Efficient and Predictable High-Speed Storage Access for Real-Time Embedded Systems

Russell Joyce

EngD

University of York
Computer Science

September 2018

Abstract

As the speed, size, reliability and power efficiency of non-volatile storage media increases, and the data demands of many application domains grow, operating systems are being put under escalating pressure to provide high-speed access to storage. Traditional models of storage access assume devices to be slow, expecting plenty of slack time in which to process data between requests being serviced, and that all significant variations in timing will be down to the storage device itself. Modern high-speed storage devices break this assumption, causing storage applications to become processor-bound, rather than I/O-bound, in an increasing number of situations. This is especially an issue in real-time embedded systems, where limited processing resources and strict timing and predictability requirements amplify any issues caused by the complexity of the software storage stack.

This thesis explores the issues related to accessing high-speed storage from real-time embedded systems, providing a thorough analysis of storage operations based on metrics relevant to the area. From this analysis, a number of alternative storage architectures are proposed and explored, showing that a simpler, more direct path from applications to storage can have a positive impact on efficiency and predictability in such systems.

List of Contents

abstract 3
list of contents 5
list of tables 7
list of figures 9
acknowledgements 13
declaration 15
1 introduction 17
1.1 Introduction 17
1.2 Research Motivation 18
1.3 Historical Performance Trends 22
1.4 Research Hypothesis 25
1.5 Structure and Contributions 25
2 background and related work 29
2.1 Introduction 29
2.2 Overview of Technologies 29
2.3 Storage Access in Operating Systems 35
2.4 Storage Technologies and Interfaces 38
2.5 Impact of the Software Storage Stack 42
2.6 File System Functionality and Optimisation 46
2.7 Special-purpose File Systems 49
2.8 Hardware Acceleration 52
2.9 Measuring Storage Performance 58
2.10 Modelling Storage Systems 61
2.11 Measuring Efficiency and Predictability 62
2.12 Summary 65
3 analysis of storage operations in embedded linux systems 67
3.1 Introduction 67
3.2 Background and Motivation 68
3.3 Preliminary Profiling Tests 72
3.4 Measuring Storage Performance 75
3.5 Embedded System Storage Performance 84
3.6 Custom Low-overhead Profiling Component 97
3.7 Timing Periodic Storage Access 99
3.8 Low-level Storage Access Profiling 106
3.9 Simulated Real-world Benchmarks 118
3.10 Discussion and Future Work 124
3.11 Conclusion 126
4 alternative storage interfaces for embedded linux systems 129

4.1 Introduction 129
4.2 Background and Related Work 131
4.3 The CharIO Storage Interface 132
4.4 Evaluation of CharIO 140
4.5 User-space NVMe Driver for Embedded Linux 148
4.6 Evaluation of Embedded UNVMe 152
4.7 NVMe Coprocessor for Embedded Systems 163
4.8 Conclusions and Future Work 168
5 CONCLUSIONS AND FUTURE WORK 171
5.1 Summary and Contributions 171
5.2 Hardware Storage Architecture Design 173
5.3 Other Future Work 177
5.4 Conclusion 179

Appendices
A HISTORICAL TRENDS 183
A.1 General CPU Trends 183
A.2 Storage Interface Trends 185
A.3 FPGA Trends 186
A.4 Hennessy-Patterson CPU Trends 187
B EXPERIMENTAL PLATFORMS 189
B.1 x86-64 Server Platform 189
B.2 Zynq-7000 SoC Platform 190
B.3 Zynq UltraScale+ MPSoC Platform 192
B.4 Additional Hardware 194
C SOURCE CODE LISTINGS 195
C.1 File Copy for Profiling 195
C.2 UNVMe Full-disk Read/Write Benchmark 196
C.3 Filebench Benchmark Workloads 198
D EXPERIMENTAL DATA 217
D.1 Sample Output of *opreport* 217
D.2 Sample Per-symbol Output of *opreport* 218
D.3 Server Platform Filebench Outputs 219
D.4 Zynq-7000 Platform Filebench Outputs 235
E ADDITIONAL PLOTS 251
E.1 Zynq-7000 Platform CPU Frequency Scaling Results . . 251
E.2 10ms Periodic Task Results 254
E.3 Raw Device Periodic Task Results 259

REFERENCES 263

List of Tables

Table 3.1	Simulated real-world benchmark results on the server platform	122
Table 3.2	Simulated real-world benchmark results on the Zynq-7000 platform	123
Table 4.1	Comparison of storage access transfer speeds and CPU usages with CharIO	147
Table 4.2	Sequential transfer speeds for NVMe block device and Embedded UNVMe on Zynq UltraScale+	153
Table 4.3	Random transfer speeds for NVMe direct I/O block device and UNVMe driver on Zynq UltraScale+	154
Table 4.4	Sequential transfer speeds for Arm Embedded UNVMe driver and MicroBlaze NVMe driver on Zynq UltraScale+	166
Table 4.5	Comparison of proposed storage interfaces	169

List of Figures

Figure 1.1 Relative single- and multi-core CPU, and memory performance increases since 1980 23
Figure 1.2 CPU clock speeds over time 23
Figure 1.3 Memory interface speed progression over time 24
Figure 1.4 Storage interface speed progression, relative to 1980 24
Figure 2.1 Position of the Linux VFS as an interface between applications and concrete file systems 37
Figure 2.2 Potential areas of storage stack, grouped by high-level domain 62
Figure 3.1 Basic control flow of standard file operations in Linux 69
Figure 3.2 The operation of direct I/O in the Linux kernel, bypassing the page cache 71
Figure 3.3 Full call graph of file copy test 73
Figure 3.4 Limited call graph of file copy test 74
Figure 3.5 Read speeds for HDD, SSD and NVMe SSD on the server platform 77
Figure 3.6 Write speeds for HDD, SSD and NVMe SSD on the server platform 78
Figure 3.7 Kernel CPU utilisation for HDD, SSD and NVMe SSD read operations on the server platform . . 80
Figure 3.8 Kernel CPU utilisation for HDD, SSD and NVMe SSD write operations on the server platform . 81
Figure 3.9 Read speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the server platform 82
Figure 3.10 Write speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the server platform 83
Figure 3.11 Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz 85
Figure 3.12 Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz 86
Figure 3.13 Kernel CPU utilisation for HDD, SSD and NVMe SSD read operations on the Zynq-7000 platform at 800 MHz 89
Figure 3.14 Kernel CPU utilisation for HDD, SSD and NVMe SSD write operations on the Zynq-7000 platform at 800 MHz 90

Figure 3.15 Read speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz 91
Figure 3.16 Write speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz 92
Figure 3.17 Proportion of time spent in memory copy functions for HDD and NVMe SSD with each block size tested 94
Figure 3.18 Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at different frequencies 95
Figure 3.19 Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at different frequencies 96
Figure 3.20 Example output of the profiling timer 98
Figure 3.21 Timing of periodic high-load, low-priority, synchronous block device writes 100
Figure 3.22 Periodic block device synchronous reads and write speeds 102
Figure 3.23 Periodic ext4 file system synchronous reads and write speeds 103
Figure 3.24 Periodic block device asynchronous reads and write speeds 104
Figure 3.25 Periodic ext4 file system asynchronous reads and write speeds 105
Figure 3.26 Profiling results for 4 KiB and 8 KiB reads from ext4 and NVMe block device 108
Figure 3.27 Profiling results for 4 KiB and 8 KiB writes to ext4 and NVMe block device 109
Figure 3.28 Profiling results for 40 KiB and 80 KiB reads from ext4 and NVMe block device 111
Figure 3.29 Profiling results for 40 KiB and 80 KiB writes to ext4 and NVMe block device 112
Figure 3.30 PDF of 4 KiB and 8 KiB read timings from ext4 and NVMe block device 113
Figure 3.31 PDF of 4 KiB and 8 KiB write timings to ext4 and NVMe block device 114
Figure 3.32 PDF of 40 KiB and 80 KiB read timings from ext4 and NVMe block device 115
Figure 3.33 PDF of 40 KiB and 80 KiB write timings to ext4 and NVMe block device 116
Figure 3.34 Average profiling results for 40 KiB and 80 KiB reads from ext4 and NVMe block device 117
Figure 4.1 Storage access layers in Linux 132
Figure 4.2 Storage access layers using CharIO driver . . . 133
Figure 4.3 CharIO Linux kernel module structure 135

Figure 4.4 Operating system and hardware latency measured for 40KiB and 80KiB reads from ext4, block device and CharIO storage 142
Figure 4.5 Box plot of interrupt counts across 1,000 512 MiB reads 143
Figure 4.6 Box plot of interrupt counts across 1,000 512 MiB writes 143
Figure 4.7 PDF of 4 KiB, 8 KiB, 40 KiB and 80 KiB read timings from CharIO device 145
Figure 4.8 PDF of 4 KiB, 8 KiB, 40 KiB and 80 KiB write timings from CharIO device 146
Figure 4.9 Architecture of the Embedded UNVMe driver 150
Figure 4.10 Read speeds for NVMe block device and UNVMe on Zynq UltraScale+ platform 155
Figure 4.11 Write speeds for NVMe block device and UNVMe on Zynq UltraScale+ platform 156
Figure 4.12 CPU utilisation for NVMe block device and UNVMe read operations on Zynq UltraScale+ platform 157
Figure 4.13 CPU utilisation for NVMe block device and UNVMe write operations on Zynq UltraScale+ platform 158
Figure 4.14 Read speed per percent of CPU utilisation for NVMe block device and UNVMe on Zynq UltraScale+ platform 159
Figure 4.15 Write speed per percent of CPU utilisation for NVMe block device and UNVMe on Zynq UltraScale+ platform 160
Figure 4.16 Architecture of the MicroBlaze NVMe driver . 164
Figure 5.1 High-level view of persistent storage as a first-class member of a system's memory map . . . 174
Figure 5.2 Proposed hardware storage architecture design 175
Figure A.1 CPU Dhrystone MIPS over time 183
Figure A.2 CPU core counts over time 184
Figure A.3 Storage interface speed progression over time 185
Figure A.4 FPGA transistor count progression over time . 186
Figure A.5 FPGA programmable logic size progression over time 186
Figure A.6 Relative CPU performance increase since 1978 187
Figure B.1 Memory performance of x86-64 server 189
Figure B.2 System architecture of the Zynq-7000 SoC test set-up 190
Figure B.3 Memory performance of Zynq Mini-ITX development board 192
Figure B.4 System architecture of the Zynq UltraScale+ MPSoC test set-up 193

Figure E.1 Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 256 MiB block size at different frequencies 251
Figure E.2 Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 512 KiB block size at different frequencies 252
Figure E.3 Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 512 KiB block size at different frequencies 252
Figure E.4 Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 16 MiB block size at different frequencies 253
Figure E.5 10ms periodic synchronous read speeds 255
Figure E.6 10ms periodic synchronous write speeds . . . 256
Figure E.7 10ms periodic asynchronous read speeds . . . 257
Figure E.8 10ms periodic asynchronous write speeds . . . 258
Figure E.9 Raw device periodic read speeds 260
Figure E.10 Raw device periodic write speeds 261

Acknowledgements

Firstly, I would like to thank Neil Audsley for his patient supervision over many years, as well as giving me the opportunity and encouragement to complete this doctorate and the beginnings of a career in research. Additionally, Leandro Soares Indrusiak for co-supervising in the early years, and always being available for support throughout, and Scott Hansen and The Open Group for agreeing to act as my industrial supervisor and sponsoring organisation, without whom this EngD would not have been possible. I would also like to thank my examiners, Robert Davis and Peter Pietzuch, whose thorough feedback has undoubtedly made this thesis a vastly better document.

The experience and knowledge I have gained in the pursuit of my research would not have been half as rich without my friends and colleagues in the Department of Computer Science, but most importantly those I shared an office with during the formative years of my EngD – Dr. Ian Gray, Jamie Garside, Gary Plumbridge and Gareth Lloyd. This thesis might have been completed sooner if it were not for the influence of CSE/120, but the gain was definitely worthwhile.

The number of other people who have helped and supported me along the way are too numerous to enumerate, but thank you to you all. Special mentions are certainly deserved by José for the many much-needed tea breaks, Imran for the sambuquila, Chris and Jim for providing support across a number of time zones, Sam for our unspoken pact to both drag the EngD process out for as long as possible, and Alan Millard for providing the ultimate motivation through his constant insistence that I was going to fail.

Sincere thanks also go to my family for (maybe sometimes reluctantly) embracing my decision to stay at university seemingly indefinitely, and to my partner, Mădălina, for just being the best.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

The following publications have arisen from work presented in this thesis (note that [1] and [2] have the same content):

[1] R. Joyce and N. Audsley, 'Exploring storage bottlenecks in Linux-based embedded systems', in *Proc. 5th Embedded Operating Systems Workshop*, Amsterdam, Netherlands, Oct. 2015.

