made through changes to the way storage is handled by the operating system. The work does not explicitly reference embedded systems, however the same theories apply in greater measure, due to even greater restrictions on processing resources.

### *Hardware file systems.*

Efforts such as [5] and [12] attempt to improve the storage performance in an embedded system by offloading certain file system operations to hardware accelerator cores on an FPGA. While the hardware file system implementation in [5] is quite limited and specialised in its operation, it is motivated by a similar need to optimise storage access beyond what was capable by the CPU in the target system. These hardware file system accelerators are aimed more towards usage in high-performance computing environments than stand-alone embedded systems.

## 5. DISCUSSION AND FURTHER WORK

The results presented in section 3 show that when CPU resources are sufficiently constrained, there are clear bottlenecks in storage operations, besides the access times of storage devices themselves. In order to utilise the full po-

tential of high-speed storage devices in an embedded Linux environment, and to avoid their use degrading the operation of other tasks running in the system, changes must be made to the storage stack to optimise how they are accessed.

## 5.1 Potential Solutions

There are several potential solutions to the problems covered, ranging from optimisations in existing software implementations to more radical system architecture changes.

### 5.1.1 VFS Optimisations

It may be possible to reduce storage overheads through restructuring the storage stack in Linux to better optimise it for high-speed storage with lower CPU usage. Results from profiling may be used to identify the areas of the storage stack that are performing particularly inefficiently, or that are simply unnecessary for the required tasks. Such optimisations would potentially require large changes to the structure of the Linux kernel.

### 5.1.2 Improved Direct I/O

The performance results show that using direct I/O can give a major boost to performance, especially with the fast SSD, if block sizes are above a reasonable threshold for disk access operations, but can also severely reduce performance if used for small block sizes.

Given its potential benefits, a reimplemented pseudo-direct I/O could operate with the benefits of direct I/O for large block sizes, but attempt to efficiently buffer storage device accesses when block sizes are below a practical limit. Standardising direct I/O so its operation can be guaranteed across file systems and kernel versions would also allow its usage to be more widely accepted.

### 5.1.3 Hardware Acceleration

One possible method of relieving CPU load during storage operations would be to introduce hardware acceleration into a system, in order to perform some of the tasks associated with the software storage stack in hardware instead. These accelerators could range from simple direct memory access (DMA) units, used to perform the expensive memory copy operations without taking up CPU time, or more-complex file-system-aware accelerators that access the storage device directly, effectively shifting the hardware/software divide further up the storage stack.

Introducing hardware that can access storage independently of the CPU may give an advantage for applications that use large streams of data, as more than just the storage device can be attached to the hardware. For example, a hardware accelerator could directly receive data from a hardware video encoder and write it straight to a file on persistent storage, with little CPU intervention and no buffering required in main system memory.

## 5.2 Further Work

There is potential for much deeper investigation into the operation of fast storage devices in an embedded Linux environment, in order to fully understand the bottlenecks involved and propose more comprehensive solutions.

While the results presented in section 3 highlight some examples of circumstances where storage speeds are heavily limited by areas other than storage devices themselves, they only focus on basic tests working with sequential data on a single file system type and clean devices. Further experiments will be carried out with other I/O patterns, such as random reads/writes, and benchmarks based on real-world usage patterns, in order to better gauge the scope of the issue and the focus for improvements.

Further work will also involve modifying areas of the test platform, such as the CPU clock speed and the number of available cores, in order to give insight on the direct affect this has on results. Alternative platforms, such as more-powerful server hardware, can be used to test exactly how much of a limiting effect the embedded hardware has on storage capabilities.

As well as experimental work on existing implementations, practical work to test the feasibility of solutions suggested above will be necessary in order to improve on the current situation. Modelling storage system operation based on experimental results may assist in implementation work, through the identification of areas that can be improved and giving a base on which to test solutions in a more abstract way.

# 6. REFERENCES


[1] open(2) – Linux Programmer's Manual. Release 4.02.

[2] Dstat: Versatile resource statistics tool, Mar. 2012. Online: `http://dag.wiee.rs/home-made/dstat/`.

[3] OProfile – A System Profiler for Linux, Aug. 2015. Online: `http://oprofile.sourceforge.net/`.

[4] A. M. Caulfield et al. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *Proc. 2010 ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage, and Analysis*, New Orleans, LA, Nov. 2010.

[5] A. Mendon. *The case for a Hardware Filesystem*. PhD thesis, University of North Carolina at Charlotte, NC, 2012.

[6] National Instruments. Data acquisition: I/O for embedded systems. White Paper, Oct. 2012. Available: `http://www.ni.com/white-paper/7021/en/`.

[7] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. 41st Int. Symp. Computer Architectures*, June 2014.

[8] R. Sass, W. Kritikos, A. Schmidt, S. Beeravolu, and P. Beeraka. Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing. In *Proc. 15th Annu. IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 2007.

[9] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads. In *Proc. 8th USENIX Conf. File and Storage Technologies*, Feb. 2010.

[10] S. Swanson and A. M. Caulfield. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage. *Computer*, 46(8):52–59, Aug. 2013.

[11] L. Torvalds. Re: O_DIRECT question. Linux Kernel Mailing List, Jan. 2007. Available: `https://lkml.org/lkml/2007/1/11/121`.

[12] V. Varadarajan, S. K. R, A. Nedunchezhian, and R. Parthasarathi. A reconfigurable hardware to accelerate directory search. In *Proc. IEEE Int. Conf. High Performance Computing*, Dec. 2009.