[2] R. Joyce and N. Audsley, 'Exploring storage bottlenecks in Linux-based embedded systems', *ACM SIGBED Review*, vol. 13, no. 1, pp. 54–59, Jan. 2016.

[3] R. Joyce and N. Audsley, 'Improving efficiency of persistent storage access in embedded Linux', in *Proc. 10th York Doctoral Symposium on Computer Science and Electronic Engineering*, York, UK, Nov. 2017.

The following software publications have additionally arisen from work presented in this thesis:

[4] R. Joyce, *RTSYork/profiling-timer: First release of profiling timer component*, Sep. 2018. DOI: 10.5281/zenodo.1436577. [Online]. Available: https://doi.org/10.5281/zenodo.1436577.

[5] R. Joyce, *RTSYork/linux-4.1.15-rt17-profiling: First release of Linux profiling modifications*, Sep. 2018. DOI: 10.5281/zenodo.1436031. [Online]. Available: https://doi.org/10.5281/zenodo.1436031.

[6] R. Joyce, *RTSYork/chario-module: First release of CharIO kernel module*, Sep. 2018. DOI: 10.5281/zenodo.1436029. [Online]. Available: https://doi.org/10.5281/zenodo.1436029.

[7] R. Joyce, *RTSYork/chario-tests: First release of CharIO test code*, Sep. 2018. DOI: 10.5281/zenodo.1436027. [Online]. Available: https://doi.org/10.5281/zenodo.1436027.

[8] R. Joyce and D. Hong, *RTSYork/unvme-zynq: First release of UNVMe port to Zynq MPSoC*, Sep. 2018. DOI: 10.5281/zenodo.1435241. [Online]. Available: https://doi.org/10.5281/zenodo.1435241.

[9] R. Joyce, *RTSYork/nvme-microblaze: First release of NVMe driver for MicroBlaze*, Sep. 2018. DOI: 10.5281/zenodo.1435246. [Online]. Available: https://doi.org/10.5281/zenodo.1435246.

[10] R. Joyce *et al.*, *RTSYork/filebench: filebench-1.5-alpha3+4kdirect*, Jan. 2020. DOI: 10.5281/zenodo.3598521. [Online]. Available: https://doi.org/10.5281/zenodo.3598521.

1

Introduction

This chapter provides an overall introduction to the thesis, outlining the general research direction and motivation. A research hypothesis is also presented, along with a list of contributions, and the structure for the remainder of the document.

1.1 Introduction

Persistent secondary storage is an integral part of almost all modern computer systems, with advances in Hard Disk Drives (HDDs), Solid-State Drives (SSDs), and other Non-Volatile Memory (NVM) technologies constantly pushing the boundaries of the size and speed of data available to a system. Data stored on these devices typically relies on abstractions provided by a file system for access, allowing applications to read from and write to storage in a structured way that separates a logical collection of data (such as a specific file) from its physical representation on the medium (such as a set of sectors on a hard disk). Below file-level abstractions, most storage devices are treated as block-level memories, allowing read and write operations only on relatively large portions of data (typically 512 or 4096 bytes). This is in contrast to the byte- or word-level access granularity of primary system memory (potentially via a cache fetching data from RAM in line lengths), but was historically a necessary trade-off for persistent storage in order to provide efficient access to large or slow devices. Emerging types of non-volatile memory are beginning to allow byte-addressable access to persistent storage, however these technologies are currently limited to experimental architectures and are not widely available [11].

Traditionally, persistent storage media have been many orders of magnitude slower than main system memory, meaning file systems and the operations around them were not required to operate at a particularly high speed. This has led to issues of storage technology speeds overtaking the abilities of Central Processing Units (CPUs) and interfaces, especially when dealing with embedded systems that have

limited resources, or real-time large-scale data problems, which require the reliable storage and retrieval of data at high bandwidths [12].

Some efforts are being made by hardware manufacturers at the operating system and interface levels to address the issue of high CPU usage caused by high-speed storage, with the introduction of technologies such as NVM Express (NVMe) [13] that are focused on providing more optimised access to storage. Currently the majority of secondary storage devices are connected through a Serial ATA (SATA) interface, however, and accessed using the Advanced Host Controller Interface (AHCI) standard, whose development was originally focused on relatively slow mechanical hard disks, and is therefore not efficiently implemented for high-speed storage [14]. While the move to more efficient protocols relieves some of the software load associated with fast access to storage, the effect may still be limited if storage speeds continue to increase relative to other hardware, potentially requiring a more fundamental change to the layers of a storage system present in an operating system kernel, or split between software and hardware.

On top of this, the use of large, high-speed persistent storage in embedded systems is becoming increasingly both practical and necessary, due to the lower power requirements and greater robustness of solid state storage compared to mechanical media, and the handling of large volumes of data that cannot reasonably be stored in volatile system memory or embedded flash storage.

The increases in speed and consistency of storage access times brought about by modern persistent storage technologies, combined with the expanding storage demands of many embedded and real-time applications provides the overall motivation for this research examining more efficient, predictable and appropriate storage access systems for real-time embedded systems.

The real-time implications of a more efficient and predictable storage architecture motivate a stronger look into real-time properties of a potential system, and not just any possible raw performance improvements. This includes considering the priority and accountability of a task that is performing storage operations, investigating ways that time spent blocking on a storage operation can correctly be attributed to a task, and considering any side-effects that storage operations may have across multiple real-time tasks.

1.2 Research Motivation

The core motivation for this research is derived from the need to address potential operating system shortcomings in accessing high-speed storage devices in the context of real-time and embedded systems. Currently, emerging high-speed storage technologies are pushing the boundaries of software file system processing, with CPU require-

ments being very high in order to saturate the fastest interfaces and devices [12]. Similar issues are experienced by many large-scale data systems, in areas where storage is by far the most intensive resource in a system.

A potential lack of real-time predictability is combined with the speed impact caused by the CPU pressure of high-speed storage, as other processes running on a system have a high possibility of impacting the speed of data storage, and the inherent complexity of storage architectures causing the CPU load leads to access patterns that are very difficult to predict. Predictability is essential in many large-scale data applications, as a system often must be able to operate at a guaranteed bandwidth in order to process and store incoming data reliably.

One method that has potential to address these issues is hardware acceleration – reducing the amount of the storage stack executing in software by offloading a number of layers into dedicated hardware. This may reduce the pressure on the CPU, allowing for increased storage speeds to be properly utilised, and improving the predictability of storage operations. A less dramatic approach is to simplify the software side of the storage stack while retaining the same hardware support, collapsing and removing a number of potentially unnecessary layers on the path between applications and storage.

1.2.1 Storage for Embedded Systems

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 [12]. Additionally, mechanical hard disk drives are fundamentally limited in their latency and Input/output Operations Per Second (IOPS) by physical constraints, meaning even a high-end 15,000 RPM device can have a 2.7 ms seek time and achieve just 210 IOPS [15], compared with latencies in the order of microseconds and IOPS in the order of tens to hundreds of thousands from SSDs [16, 17], which do not experience the same inherent issues.

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 a general-purpose operating system such as Linux perfectly adequately for many tasks, it may not be able to take advantage of the full speed of the SSD using traditional methods of storage access in the operating 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.

1.2.2 Real-Time Storage Access

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. Standard methods of accessing storage through a general-purpose operating system kernel are also unlikely to consider real-time requirements and the predictability of operations.

Stand-alone embedded systems that use storage devices create motivation for efficient, real-time access to storage, for applications such as logging sensor data and recording high-bandwidth video streams [18], where information will be lost if it cannot reliably be stored in a timely manner. 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 method that can help to meet these requirements is desirable, especially while taking into account other potential requirements of a system, such as minimising energy usage.

If CPU time can be removed as far as possible from the operation of copying data to storage, the impact on this from other, non-storage processes will be reduced, potentially 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. Explicit modification of the storage stack to allow for real-time accountability and to reduce variability would also make real-time storage access a more reasonable task.

1.2.3 Large-Scale Data Systems

While not strictly an embedded systems issue, an ongoing problem in computing is that of storing and processing large-scale or 'big' data – sets of data that are too large to be processed by traditional methods. For example, while a typical method of operating on data would be to load files from a secondary storage device into main memory, perform some process on these files, then save the result back to disk, the size of large-scale data sets means this is not possible. Typically, data is stored over many servers and streamed to applications in real-time as it is produced or used, exhibiting some of the same requirements as real-time embedded systems, such as high efficiency and predictability.

A prime example of a large-scale data problem is the results produced by CERN's Large Hadron Collider, whose experiments yield around 50-70 petabytes per year (over 1.5 gigabytes every second) to be stored for later analysis, even after filtering out and discarding 99% of results data in real-time [19]. In order to store and retrieve this amount of data at a practical speed specialist systems need to be used, as traditional, general-purpose file systems and databases create overheads and perform many operations in a manner that is not efficient for such large data sets [20]. This heavy reliance on storage leads to the storage architecture becoming a limiting factor in the system, thanks to the disproportionate pressure it puts on other areas.

Due to their low power usage, high performance and high reliability, embedded accelerators are increasingly being used to assist with large-scale data applications [21]. Embedded systems often have limited resources, especially in areas such as CPU speed, meaning hardware cores implemented in programmable logic, rather than software implementations of functionality, are used for many tasks. Currently, direct access to storage from hardware accelerators is not widely implemented, with most systems requiring a software file system to service storage requests. This presents limitations in storage access speeds, and removes some of the potential real-time predictability of the accelerator, due to limited embedded CPU speeds throttling operation [22].

1.3 Historical Performance Trends

This section presents historical trends in multiple areas of computer systems related to the research presented in this thesis, including CPU and memory speeds, storage interface bandwidths, and programmable hardware capabilities, in order to contextualise current discrepancies between high-speed storage and processing improvements, and provide motivation for the research. Data was collected from a number of sources to quantify improvements in various computing resources over recent history, with a focus on their relationship to storage technologies.

The following areas are identified to provide relevant data for comparison of the advancement of system components:

- Single- and multi-core CPU speed
- Volatile memory bandwidth
- General storage performance
- Storage device and interface bandwidths
- FPGA size and capabilities

Data is gathered from various sources, including books, online collections of benchmarking data, product datasheets, and technical standards. In some cases, the breadth and quality of readily available data has proven too limited for thorough conclusions to be drawn, with information from many benchmarks often not providing enough context to be reliably useful. Data from published standards and literature is more useful in providing applicable information, however this still provides limited context around how much of potential maximum values are used in real-world contexts, and disparities between publication dates and when technologies are actually commonly in use.

In general, results show trends that are to be expected, with all areas showing reasonably steady improvement over time. The main results worth noting arise from the progression of storage interfaces, whose speeds are being increasingly driven by the capabilities of storage devices, while single-core CPU speeds are failing to progress at rates that were previously taken for granted. This difference between CPU and storage speed growth provides a basic level of motivation for further exploration of storage optimisation, as the traditional assumption in operating systems design that CPU speeds far outperform storage technologies is clearly beginning to break.

1.3.1 Analysis of Historical Trends

Figure 1.1 shows the 'memory gap', proposed by Hennessy and Patterson [23] where CPU speeds and volatile memory have progressed at quite different rates since the early days of mainstream computers in

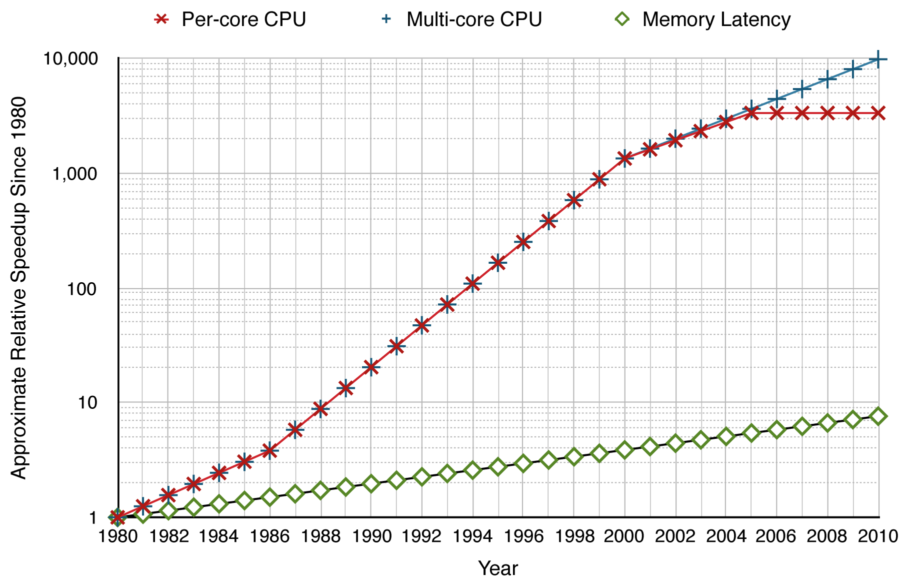


Figure 1.1: Relative single- and multi-core CPU, and memory performance increases since 1980, adapted from [23]

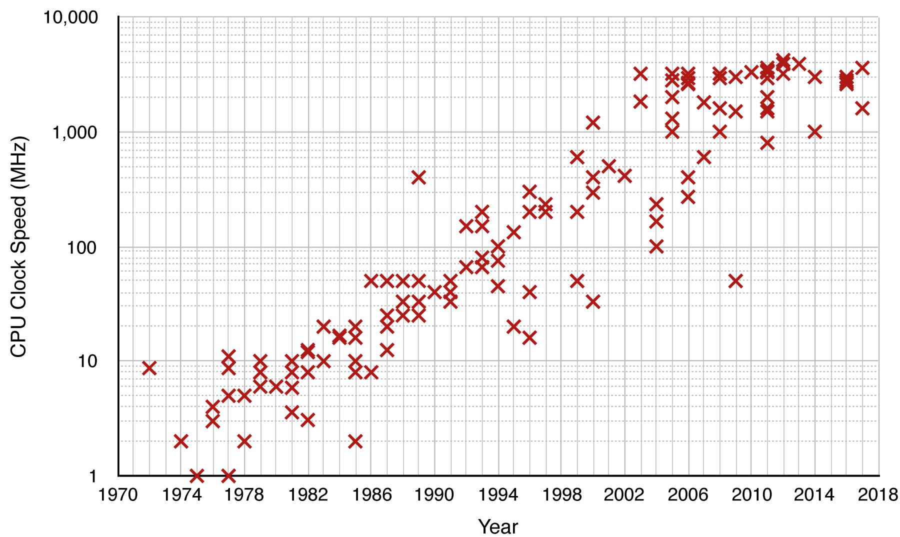


Figure 1.2: CPU clock speeds over time, data from [24]

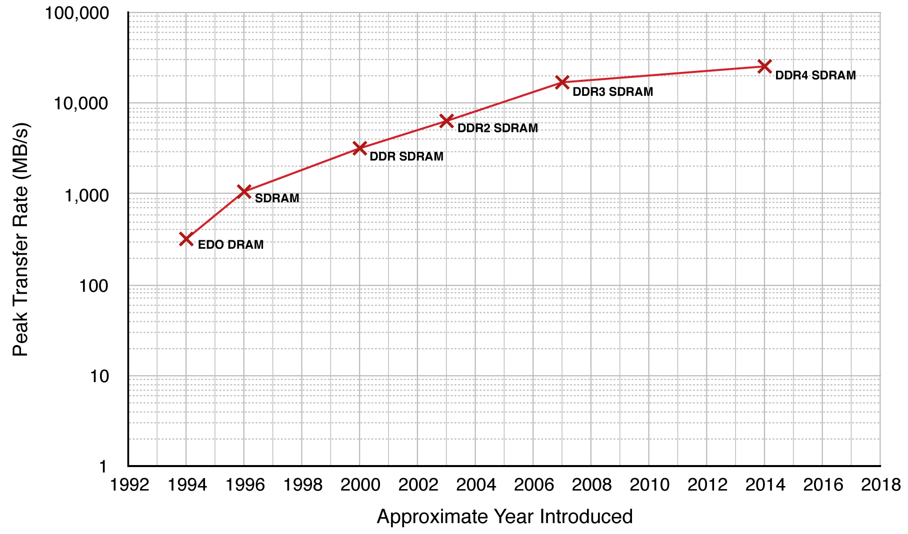


Figure 1.3: Memory interface speed progression over time, data from [23, 25–28]

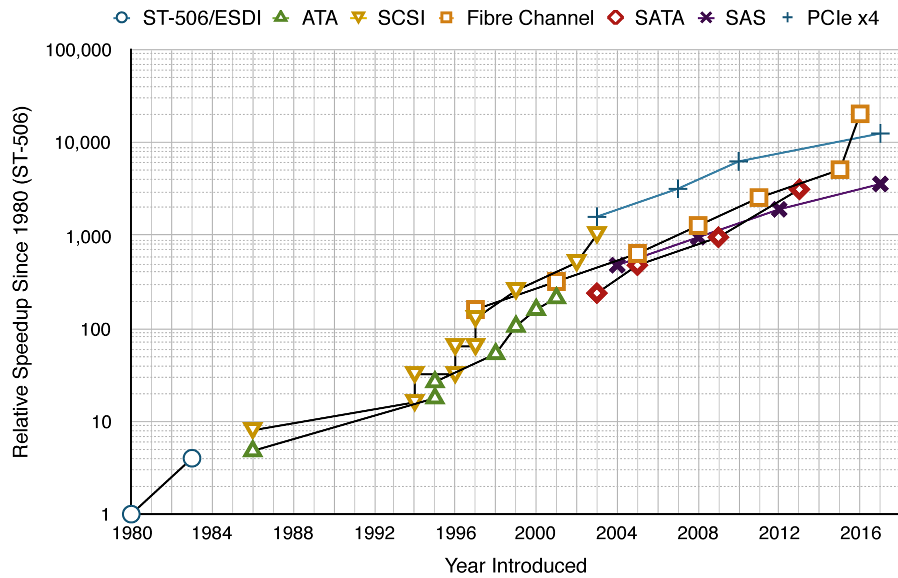


Figure 1.4: Storage interface speed progression, relative to 1980, data from [29–34]

1980, with CPU frequency improvements dwarfing those of SDRAM. Since around 2005, however, single-core CPU speed scaling has stalled, with more modern processors relying on parallel, multi-core architectures for performance improvements.

The limits of CPU frequency scaling are reinforced by Figure 1.2, which displays the real-world frequencies of various CPUs released between the early 1970s and 2018 [24]. The point at which the upward trend of CPU frequency speeds stopped can clearly be seen in the plotted devices.

While not as sudden as CPU speed scaling shifts, Figure 1.3 shows a general trend of memory speed increases slowing too, according to various memory standards [23, 25–28], with each new interface providing a smaller performance increase than the last, and being released after a greater period of time.

In contrast to these trends, Figure 1.4 shows a far more constant increase in maximum storage interface speeds over the last forty years, according to various storage standards [29–34], suggesting that this area is not suffering from the same limitations as CPU and memory scaling. This strongly motivates a need to adjust the assumptions used by software systems when accessing storage, not just in embedded systems, in order to address the changing relative performance of storage against the rest of a system.

Additional plots compiled using data collected from multiple sources can be found in Appendix A, including CPU performance and core counts, storage interface speeds, and FPGA device trends.

1.4 Research Hypothesis

Traditional methods of accessing high-speed storage are problematic for real-time embedded systems, particularly due to their complexity, potentially causing high CPU demands and large timing variability. To this end, this thesis addresses the following hypothesis:

> Reducing the complexity of the software storage stack in real-time embedded systems results in more predictable and efficient access to storage.

To expand on this, efficiency is defined as reducing the execution time required for a given size of storage access, while predictability covers both the variability of this execution time, and the ability to predetermine and guarantee the control flow and pattern of a storage access operation.

1.5 Structure and Contributions

The remainder of this thesis is structured as follows.

Chapter 2 covers a wide range of related literature in the areas touched by the thesis, as well as providing background on storage access in real-time embedded systems, detailing any technologies used, and suggesting models and metrics for the effective measurement of results. The main technical contribution of this thesis begins in Chapter 3, with a detailed analysis of storage systems in embedded Linux, with a focus on efficiency and predictability when dealing with large data sets. This is followed in Chapter 4 with the development and discussion of a number of possible alternative storage stack implementations, with the aim of improving the key identified metrics, along with thorough evaluation in comparison to standard storage access methods. Chapter 5 then proposes a number of future directions for the work presented, including a proposal for a novel hardware storage architecture design, based on the motivation and ideas in the thesis and any limitations discovered, before finally concluding the thesis.

1.5.1 Research Contributions

The following research contributions are offered by this thesis in Chapters 2 to 4.

Chapter 2: Background and Related Work

Chapter 2 provides a comprehensive overview of literature related to storage in real-time embedded systems, highlighting issues related to the area, outlining work that has been carried out to address these issues, and identifying limitations of existing approaches to improvements. The chapter also provides a general background in areas covered by the thesis, including the evolution of persistent storage, operating systems, and hardware acceleration. A high-level model of storage access is defined, categorising potential operations provided by a typical storage stack into more general domains, along with a series of metrics for consistent measurement of storage systems. Ideas and analysis from background work and related literature inform and motivate the remainder of the thesis.

Chapter 3: Problem Analysis

Chapter 3 provides a large amount of analysis on storage access in Linux, specifically in how it relates to the predictability and efficiency required for real-time and embedded systems. Initial profiling work leads to a thorough benchmarking of standard and direct I/O access to a number of storage devices in an embedded platform, examining both the speed and CPU usage to give an indication of efficiency. This is followed by the presentation of a custom hardware component for low-overhead timing of software and hardware triggers, which is

used to measure the variability of periodic storage accessed, as well as accurate low-level profiling of storage access to expand on these results. The results from this chapter serve to both inform on the differences between current storage access methods in Linux, and to further motivate the need for alternatives in order to improve results based on the desired metrics.

Chapter 4: Alternative Storage Interfaces

Chapter 4 presents a number of alternative storage interfaces for Linux, largely focussing on predictability and efficiency over unnecessary convenience or complex features. CharIO, a kernel driver presenting SSD storage as as character device, is presented and evaluated, showing promise for simplifying the kernel storage stack. This is followed by removing kernel interference to a greater level, through a port of the UNVMe driver to the Zynq UltraScale+ embedded platform, and a complete NVMe driver implementation running on a MicroBlaze soft processor core separate to the main CPU.

Chapter 5: Future Work

Chapter 5 discusses a number of avenues for future work, including a proposed design for a more ambitious restructuring of the storage stack, using custom hardware components to create and manage a simple memory-mapped interface for software to access storage.

2

Background and Related Work

This chapter of the thesis covers relevant background information, and discusses and reviews existing work in topics related to the research, including the fields of operating systems, storage technologies, file systems, and hardware acceleration.

2.1 Introduction

The areas of work related to this thesis cover a broad range of topics, including research into operating systems design, storage technologies, storage and device interfaces, file systems, hardware acceleration, and the wider areas of real-time and embedded systems. Due to the highly practical engineering side of these areas (and to some extent, of the engineering doctorate), a number of sources outside of traditional research articles must also be considered in order to gain a full understanding of the state of the art.

The technical aspects of all the areas related to this thesis are in some places quite distinct, leading to this chapter presenting a mixture of literature review and a more technical introduction to background topics for the thesis, and providing a contribution in both areas.

In order to effectively investigate the hypothesis of this thesis, the concepts that apply to storage in real-time and embedded contexts must be defined, along with suitable methods of measuring any improvements. To this end, Section 2.10 presents a high-level model of storage that can be selectively applied to many systems, defining common terms for use throughout the research. A comprehensive set of metrics are also outlined in Section 2.11, describing how the measurement of both predictability and efficiency, in terms of raw performance and system utilisation, can be quantified.

2.2 Overview of Technologies

Various technologies have been investigated in the areas of storage devices, interfaces, programmable hardware and operating systems,

in order to determine their feasibility for use in this research project. These technologies are outlined here in order to provide appropriate background and context for the thesis, and to motivate their use for the research.

2.2.1 Operating System Choice

In order to focus the research in a way that makes the implementation of ideas possible, theoretical concepts are related to a main practical target operating system. The Linux operating system is chosen as a suitable target for implementation work in Chapters 3 and 4 for several reasons.

Firstly, the Linux kernel, along with much of the software that builds upon it, is free, open-source software, so its code is readily available and modifiable for research purposes with no special access or licences required.

Secondly, the Linux kernel supports a large range of architectures and devices, ranging from large-scale high-performance computing systems to low-powered embedded devices. This means a large range of devices may be targeted through the same knowledge and with common code; for example, the same basic kernel code (with the exception of architecture-specific areas) will run on a small MicroBlaze soft processor core on an Field-Programmable Gate Array (FPGA), an embedded Arm processor, and a powerful x86-64 server.

Thirdly, real-time extensions to the Linux kernel are available if required, so research into storage systems may be carried out on Linux in a real-time context.

Finally, Linux is a very widely-adopted operating system that is actively developed in many areas, with a large amount of support and expertise available. Its wide-scale use also means that many device drivers are available for it, allowing a large number of peripherals and platforms, both old and new, to be used natively.

2.2.2 Hardware Acceleration

In order for an embedded application accelerator to perform file operations entirely in hardware, it would require knowledge of the file structure on the physical disk. To facilitate this, control of the file system itself may be transferred to the accelerator, allowing it to have complete control over the storage, or the file system could be discarded entirely if it is not necessary for the application. The idea of moving file system functionality into hardware has been investigated to varying degrees in the past, through both simulation and implementation, including for enabling direct storage access from embedded devices [22, 35], to remove reliance on operating system compatibility with the file system of removable storage [36], in efforts

to improve performance of specific file system operations [37], and in wider projects to enable operating system-like access to system devices directly from FPGAs [38].

The potential of read and write speeds to be significantly improved through hardware acceleration, particularly when using fast solid-state storage, is shown by one hardware file system implementation, achieving around double the speed of *ext2* running on a server platform, and up to a 16x speed-up compared to an embedded MicroBlaze CPU [22]. A potential use for an accelerated file system is also briefly discussed in the EU-funded JUNIPER project [39], for the assistance of FPGA accelerators used in real-time big data applications.

As a comparison to this, an example of another software function of an operating system that has greatly improved over time due to increasingly advanced hardware accelerators is graphics processing. Originally, the majority of computers performed all graphics processing in software routines on the CPU, but as technology improved and demand for better graphics performance increased, it was necessary to offload graphics functionality onto hardware accelerators, beginning with more limited Application Specific Integrated Circuit (ASIC)-based solutions, and developing into modern Graphics Processing Units (GPUs), which include programmable architectures to support a wide range of operations running alongside the CPU [40]. While storage tasks involve many aspects that are different to graphics processing tasks, both are data- and computation-intensive processes, and the increase in speeds of physical storage hardware and the increasing demands placed on storage systems show some parallels to the history of graphics processing, and may also benefit from a degree of hardware acceleration.

2.2.3 Programmable Hardware and Systems

When developing an embedded system with the potential for hardware acceleration, a key part of this is the hardware on which applications are run and accelerator cores are implemented. For a research project, an FPGA is a good choice for developing hardware designs, as they offer good functionality and performance, allow for a relatively fast development cycle, and are readily available on development boards with an array of auxiliary peripherals and Input/Output (I/O) interfaces. Within this thesis, research focuses on FPGAs and tools from Xilinx, however similar devices and software is available from other manufacturers, such as Intel.

Xilinx Zynq System-on-Chips

The Zynq-7000 [41] and Zynq UltraScale+ [42] series of devices from Xilinx offer FPGA programmable logic alongside multi-core Arm microprocessors and a variety of supporting peripherals on a single

System-on-Chip (SoC). These devices allow for a full Linux operating system to be installed on a capable embedded processor, while being closely attached to programmable logic that may be dynamically reconfigured with hardware accelerators and interfaces. Interconnect fabric between the processor cores and programmable logic allows the Linux kernel to access custom coprocessors for hardware acceleration as if they were built into the chip as standard components.

Using a Zynq-based system for experimentation allows for a wide range of storage peripherals to be tested while having a tight coupling with hardware accelerators. Development platforms are available, such as the Avnet ZedBoard Mini-ITX development board [43] and Xilinx ZCU102 [44], that allow for the connection of storage devices over SATA, PCI Express (PCIe) and Universal Serial Bus (USB) interfaces. A set-up using a board like this allows for hardware cores to be developed that interface secondary storage with a Linux kernel running on the Arm CPU, as well as additional logic for creating custom supporting peripherals for experiments, accelerating aspects of the storage stack, and bringing functionality into the hardware. The large variety of interfaces allows for testing and benchmarking of different storage devices in a controlled set-up, in order to measure the effect this has on current file systems along with the solutions developed. A number of high-speed networking interfaces are also available to the Arm cores and programmable logic, allowing for potential set-ups involving storing and retrieving streaming network data to and from a storage device.

Soft Processor Cores

As an alternative to a mixed hard-CPU and soft-logic SoC platform, a similar set-up using a large FPGA (such as a Xilinx Virtex-7 [45]) and one or multiple MicroBlaze soft processor cores programmed onto the logic could achieve similar results, however the Arm cores on the Zynq platforms allow for a much more capable Linux system to be used, and the ability to fully reprogram the FPGA logic without halting the operating system and other software.

Host Server with FPGA Accelerators

A further alternative to a SoC-based FPGA platform would be to use a standard x86 server running a host operating system, which interfaces with one or more FPGA boards using PCIe. Applications on the server would then communicate directly with various accelerators through this interface, however this moves further from the realm of embedded systems that is most targeted by this thesis.

2.2.4 Embedded FPGA Storage Interfaces

The two main physical interfaces used to connect secondary storage to a modern computer system are SATA and PCIe, both of which can be interfaced with an FPGA when appropriate hardware is used. Some more recent embedded SoCs, such as the Zynq UltraScale+, contain dedicated controller peripherals for SATA and PCIe. Alternative interfaces such as Serial Attached SCSI (SAS) or SATA Express, a more recent interface designed to operate as both SATA and PCIe in a single form factor, can also be used to connect storage but are less common and currently less well supported from FPGA designs. Legacy parallel storage interfaces such as Parallel ATA (PATA) and SCSI Parallel Interface (SPI) are not considered in this thesis, due to their low speeds and limited support with modern hardware, despite being simpler to operate.

While SATA is an interface explicitly designed for the connection of storage devices, PCIe is more generic, requiring a secondary interface for actual communication with a drive. This secondary interface may be SATA, in either an AHCI or Redundant Array of Independent Disks (RAID) configuration, a specialised standard such as NVMe, or a proprietary system, and this may be controlled in either software or hardware.

SATA Core Implementations

As a result of the high-speed serial transceivers available in many modern FPGAs, it is possible to directly interface with SATA devices from programmable logic through the use of a Host Bus Adapter (HBA) IP core. This core is responsible for setting up and controlling the transceivers, managing the encoding and decoding of data that is sent over the physical interface, presenting a control interface to the rest of the FPGA system, and often includes memory interfaces for fast Direct Memory Access (DMA) transfer of data.

Several commercial SATA HBA cores are available for use on FPGAs [46–48], however these can be prohibitively expensive, and do not offer the ability for heavy customisation as they are delivered as 'black box' designs without underlying source code. Due to these restrictions, there exists to some extent a number of free, open source alternatives to these commercial cores. Two main open source SATA cores are available: one developed in the Reconfigurable Computing Systems Lab at the University of North Carolina at Charlotte (UNCC) [49]; and 'Groundhog', which has been jointly developed by researchers at ETH Zürich, Microsoft Research and Xilinx [50]. Both these cores offer SATA revision 2.0 connectivity to devices, running at 3 Gbit/s, which has a potentially lower performance than the SATA revision 3.0 connectivity offered by most commercial cores, which runs at up to 6 Gbit/s. Additional, a further open source SATA core is based on the

UNCC core [51], which targets older Virtex-4 FPGA family devices and re-uses a large amount of the UNCC core code.

Xilinx also provide a guide on how a Linux system on a Virtex-4 FPGA-based embedded system might be used with a SATA disk using a custom hardware design, along with possible uses for this such as streaming high-speed data from a network interface to a disk [52]. Although the actual HBA IP used in the design is proprietary and unavailable, the overall system shows a possible set-up with DMA and high-speed serial transceivers that can be applied in a wider context.

The Xilinx UltraScale+ series of SoCs contain a hard SATA 3.1 peripheral [42], allowing SATA devices to be used from the Arm CPU or FPGA logic without requiring a SATA HBA core to be implemented in the FPGA logic.

PCI Express Implementations

PCIe implementations for FPGAs are far more common than SATA, thanks to its usage as a more generic high-speed interconnect. For example, Xilinx provide a free hardware core for many of their capable FPGAs, including certain Zynq-7000 devices, allowing a simple link between a PCIe device and main system memory buses. Beyond simple, generic PCIe interfaces, several commercial IP cores exist offering access to specific classes of PCIe devices from hardware, such as network controllers or NVMe storage devices.

Similar to SATA, Xilinx UltraScale+ devices contain a hard PCIe Gen2 peripheral [42], allowing PCIe devices to be used at this speed without requiring a separate core to be implemented in FPGA logic. The high-speed transceivers on these devices can additionally allow up to PCIe Gen4 speeds, but utilising this requires a dedicated FPGA core to manage the PCIe interface.

2.2.5 Development Tools and Processes

Various tools and programming languages are available for creating IP cores to be implemented on FPGAs. For recent Xilinx devices, the main tool for system development is the Vivado Design Suite [53]. This suite of software allows for full system design, using either FPGA-based devices, including with soft-core processors, or SoC devices with integrated hard CPU cores alongside programmable logic. Software for the system can be developed with the same tool flow by utilising Xilinx SDK after hardware generation. SDK can be used to develop and debug both bare metal and Linux applications, through Xilinx's PetaLinux distribution [54], running beside custom hardware.

Hardware cores may be written in a variety of languages, ranging from low-level Hardware Description Languages (HDLs) to more capable High-Level Synthesis (HLS) languages. Using a combination of these is typically required for a complex design, alongside the use

of standard IP blocks. Any parts of a design that are timing-critical or interface with external devices generally require writing in an HDL, such as VHDL or Verilog. These languages allow for the greatest level of control over hardware, but at the cost of increased design difficulty and code size.

At a higher level of abstraction, HLS languages, such as Vivado HLS [55], provide translation from C, C++ or SystemC code down to IP cores that may be added to an FPGA design. These higher-level tools are useful for implementing complex functionality into accelerators, as well as allowing the partial re-use of code from software projects. These tools also facilitate rigorous development through the use of simulation (both at the functional and hardware levels) as well as providing static timing analysis for generated hardware.

Additional tools are available to automate aspects of the hardware/software co-design process further, within specific scopes. The Xilinx SDSoC development environment [56] has support for development of C and C++ designs that may be trivially moved within the tools for compilation targeting software or hardware on a Zynq-based SoC, based on performance data generated from static analysis and live code profiling. This process can potentially simplify the development of an embedded application that utilises a hardware accelerator, but gives less overall control over low-level functionality to the developer. A similar system is provided by the SDAccel development environment [57]; these tools target a server host and accelerator board connected over PCIe, selectively compiling OpenCL, C or C++ code for the two platforms, in order to provide simple support for accelerators in a high-performance computing environment.

2.3 Storage Access in Operating Systems

2.3.1 Storage in Linux

The design of Unix-based operating systems, such as Linux, is centred around the file system, with pipes, network sockets and raw devices appearing to the user in the same way as more-traditional files stored on storage devices [58]. This means the file system stack is one of the most important parts of the kernel when considering efficient execution, as any I/O operations performed by an application will make use of it. The stack includes related standard library functions and system calls, the Virtual File System (VFS), concrete file systems for physical storage devices, and interface and device drivers. The extent of functionality provided by the storage system also affects application areas other than raw performance, with features such as journaling and error checking potentially affecting data integrity, and high CPU usage also consuming large amounts of energy.

Due to its modular structure and open source nature, the Linux operating system supports many file systems, including those developed specifically for it, and more general systems that have been ported to the OS. These file systems may be built as part of the kernel, dynamically loaded as kernel modules at runtime, or executed as user mode processes communicating with a 'bridge' file system interface in the kernel [59].

In Linux, file systems are comprised of three major types: disk-based file systems, used to access files on local storage media such as HDDs and SSDs; network file systems, used to access files on remote servers over a network connection; and special file systems, which give access to virtual data structures internal to the kernel, rather than to files stored on physical media [58].

The Linux Virtual File System

In Linux, the storage stack contains two levels of file systems: concrete file systems and the VFS.

Concrete file systems define the actual structure of data on a device, whether that is a disk, a remote networked server, or some other data structure, as well as the operations that may be applied to it. Many concrete file systems may be in use on a system at once, for example, in order to access data stored on physically different storage devices.

At a higher level than concrete file systems, the VFS is a standard interface presented to user-space applications running on the system. It is a form of abstraction, allowing applications to interact with many concrete file system types without being concerned with the underlying implementation differences [59]. For example, an application may operate on a file from a remote server using an *NFS* file system and a file from an internal disk formatted with an *ext4* file system using exactly the same library functions and system calls. Figure 2.1 shows a high-level view of the VFS and its interfaces to concrete file systems and user-space applications via system calls and standard library functions.

The Linux Common File Model

The Linux VFS operates by providing a 'common file model' – an abstraction of storage capable of representing all file systems supported by the kernel, allowing a common set of operations to be available for every file system [59]. The VFS interface attempts to provide a complete set of file operations, including creation, deletion, reading, writing, permission enforcement and access control, typically accessed by applications through standard libraries, or directly via system calls. Internally, the common file model stores information as a collection of index node (*inode*), directory entry (*dentry*) and file objects, along with the *superblock*, which stores the top level of data about a file

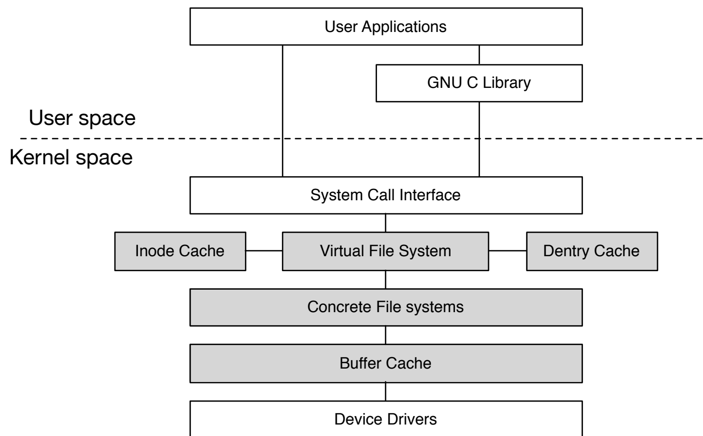


Figure 2.1: Position of the Linux VFS as an interface between applications and concrete file systems, adapted from [60]

system. With this model, directories in the file system are treated in the same way as any other file node, with directory entries providing hierarchical relationships between objects.

When an application opens a file, it will be given access to a file object from the VFS, which references a particular directory entry for the file, as well as including information such as its permissions and the current seek offset. This directory entry then refers to either a further directory entry, or an index node that references the file system superblock and the actual data pages containing the file's contents. From an application's perspective, the underlying structure of the common file model is rarely important (assuming the VFS operates as expected), however low-level details such as caching of directory entries and index nodes can have some impact on performance [61].

While the common file model provides a sensible abstraction for file systems to link to, it can also limit the efficiency of file systems that follow different models, as they must translate their operations to match those in the VFS. For example, the FAT file system architecture stores the location of files within directories as a special 'file allocation table' structure, which differs from the common file model's method of treating directories as standard files containing lists of their children. This means Unix-compatible directory files must be created dynamically for the file system, and kept only in system memory rather than permanently on the storage device [58]. The VFS also adds a slight layer of inefficiency through the necessity of translating any generic file system request into the specific function for that concrete implementation.

Linux Storage Device Drivers

Once file operations have been routed through the VFS and the underlying file system, the kernel must access the physical storage media

containing the data using lower-level device drivers. In Linux, devices are accessible through 'device files', which use a special file system connected through the VFS to the rest of the system. Most storage devices are modelled as 'block devices', allowing a file system to specify that data from a file should be written to or read from a specific 'block' of storage – the basic operation of a file system is to map these blocks to specific files, and to store data structures allowing for the directory tree and file attributes to be saved [58].

This system limits the control a file system has over the disk itself, due to the abstraction of storage hardware to a block device. At a lower level, the device driver will typically implement a standard interface to a controller module, such as AHCI, which will in turn control a disk using a protocol such as SATA [62].

2.4 Storage Technologies and Interfaces

Recent advances in storage technologies are greatly reducing the hardware overheads involved in accessing data from secondary storage, resulting in smaller read and write latencies and increased bandwidth [12, 63]. This reduction increases the pressure on the rest of the storage stack considerably, as any latency from areas such as the file system have a much higher effect in relative terms.

2.4.1 Storage Technologies

The performance and functionality of persistent storage technologies has progressed rapidly in recent years, with an ever-increasing adoption of SSD devices replacing slower and less power-efficient HDDs. The major performance differences between HDD and SSD devices come from their physical operation – with a magnetised mechanically spinning disk versus wholly electronic integrated circuits. The majority of latency on a HDD comes from read/write head movement, waiting for the platter to spin to the correct location, and the inherently sequential nature of reading from a spinning medium. SSDs address these issues, providing much lower access times, especially for random patterns of reads and writes, with average I/O latencies being in the order of 1000x lower [15, 16].

On top of the physical improvements to storage, advances in controller technology has lead to more sophisticated functionality being placed on devices themselves, allowing for potentially higher performance to be achieved. Traditionally, consumer-grade storage devices, often using a PATA interface, provided no advanced functionality on the device itself, with high-level devices offering a little more intelligence through more advanced command sets, such as Small Computer System Interface (SCSI); however, modern storage devices often include intelligent controllers that manage scheduling at the physical

device level, as well as providing fast caches for frequently-accessed data.

Because most operating system storage stacks were originally designed in an era when relatively slow, basic HDDs were state-of-the-art, they attempt to take advantage of the high access latencies and basic operation of physical storage, causing potentially detrimental effects when more-modern devices are used.

The majority of current generation SSDs use NAND-based flash memory chips, which offer higher speeds than HDD storage, but have inherent problems that may be addressed by other emerging technologies [64]. Some alternative non-volatile memory technologies are beginning to become available that improve performance further over NAND flash, such as 3D XPoint memory [65]. Future non-volatile memory technologies, including phase-change memory and spin-transfer torque memory, are set to reduce storage latencies even further while also adding byte-level access granularity, adding even more pressure to ageing storage stack models [64].

2.4.2 Addressing Data

As well as lower access latencies, another change that new memory technologies are bringing to secondary storage is in how data is addressed. With traditional disk-based storage devices, as well as other devices based on the same logical model, data is accessed in blocks, typically consisting of 512 or 4,096 bytes corresponding to a physical sector of a disk. New non-volatile storage technologies, such as phase-change memory and other Byte-addressable Persistent Random Access Memories (BPRAMs), give much finer-grained access to data, allowing for file systems and other storage concepts to take advantage of accesses that are more similar to typical volatile system memory.

Research into utilising the possibilities of BPRAM's addressability in a storage system includes BPFS [11], a file system that is specifically designed to exploit the ability to quickly write individual bytes of data to a storage device. BPFS redefines a large amount of the storage subsystem, reducing reliance on system memory for buffering and caching data, as this can add overheads rather than performance improvements when storage is fast and addressable at the byte level. To achieve this, BPRAM is attached on the main memory bus and addressed like any other system memory, instead of being connected to an explicit storage interface, causing significant changes to the architecture of the storage system, but allowing for potentially large performance improvements for non-sequential accesses.

2.4.3 Storage Interfaces

In order for storage devices to connect to the rest of a computer system, they must conform to an interface that defines both their physical connection and logical controller capabilities. Several interfaces types are available, with the performance of each depending on both the electrical connection and software complexity in kernel drivers.

Serial ATA and AHCI

The main interface used in the current generation of secondary storage devices is Serial ATA (SATA), which allows for bandwidths of up to 1.5 Gbit/s, 3 Gbit/s or 6 Gbit/s in revisions 1.0, 2.0 and 3.0 of the specification respectively [66]. The physical interface operates as a set of two differential signal pairs, allowing for a single channel of half-duplex serial transmission. SATA is typically paired with AHCI, a controller interface that allows the operating system to use a standard set of commands that are valid for many storage devices.

Because SATA and AHCI are based on the older PATA standard, and the protocol and command set was originally designed for HDD devices, they are not optimised for fast execution or efficiency with high-speed NVM technologies. SATA includes features that are mainly targeted at improving the performance of spinning disks, but have more limited benefits in SSD devices. One example of this is Native Command Queueing (NCQ), which allows a disk controller to reorder requests so head movements can be optimised on a mechanical storage medium [67].

It is also possible to interface with SATA devices using schemes other than AHCI, which can give more control over the underlying storage and lower overheads in kernel code. These interfaces include direct connections to programmable hardware cores (such as those discussed in Section 2.2), as well as RAID systems that use custom drivers and hardware to increase the speed or reliability of storage.

Serial Attached SCSI

The Serial Attached SCSI (SAS) protocol is a higher-performance alternative to SATA, generally used in enterprise or mission-critical applications rather than consumer products. While it offers a more capable physical interface than SATA, with up to 12 Gbit/s bandwidth and full-duplex data transmission, SAS is still based upon the older SCSI command set, which was originally designed for use with relatively slow media. The standard is being actively updated and developed, with the upcoming SAS-4 standard supporting 22.5 Gbit/s data transfer speeds [68].

PCI Express

Recently, many high-performance SSD devices are moving to using a PCI Express (PCIe) interface for connection to a host system, due to the high speeds available using multiple lanes of serial communication, and the increased flexibility when compared to storage-specific interfaces such as SATA.

Because PCIe is a general-purpose interconnect, drivers and controllers can be developed specifically to take advantage of the features of high-speed storage technologies, bypassing the limitations of legacy interfaces like ATA and SCSI. This means some manufacturers have created custom interfaces for their storage products, while others have extended existing standards, for example by adding multiple AHCI chips to a single PCIe card. While these methods allow for a potentially higher performance than using SATA or SAS, they are limited by driver support and the implementation of the operating system storage stack.

PCIe is also combined with SCSI or SATA respectively to create the SCSI Express and SATA Express storage interfaces. SCSI Express allows a SCSI device to be connected directly to a PCIe backplane, whereas SATA Express is a composite interface, combining both a SATA link and direct PCIe lanes in a single connector.

NVM Express

In an effort to combat standardisation and compatibility issues amongst PCIe storage devices, the NVM Express (NVMe) standard was developed as an interface specifically for use with fast NVM storage devices connected over PCIe [14]. Because of its specialised scope, the standard contains features targeted for SSD access, rather than backward compatibility or HDD-focused optimisations, while concentrating on providing low-latency access with little software overhead and support for fast physical connections.

An additional benefit of the NVMe standard is the partial restructuring of the storage stack in the Linux kernel implementation. Both SAS and SATA storage drivers use additional layers of abstraction (SCSI and ATA, respectively) as they share common features with legacy protocols, whereas the NVMe driver interfaces directly with both the block layer and VFS. The optimised driver stack can lead to over a 50% reduction in software overhead compared to SAS on a typical architecture, giving better I/O throughput for the same CPU utilisation, and having the potential to support faster future storage technologies without causing a software bottleneck [13]. NVMe also removes many of the overheads introduced by storage controllers, such as AHCI, as the storage device communicates directly with the kernel driver [69].

It is clear from benchmarks of NVMe SSD devices that they offer higher performance than most other available interfaces, reducing the latency caused by software overheads in the Linux kernel by over 50% when compared to SAS [13]. While these software overheads have been reduced through restructuring and collapsing the layers of hardware and interface drivers, this then causes increased pressure on other, unchanged areas of the storage stack, including file systems and the VFS. Therefore, focussing acceleration work on these additional areas could potentially have a significant relative effect on storage latency.

System Memory Bus

A further potential storage interface that is increasingly being investigated for use with byte-addressable NVM is the main system memory bus [11]. Connecting storage directly to this bus, alongside traditional volatile DDR memory, would allow very fast access from the CPU and other system devices, with storage being directly allocated into the system's memory map. This set-up requires a storage device to be accessible at a smaller granularity than is typically available, however, as memory buses are designed for byte-level rather than block-level addressing and transfers.

2.5 Impact of the Software Storage Stack

In the past, the software storage stack has had limited impact on the speed and efficiency of data access, as the major limitation has been down to the slow, mechanical nature of HDDs.

With the current state of file systems and storage device drivers in Linux, software contributes very little to the overall impact of storage operations when using a traditional mechanical SATA HDDs, accounting for around 0.3% of latency and 0.4% of energy usage in a typical system [12]. While this is negligible for HDDs, as the speed of storage devices increases and power consumption decreases, the relative impact of software execution greatly increases. For high-speed, low-power SSD technologies using more efficient interfaces such as PCIe, the same absolute software overheads have a much higher relative effect, accounting for up to 94.1% of latency and 98.8% of energy consumption [12].

Traditionally, fixed overheads from software storage enhancements such as Logical Volume Management (LVM) are virtually transparent in terms of latency and energy usage on many systems, due to the far larger baseline values from the hardware storage device than the software, leading to their widespread use without much regard for negative performance effects. However, as access latencies decrease with modern fast storage devices, this balance can shift greatly, and

the additional layer of block indirection created by using LVM can have significant effects. For example, an increase in latency of 0.03% and energy consumption of 0.04% caused by enabling LVM with a HDD, causes an equivalent increase of 10.70% and 18.70% respectively for a fast NVM-based SSD on the same Linux system [12]. While LVM is optional in many systems, and may typically be avoided in high-performance and embedded scenarios due to its overheads, the extent of the difference in relative effect between slow and fast storage demonstrates an inherent issue with traditional storage stack assumptions.

One possible method to improve the performance and energy usage of software supporting high-speed storage devices is to refactor and simplify portions of the Linux storage stack. This could be achieved by removing functionality that was designed specifically to improve performance of HDDs but can cause lower performance when used with SSDs, while also restructuring file systems and the levels of the kernel that access devices [12].

One possibility when refactoring the storage stack would be to alter the separation of functions between software and hardware, with higher-level operations being output to devices from kernel drivers, and more computation being carried out in the storage interface and controller.

2.5.1 Kernel Design Overheads

The design of the VFS within the Linux kernel is optimised for slow storage devices, making heavy use of caching and other policies that generally favour increased software and memory usage over storage device accesses. For example, the VFS caches recently-used directory entries and index nodes, allowing for frequently used files to often be opened without requiring disk reads to determine their location on the physical medium. Additionally, actual file data is typically read from and written to the page cache instead of directly to a disk, in order to reduce the apparent file system latency from an application's perspective. Modified data in the page cache can then be written to disk during idle periods, when memory contention forces it, or manually using the `sync` system call, allowing for multiple successive writes to the same page of data to be written to the disk just once. While these optimisations are sensible for the slow storage devices for which they were designed, such as HDDs, as storage speeds increase toward those of main system memory, the additional overheads in computation and memory bandwidth involved in keeping comprehensive caches may begin to have a detrimental effect on overall performance, especially in environments with constrained resources [64].

2.5.2 File System Overheads

As a key part of the software storage stack, one focus of research is in quantifying the effects of file system overheads on emerging memory technologies, such as NVM-based storage devices attached to the CPU using fast memory buses.

In order to perform suitable benchmarking, one approach developed a device to emulate high-speed memory technologies with a tunable latency, allowing for possible future storage implementations to be tested alongside current technologies [64]. Results from this work show that for high-speed devices, access times for transfers with small block sizes are considerably worse when a file system is used, compared to when media is accessed as a raw block device [64]. This is caused by the slow speed of software kernel routines and file system functions running on a CPU. The overall performance of the storage is also shown to be greatly influenced by its connection with the rest of the system, with a Double Data Rate (DDR) interface having much lower latencies than PCIe.

As well as basic file system operations, new approaches in areas such as data integrity are also needed in order to exploit the increased speeds of fast storage technologies. Current transaction-based systems, used to ensure consistency following operations on data, are designed around disk-based storage devices, which have slow write speeds and benefit from sequential accesses.

Work on improving transaction mechanisms for fast storage includes MARS [70], a system that uses 'editable atomic writes' to closely tie a fast storage device with its transaction support. Portions of the system are implemented in hardware alongside an SSD, reducing the software overheads involved in logging and enforcing transactions, and ultimately offering a reasonable speed improvement compared to other transaction schemes used with the same storage device. The concept of editable atomic writes was extended further for the Willow user-programmable SSD [71], allowing them to be used as a feature that can be optionally and dynamically programmed into storage hardware, alongside other 'SSD Apps' also running on the storage hardware itself.

2.5.3 Bypassing the Kernel

In order to reduce the overheads involved in accessing fast storage-class memory technologies, work has been carried out to bypass the kernel storage interface to allow certain applications direct access to storage devices.

The Moneta Direct (Moneta-D) storage architecture [72] moves virtualisation and protection policy enforcement out of the kernel and file system, and into hardware built upon the Moneta SSD [73]. This al-

lows applications to safely interface with high-speed storage, while bypassing the computation bottlenecks present in standard kernel-based solutions, but still maintaining full POSIX compatibility. Moneta-D eliminates most file system and operating system overheads, and provides a significantly lower latency interface when compared to the same storage device used with the standard operating system storage stack. Benchmarks show the total reduction in write latency can reach 42% to 69%, depending on access size, with similar values for read latency [72].

Aerie [74] is another, more recent file system architecture that also has the ability to bypass the Linux VFS in an attempt to remove the inefficiencies that it introduces to the storage stack. Two file systems are implemented and tested on top of Aerie: PXFS, a full POSIX-style file system; and FlatFS, a specialised, simple file system with limited functionality. The aim of implementing the two file systems is to show how both a POSIX-compliant file system and a specialised file system can take advantage of direct communication with an application in different scenarios. The FlatFS implementation relates to the idea of specialised, application-specific file systems discussed in [75], demonstrating that a file system created for a specific use, with no unnecessary general-purpose functionality, has the potential to improve performance. The performance improvements achieved with Aerie vary, with some scenarios achieving throughput improvements of up to 109% compared to a standard in-kernel file system, while others show little or no improvement [74]. Both PXFS and FlatFS give the greatest improvements when workloads are matched to their strengths, compared to those of the VFS, with FlatFS especially offering a large improvement for applications that can benefit from its simplified interface.

DevFS [76] is a further hardware-supported file system that aims to reduce kernel overheads by moving file system functionality into storage device hardware. User applications can directly access data stored using DevFS through a standard POSIX interface with minimal kernel involvement, while retaining the integrity, concurrency, crash consistency and security guarantees of a standard kernel-level file system, assuming the file system code running on the device itself is trusted. As well as any potential performance improvements, moving file system functionality onto storage hardware allows for additional benefits, such as the ability for file system operations to be safely completed by the device in the event of a system crash. An emulated version of DevFS shows a possibility for doubling I/O throughput and reducing RAM utilisation by a factor of 5 with certain benchmarks [76].

All of these systems demonstrate the inefficiencies of a traditional storage stack when combined with high-speed storage technologies, and show how bypassing the kernel and VFS, potentially with additional hardware support, is a viable and effective method of reducing

latency in a storage system, as well as allowing for further features and benefits.

2.5.4 Asynchronous I/O Stack

An alternative to bypassing the kernel for storage access is to attempt to improve the operations being performed inside the kernel, taking advantage of the low-level hardware access and elevated privileges afforded by executing code at this level. One method that can improve performance of storage stack operations running on the CPU is processing them in parallel with hardware accesses, creating an asynchronous I/O stack [63]. Several areas are identified that can be performed in parallel with the storage device access, which are traditionally processed sequentially before or after the storage device has performed the physical read or write of data. These include cache management, memory page allocation, DMA mapping and unmapping, and processor context switching. In combination with a lighter-weight block I/O layer, running these operations in parallel with storage device operations can produce significant latency savings with low-latency storage hardware – up to 33% for random reads and 31% for random writes on synthetic FIO benchmarks, and a 11-44% reduction in observed latency for real-world database benchmarks [63].

2.6 File System Functionality and Optimisation

Research into file systems generally has two main focuses. The first is concerned with the overall functionality of a file system, enhancing this through the introduction of features such as journaling and copy-on-write, or examining how a file system can work in non-standard situations, such as in a distributed system. The second is about efficiency of file system operation, with focus on how a storage system can potentially be improved for specific situations or device types, through simplification, specialisation or radical changes to the software/hardware stack.

These areas are not entirely exclusive, as new functionality may be introduced with the aim of improving performance and increasing efficiency, especially when focussing a file system for a certain purpose.

2.6.1 File System Functionality

The functionality of file systems is an active area of research, with aims to increase the features available to applications further up the storage stack, and to better address the changing state of storage devices on which file systems are used. Some changes in functionality are more fundamental than others, with some focussed on enhancing specific

scenarios, such as real-time embedded systems or high-performance computing applications, and others focussed on more general-purpose improvements for a variety of computer systems.

Layered/Stackable File Systems

Layered or stackable file systems are a concept to allow functionality to be incrementally developed into a full file system over time, or for the creation of file systems that select several pre-created functions and combine them into a single logical file system [77, 78]. Each layer of the file system may extend functionality of the others, for example, a cryptographic layer may be combined with standard read and write operations in order to provide encryption of stored data on a file system that would not otherwise support it.

An approach such as this may be useful for situations where a standard monolithic file system is not entirely appropriate, such as in an embedded device where resources are limited, or in a heterogeneous system where different parts of the file system could be executed on different architectures, for example, partly on a CPU and partly on hardware accelerators. Examples of applications developed from this approach include a 'redundant array of independent filesystems' [79] – an alternative to RAID that replicates or stripes data at the file level instead of operating solely on lower-level blocks, allowing for more intelligent optimisations to be made.

2.6.2 File System Optimisation

File system performance can be improved through a number of methods, covering both optimisation of functionality, and design decisions that benefit certain workload patterns. Optimisation may cover multiple metrics, including raw I/O throughput, power usage and storage device wear.

File System Specialisation

Many modern file systems contain a large number of features beyond basic file operations, including encryption, journaling and access control lists, which may not all be required by a large number of applications [75]. It has been proposed that these 'obese' file systems could be replaced by simpler, more specialised file systems that offer better performance for the system they are targeted towards. For example, a file system targeted for a mobile embedded device could perhaps focus on energy efficiency and low resource usage, while a file system targeted for a server containing critical data could focus on redundancy and reliability, without concern for efficient disk space or power usage.

Following from this, matching applications to suitable file systems and associated features on Linux has been found to potentially reduce CPU usage and power consumption in certain situations [80]. As file systems implement features in different ways, such as how they distribute files across disk blocks and how files and directories are indexed, basing the choice of file system and its configuration options on the expected workload can make a significant difference to performance.

Layered or stackable file systems [77] could be used as a method for preventing file system obesity, as they allow for combining multiple basic file system features into a system that is tailored for its specific workload. The overheads and complexities introduced in the stacking architecture, however, may reduce potential performance improvements.

VFS Optimisation

Due to its importance and high usage, the Linux VFS is a heavily optimised part of the kernel, both in its code implementation and its overall design. Despite this, some experimental work looks at restructuring and reimplementing the VFS, alongside low-level file system design changes, in order to offer more features specific to the goals of an underlying concrete file system design. SpadFS [81, 82] is a file system that may be used with either the standard Linux VFS or its own custom implementation, Spad VFS. Combining Spad VFS and SpadFS makes it possible for additional features to be added to the storage stack that are not possible when using the general-purpose VFS, such as optimisations for the storage of small files [82]. A restructured VFS can also offer features to other file systems that would not be possible otherwise, such as delayed allocation and cross-file readahead, that can allow for additional specialised operation [82].

Results show that using SpadFS alongside its partner VFS can result in lower CPU usage, particularly in how it scales with storage speed, with benchmarks showing up to 6x lower CPU utilisation than ext2 when writing files. [81]. However, when compared to running SpadFS or certain other file systems with the Linux VFS, it does not always offer higher performance or the best raw I/O throughput [82]. This shows the complexity involved in the VFS layer, and how making changes that benefit certain scenarios can also be detrimental for more general-purpose usage patterns; for example, the SpadFS/SpadVFS combination could be suitable for use in systems where low CPU usage is more important than high speed, or where CPU bottlenecks are the reason for speed limits.

2.7 Special-purpose File Systems

File system features and optimisations can be focussed on improving support for special-purpose applications, such as high-performance distributed computing or real-time embedded systems. Special purpose platforms can present specific problems that must be overcome, as well as allowing for optimisations that may not be available with more general-purpose systems.

2.7.1 Real-Time and Embedded File Systems

Real-time and embedded systems face two problems when dealing with storage. Firstly, an embedded system will typically be constrained by its available resources, such as CPU speed and memory capacity. This can be impacted further by requiring quality-of-service guarantees for other aspects of the system to be honoured, meaning a file system cannot use arbitrary amounts of system resources. Secondly, the real-time nature of a system requires that a certain level of predictability of operation is provided – it must be known ahead of time at least the worst-case timing of a data access, so system timing models can be verified.

The resource constraints of an embedded system can be mitigated to a certain extent using optimisation techniques – creating file systems that are not overly complex in terms of features, or reliant on high-performance hardware. Hardware acceleration is a method that can potentially be used to take file system load away from shared resources such as a CPU, and move it onto dedicated hardware instead, reducing the potential of interference with other system processes [22].

The real-time nature of a storage system is a more difficult problem to address, especially when working with a complex operating system. Due to the many interacting layers of the Linux storage stack, combined with the extensive buffering of data in the kernel and the abstractions provided by hardware interfaces, it is difficult to predict or control exactly what a file operation will do in terms of physical storage, and the timing of any operations. For example, once an application completes a canonical write operation on a file, there is no guarantee when the data will be actually written from memory to the storage device, or even that it is written at all, in the case of another process writing to the same data page or a system power failure occurring. In Linux, it is possible to open files in 'direct I/O' mode, which bypasses the page cache, transferring data directly from the disk into the memory space of the application; this gives more control to the application, but also increases its responsibility for the efficiency of file operations and can slow down apparent execution time [58].

While features like direct I/O mode improve predictability to some extent, underlying file systems and storage devices must also give tim-

ing guarantees in order to be considered for real-time use. Solid-state storage devices have the potential to offer higher timing predictability than traditional hard disks, mainly due to the absence of mechanical parts that inherently cause potential for variation in access times compared to purely electronic circuits [83]. This allows for possible worst-case storage access times to be kept much closer to best- and average-case times, providing advantages for real-time analysis.

File systems research has touched on real-time usage, with much of the work focussed on real-time scheduling of storage requests. Beyond this, a full real-time file system has previously been developed for the RT-Mach kernel [84]. While this work is relatively old and targets an outdated operating system, concepts and motivation may be taken from the implementation, which shows that real-time guarantees can be achieved with a low throughput penalty. Advances in storage technology that remove mechanical parts from the process may give further use to this work, which found that the movement of HDD heads was the largest obstacle to efficient predictability, although conversely an increase in the complexity of storage device controllers may add additional unpredictable aspects. In a more modern context, real-time file systems may be deployed in embedded environments to assist with the strict timing requirements of cyber-physical systems, reducing jitter and increasing predictability through avoiding the storage stack of a general-purpose operating system [85].

2.7.2 Real-Time I/O Scheduling

An area of Linux storage stack that has been investigated in depth to provide increased predictability for real-time applications is the I/O scheduler.

The Active Block I/O Scheduling System (ABISS) [86] is an extension to the Linux kernel's storage subsystem that attempts to give applications guaranteed I/O bandwidth when they request it, so a high system load should not have such a negative effect on applications that request a constant level of I/O performance. ABISS also allows for requests to be denied if they will violate the maximum available load of the storage system, warning applications of the violation rather than simply failing to fulfil requests.

The Fahrrad real-time disk I/O scheduler is a further alternative scheduler implemented for the Linux kernel [87], considering the reservation of storage resources in terms of utilisation, rather than the the more standard measure of throughput. When considering utilisation compared to throughput, more closely aligned best, worst and average-case values are common, allowing the scheduling algorithm to better prioritise access to storage to achieve a less pessimistic overall utilisation [87].

Another method used to improve I/O scheduling is through measuring disk latencies to provide more accurate estimation of the time taken for requests to be serviced by physical storage devices [88], compared to using pessimistic worst-case times or potentially inaccurate statistical distributions. Due to the complexities of modern storage device interfaces, which are implemented with logical block numbers rather than a direct mapping to physical portions of a drive, traditional disk request reordering algorithms may not produce the expected effects. Measuring latency and building an image of a specific drive's characteristics from the results allows for good predictability of performance across any device, improving the potential for real-time guarantees to be accurately met.

2.7.3 Distributed File Systems

For applications that run in a distributed manner across many compute nodes, it is desirable for file systems to also perform in a distributed way so files can be accessed from multiple locations, and so large amounts of data can be stored seamlessly across many physical disks. Two examples of such file systems are the Hadoop Distributed File System (HDFS) [89] and Lustre [90], both of which are designed to operate across many servers in order to provide large amounts of easily scalable storage, potentially in the order of petabytes, to large-scale or high-performance computing applications.

Lustre uses separate servers for metadata and object storage, to facilitate the distribution of file system data, and allows for full redundancy of nodes in order to increase reliability and access times. HDFS also operates in a generally distributed manner, but is more closely tied to working with the rest of the Hadoop tools, and uses 'data locality' to keep data close to where it is used computationally. Unlike Lustre, HDFS does not expose a fully POSIX-compliant interface, making it less easy to operate with the Linux VFS.

Ideas from distributed file systems may be taken for use in smaller-scale heterogeneous systems, such as separating metadata and storage across hardware and software on an embedded device. These file systems are also important due to their use in many large scale data systems.

2.7.4 File Systems for Hybrid DRAM/NVRAM Memory

Specific storage memory technologies can benefit from specialist file systems that are optimised and targeted towards their characteristics. For example, byte-addressable non-volatile memory that is accessed on the main memory bus, alongside traditional volatile system memory, will have vastly different performance and access properties to a block-addressable storage device.

NOVA [91] is a file system that specifically targets hybrid memory systems – those that contain both volatile and non-volatile memory on the main system bus – reducing the storage overheads of file systems designed for more traditional storage media while continuing to provide consistency guarantees. Through testing with an implementation of the file system in the Linux kernel, running on emulated storage hardware, NOVA achieves 11.4x higher operations per second than ext4 in data journal mode (which gives similar consistency guarantees) for a simulated file server workload [91]. Other workloads show similar performance improvements compared to file systems that are not optimised for this hardware set-up.

Another file system designed to take advantage of specific properties of NVRAM hybrid memory systems is DurableFS [92], which uses the speed and granularity of byte-addressable storage to provide efficient atomicity and durability guarantees for file system transactions. Across a number of write-intensive benchmarks, DurableFS is slower in transfer speed by between 9 and 11 percent compared to NOVA, but provides firm guarantees that data has been flushed from caches in the memory system and written to non-volatile storage, and that transactions have completed fully and atomically [92]. This trade-off in performance against data integrity may be worthwhile in many domains, such as in an embedded system where a reliable power supply may not always be guaranteed, or in a high-integrity system where data durability is more important than raw storage speed.

2.8 Hardware Acceleration

For high-performance, computationally-intensive tasks, accelerating functionality by offloading it to dedicated hardware is common practice. For example, most modern computers contain dedicated hardware for graphics acceleration, and many file servers include hardware RAID controllers to improve disk access times or provide redundancy without requiring software running on a CPU.

More specialised hardware accelerators are often found in embedded systems that have limited CPU speeds, or specific power or timing requirements [93, 94]. This is due to the generally higher performance available from hardware implementations of certain software routines, as well as the availability of efficient devices such as FPGAs and ASICs for these tasks.

High-performance computing applications can also be assisted by specialised hardware, in order to increase speed and reduce power consumption. This can be achieved through the use of hardware accelerators connected to a regular server set-up, for performing common but computationally intensive tasks, such as regular expression matching [95] or memory transfers [96].

Embedded accelerators are also increasingly being investigated for use in high-performance computing environments, due to their energy efficiency when compared to traditional server hardware [21, 97]. 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 [64, 80].

2.8.1 Hardware-Accelerated File Systems

One method of potentially improving the performance of file systems is to move them (or portions of them) from software running in user space or a kernel, to hardware, bringing functionality closer to peripherals that may be using storage, and allowing a file system to take advantage of the features of dedicated logic such as reducing CPU load. Research in this area is relatively limited, however a few implementations of hardware file systems have been proposed or developed to experiment with the possibilities of accelerating storage at this level.

Active Storage

Before the era of modern high-speed, flash-based storage devices, there was interest in examining the separation of software and hardware in the storage stack, prompted by the increasing possibility of embedding more functionality into hard disk controller logic [98, 99]. This 'active storage' aimed to give more context to file operations performed by an operating system from the disk's point of view, potentially allowing for intelligent data optimisations to be performed in hardware rather than in a software file system [100]. Modern mainstream disk interfaces implement some features similar to these, but with a smaller scope and less context-awareness, such as NCQ introduced in revision 2.0 of the SATA specification [67], which allows for out-of-order execution of commands so a mechanical hard disk can optimise head movements.

While introducing additional features into the storage hardware can be advantageous, it can also cause trade-offs with metrics such as predictability and real-time performance. For example, in a system where the storage device itself queues and reorders requests internally, it can be much more difficult to control and predict the exact time at which a request will be completed, likely increasing the separation of best- and worst-case execution times.

HWFS

The most fully developed hardware file system implementation is HWFS, which was created as part of a project to investigate high-performance computing accelerators implemented on a cluster of

FPGAs [101]. The aim of implementing the file system in hardware was to allow accelerator cores running within the cluster to access data from multiple mass storage devices, without relying on the use of slow, embedded processors as an intermediary.

The basic operation of HWFS is based on the widely-used Unix File System (UFS), but with much more limited and specialised functionality [22, 35]. As it is implemented entirely in hardware, and written in a basic, low-level HDL, the file system functionality is limited, with only open, read, write, delete and seek file operations supported. To further remove reliance on software, HWFS is designed to interact directly with a storage device using a purpose-made SATA 2 interface FPGA core [49].

According to benchmarks carried out on the core, HWFS performs better than an ext2 file system [102] running on x86 Linux when reading from and writing to a fast SSD [22]. This performance improvement is particularly prominent when the hardware-accelerated core is compared to an embedded, soft MicroBlaze processor, showing over a 16 times performance improvement on some operations, despite being run at a slower clock speed.

The large increase in speed between ext2 running on a MicroBlaze and HWFS shows how the limited performance of embedded processor cores can have a significant impact on file I/O access times, and how it is possible to make improvements through hardware acceleration on this kind of system.

Since these results were published, high-speed storage technologies have moved on some way, with the faster SATA revision 3 and PCIe interfaces becoming more common. While FPGA and CPU technology has also progressed, the relative increase in storage access speeds has outpaced this, so there is likely to be an even greater potential for acceleration to give a considerable performance improvement with more-modern hardware.

The constrained functionality of HWFS when compared to most general-purpose file systems, such as the flat file organisation structure used instead of a directory hierarchy, make it unsuitable for many applications, which is partly due to its design aims, but also the complexity of implementing functionality in low-level hardware description languages such as VHDL and Verilog. The use of HLS languages and tools should allow for more advanced functionality to be added to a hardware file system implementation, making it more suitable for general-purpose computing, however the scope of the file system should still remain focussed on its intended use to avoid introducing unnecessary inefficiencies.

File System Access Accelerator

An alternative approach to file system acceleration is taken by File system Access Accelerator (FAA) [37], which offloads just part of the

file system, specifically directory search, to a hardware accelerator core implemented on an FPGA. The idea behind the accelerator is to reduce the latency involved with the translation of file names into inode values by a Unix-style file system, by keeping a dedicated cache of items in fast flash memory, and taking advantage of the programmable logic's ability to perform a vastly more parallel search than a CPU. The accelerator subsystem is connected directly to the host CPU via the Front-Side Bus (FSB), next to system memory, giving the software kernel fast access to results from the core.

The addition of FAA to a file system would increase the overall complexity of the storage stack, however full original file system operations are retained, as the file system is still mostly operating unmodified in software. While no implementation of the accelerator system was produced, a high-level model of the timings involved shows potential for reducing file system latency using this method.

File System on Chip

The idea of including an embedded file system on a storage device is proposed by File System on Chip (FSOC) [36]. This project aimed to move all file system functionality from the host computer to a removable storage device, so the host requests high-level logical 'file' objects, rather than specific physical sectors from the device when it is accessed. The interface to this is either implemented as a custom kernel module driver on the host, or as a pseudo file system that passes requests through to the hardware using an interface similar to remote procedure calls.

The benefits that FSOC aims to achieve are focused mainly on removable storage, allowing the file system to be tailored to low-speed devices and for scenarios such as sudden power loss. One reason cited for performing file system operations on the device is to avoid the requirement of specific file system support on the host, however it introduces the additional requirement that the kernel has support for the custom hardware interface used. A benefit of this is allowing for extra file system features such as encryption or compression to be added transparently to a device's file system, without any changes being needed on the host.

Overall, evaluation of FSOC found it to perform faster than the same file system design running on a host system with a low-powered embedded CPU for certain high-load tasks [36], however results varied greatly, mainly depending on the ratio of compute to I/O time in a workload. While the specific characteristics of the embedded hardware used will have a large effect on the absolute performance results of such a system, the general trends of workload types affecting relative performance remain relevant across a larger variety of systems.

While the focus of FSOC differs from ideas of accelerating high-performance storage for large-scale data, similarities exist in the mo-

tivation of reducing CPU load and utilising the parallel capabilities of dedicated hardware for increased performance. The way that the work presents models for the calculation of possible effectiveness of the system, and abstracts metrics from the low-level programming of the hardware, is a useful method to consider when carrying out research into possible acceleration schemes.

Further Implementations

A further hardware file system is proposed and partially implemented in [103], based on a simplified ext2 model. While the premise behind the file system is described in detail, the actual implementation is missing some core functionality required for it to be universally usable, such as the ability to create and delete files. This demonstrates the potential difficulties associated with implementing a complete file system in hardware, with factors such as low-level HDLs and high build times adding overheads into the development process.

2.8.2 Hardware-Accelerated Data Processing

As well as hardware acceleration of storage access itself, a hardware acceleration approach may also be taken with other processes that operate on stored data, such as compression, encoding and encryption, allowing for applications to transparently access files in a representation that is different from the data on the actual storage media. This acceleration may be placed at various points in the storage stack, either before or after data is translated from logical files into physical block addresses by the file system.

One use of accelerated data compression is for reducing the bandwidth requirements of a large-scale distributed system, where a large amount of data may be passed over a network between compute and storage nodes. AltraHD [104] is an example of a hardware-accelerated data processor that compresses files before writing them to disk or sending them between nodes of an Apache Hadoop cluster. It is implemented as a PCIe expansion card that intercepts data between HDFS and a native file system, in order to provide fast compression and decompression that does not affect the operation of other parts of the system. The actual file system access is carried out through a standard software interface, rather than bringing this into hardware alongside the data processor.

Similar designs have been used for other data- and computation-intensive applications such as real-time data encryption and decryption, with accelerator cards providing transparent access to storage whose contents is encrypted [105].

While this model of acceleration offers the advantage of requiring minimal modification to the original system, an area for research would be into the combination of data manipulation and lower-level

functionality like a file system into a single accelerator, potentially allowing for further reductions in CPU overheads for large-scale data processing and storage.

2.8.3 Operating System Acceleration

Beyond storage and data processing, the traditional hardware/software split when managing low-level system tasks can be reconsidered in other areas of an operating system. As an operating system performs many functions on a computer, dealing with almost all interactions between applications and hardware peripherals, it has the potential to slow down operations and use large amounts of processing resources. While operating system kernels are generally very heavily optimised for efficiency, it may still be beneficial to go further in certain cases and offload part of their functionality onto hardware accelerators. This can also allow more streamlined access from other accelerator cores in the system, giving them direct access to devices that would otherwise require communication through software.

Memory Copy Acceleration

A common function performed by an operating system kernel is the copying of data between different locations in memory. In Linux, when accessing file systems and drivers implemented in the kernel from a user-level process, data must be copied between areas of memory assigned to each privilege level, often using the *memcpy* function [58]. As this can be a very I/O- and CPU-intensive task when used for constructing large regions of memory as buffers for processes such as network and storage accesses, *memcpy* has been demonstrated as a bottleneck in the implementation of the Bluetooth standard in the Linux kernel [106].

A hardware implementation of *memcpy*, created as a coprocessor core on an FPGA, has been shown to significantly improve the performance and reduce the CPU load of certain memory copy operations, and in turn the operation of applications and drivers that depend on it [96]. This shows an example of a situation where hardware acceleration can be feasible and beneficial for specific functionality within an operating system, although it may not be generalisable to all platforms and applications. Alternative optimisations may also be possible through software changes, as well as hardware acceleration.

FPGA Operating Systems

An alternative approach to hardware acceleration of specific operating system functionality is to treat an FPGA accelerator more like its own computer system, adding an operating system into the logic that can perform functionality on a per-FPGA level. One example of this

is the Feniks FPGA operating system [38], developed to assist with FPGA resource sharing in cloud computing environments. As well as the advantage of reducing reliance on a system's CPU for tasks such as networking, storage access and memory management, the inclusion of these functions on the FPGA itself allows it to operate as a more independent unit, with multiple applications being able to run separately on a single FPGA above the operating system, and multiple FPGAs within a single physical system having the ability to manage their own resources. The ideas of cloud computing are further supported by the FPGA operating system instances being able to communicate directly across a network, allowing accelerated applications to be spread transparently between physically separated nodes.

2.8.4 Accelerator-Aware Operating Systems

As an alternative to using a hardware file system or explicit operating system accelerators for allowing access to storage and devices from other accelerator cores, the BORPH operating system [107] gives hardware direct access to a traditional file system and other services running within a software kernel, via a hardware system call interface [108]. The idea of this is to keep the main operating system compatible with standard Linux applications, while hardware access to kernel operations is provided by additional system calls and extended functionality for operations such as initiating data transfers from applications running as hardware modules. This allows hardware accelerators to act far more like software processes (one of the key ideas behind BORPH), with Unix pipelines used to stream data from one process to another transparently, regardless of whether it is running in software or hardware.

Results from this work show potential for extending software operating systems to allow more transparent interoperability with hardware accelerator cores, including access to storage via a file system hardware system call interface; however the large amount of I/O overhead required may also cause an overall reduction in performance, especially as the CPU must carry out all file system operations as well as processing the specialised system calls. Due to this, a full hardware file system may offer better results when maximising performance is a key factor for storage operations, although this has other limitations such as added complexity and reduced transparency.

2.9 Measuring Storage Performance

In order to effectively measure the performance of storage systems at various levels, consideration must be given to relevant profiling and benchmarking techniques and tools, and their appropriate applica-

tion to experiments. While benchmarking allows performance to be measured at a high level, with measurements showing potential best, worst and average latencies, overall bandwidths, and the observation of patterns within data access types, profiling exposes the low-level operation of software and hardware, and is necessary to understand the reasons behind specific benchmark results.

2.9.1 Storage System Profiling

Several tools exist for performing application and kernel profiling in Linux. Generally, these tools monitor hardware performance counters in the CPU, along with periodically interrupting code execution and storing the current instruction pointer, in order to measure paths through a program and to give an impression of the time spent within each major function [58].

Several considerations must be made when performing profiling, in order to capture useful data and interpret results correctly. As profiling must intermittently interrupt a program's execution to record data, it can affect the performance of a system while it is active. Also, due to execution only being sampled periodically, running a program for a longer period or multiple times is required to gain an accurate impression of where time is spent during execution, as more samples will be taken covering a broader area.

Two major statistical profiling tools for Linux, which make use of the kernel's performance events subsystem, are *perf* [109] and *OProfile* [110], both of which provide similar features including support for profiling user- and kernel-level code from a single process or a system-wide perspective, and outputting call-graph information, annotated source code, and various forms of report data [111]. Another popular Linux performance analysis tool is *gprof* [112], which can generally be used in a similar way, but cannot measure time spent in kernel mode, so is not useful for measuring low-level system operation.

Besides these statistical profiling tools, the main alternative method of profiling involves timing specific sections of code, such as individual functions or loops (either manually or using automated tools), and potentially associating this with a call graph of the application generated using a similar method. Explicit profiling points must be inserted into the application, either before or during compilation, which trigger profiling functions for recording timing information. This can be achieved either through explicit software timing of events in source code, using custom timing hardware, or using a tracing framework such as DTrace [113]. Explicitly tracing an application's execution can have advantages over interrupt- or timer-based profiling tools, as the results are guaranteed to cover every event that is set-up to be measured, however this comes at the cost of requiring modifications to an application's source code or machine code, and a less consistent

pattern of interference caused by measurement, which will happen more sporadically.

A further method of profiling can be to statically analyse machine code to calculate execution time estimates, however this is only feasible and accurate for relatively simple programs that perform little interaction with external systems and devices.

2.9.2 Benchmarking Performance

In order for profiling to be carried out effectively, and in order to test any effects of modifications made to a system, a reasonable set of measurements and benchmarks must be established. These can be taken from standard storage benchmark sets, or developed specifically for the task, based on potential uses of the work. While standard, well-established benchmarks are useful for comparisons with other work, they can vary significantly in quality and applicability, and certain situations are too specialised for any options to apply adequately.

Three main categories of file system benchmarks are available: macro-benchmarks, which use multiple file system operations to give an overall impression of performance; micro-benchmarks, which test a small set of specific operations; and trace-based benchmarks, which emulate behaviour recorded from real-world systems [114]. In order to gather a useful impression of a system, a combination of benchmarks should be used, with appropriate functionality targeted for intended applications, as well as more general benchmarks to ensure unexpected impacts on other areas of a system are avoided.

Many standard file system benchmarking tests and applications exist for use on Linux, however their quality, in terms of creating reproducible results and reflecting real-world system usage, varies greatly [115]. It is therefore necessary to use multiple tests in order to increase confidence in the results of any file system acceleration work, and to select these tests based on well-researched criteria.

A large amount of work exists on the complexities of file system benchmarking, including studies of benchmarks typically used in research [114], and the general desirable characteristics that benchmarks should exhibit [115]. These characteristics include the accuracy of results, the ability to repeat experiments and to reproduce results, and the classification of a benchmark in terms of its specific performance dimensions and the applicability of these to the desired metrics being judged [115].

Specific tools also exist to assist in the generation and selection of benchmarks, and for producing test suites with reproducible, realistic results [116].

File system usage patterns vary greatly between different applications [116]. For example, a web server's file system will typically mostly deal with reads from a similar set of files, allowing for a large

amount of caching to be performed efficiently; whereas a scientific application that produces a large amount of data will typically require continuous writing at a sustained bandwidth while the experiment is running followed by reading from areas of interest at the highest speed possible as data is analysed. For this reason, the use of suitable benchmarks is required in order to effectively evaluate a system designed for certain usage patterns.

2.10 Modelling Storage Systems

Creating a standard high-level model of storage access is useful in order to abstract away from the low-level details of implementation, allowing the functionality and results taken from analysis and experimentation to be demonstrated in a consistent manner. Abstracting the storage system model gives a broader look at the interactions of the various layers of the storage stack, which allows for a reasoned analysis of its operation, while being useful for predicting the effects that changes may have on a system.

Persistent storage access in many systems, from embedded devices to large-scale cloud computing platforms, operates on a similar principle – creating an abstract interface to the user and applications that ultimately represents data stored on some underlying storage device. Typically, data is indexed and accessed through a hierarchy of files (although this is not always the case), which then map to a number of fixed-size blocks on a physical device. This mapping can be managed through multiple interacting layers, including file systems, various operating system mechanisms, and processes on the storage device itself. The layers involved in this abstraction often result in the use of standard interfaces and interchangeable components at each level.

For the purpose of the work presented in this thesis, storage access time is divided into three main domains: the application, the operating system, and the storage device. While each of these areas contains multiple different sub-processes depending on the system and context, the high-level view is constant throughout. The general pattern of access also remains constant through most systems, with applications communicating in both directions with the storage device *via* the operating system, however there are cases where the operating system can set-up direct communication for applications ahead of time.

2.10.1 High-level storage stack model

Figure 2.2 shows a number of possible areas covered by each of the major storage model domains, with time spent during a storage operation progressing from left to right. Many operating systems implement all the areas outlined in the model in some form, however not all areas are required in order to create a viable storage stack.

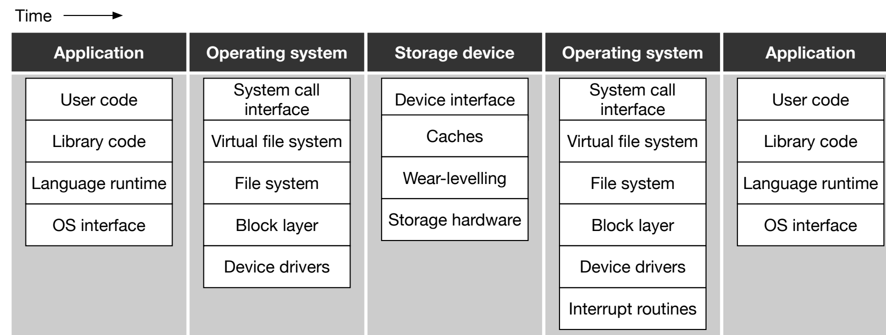


Figure 2.2: Potential areas of storage stack, grouped by high-level domain

2.11 Measuring Efficiency and Predictability

In order to provide meaningful measurement of the efficiency and predictability of storage operations (two key criteria for real-time embedded systems), a consistent set of metrics must be defined for use throughout the analysis and evaluation work presented in this thesis. These metrics are as follows:

- Read and write speeds
- I/O operation speeds
- CPU usage during transfers
- Number of device commands
- Number of device interrupts

In addition to values of these primary metrics, additional insight can be gained through further analysis using higher-order metrics, such as the following:

- Ratio of one metric to another (e.g. speed to CPU usage)
- Variation of values over time
- Distribution of values over time

The metrics are also related to areas of the high-level storage stack model defined in Section 2.10. For example, certain metrics may apply only to some areas of the storage stack, and may have to ignore the effects of other areas in order to be usefully quantifiable.

One key requirement for real-time systems is the ability to calculate bounds for execution time – for example, guaranteeing the worst case amount of time taken (at least probabilistically) for a certain operation to be executed. Within a complex system with many interacting parts, such as a typical storage stack, providing timing bounds across an entire operation can be difficult. This is especially the case when using proprietary devices such as storage hardware, whose specifications do not provide any kind of timing guarantees themselves, and whose low-level functionality is beyond the control of the user or system

creator. In such a case, it can be necessary to work around the areas of a system that have unknown timing properties, providing guarantees in areas that can be controlled, while highlighting areas that cannot.

Thorough statistical measurement of execution times can also assist in generating timing characteristics of black-box systems (or subsystems), which can be sufficient in many cases. Attributing these timings to certain areas of the overall model, and correlating them with non-timing measurements such as command and interrupt counts, can provide further useful data on the overall predictability of a system.

In addition to the raw metrics, techniques such as execution profiling can be used to identify in which areas time is spent during a transfer, however performing these in an accurate and non-intrusive manner can often be difficult.

2.11.1 Outline of metrics

READ AND WRITE SPEEDS Data transfer speeds when reading and writing to storage can give a good indication of the overall performance of a storage system, covering all layers, from an application, through the kernel, to a hardware device. When comparing raw speeds, the various parts of a system must be taken into account, as a limit can often be due to a bottleneck caused by a single factor, such as CPU or storage device speed. The nature of the specific data transfer being measured must also be considered, as copying many small blocks of data can have vastly different performance characteristics to copying fewer large blocks. Transfer speed can be a good way of measuring efficiency, assuming specific bottlenecks are understood as well as predictability when considered over time. This metric is typically expressed as the number of bytes transferred per second.

CPU USAGE DURING TRANSFERS CPU usage can show the general efficiency of the software component of a storage transfer, with lower CPU usage often correlating with higher performance. This can be particularly relevant in embedded systems, where the CPU resources available to storage tasks are often limited. Measuring CPU usage has the issue of being capped at 100% – when a CPU is fully utilised, simple results cannot be meaningfully compared as the resource is saturated, however this information can still be useful when combined with other metrics such as speed to give an indication of efficiency. Other aspects of a system must be considered alongside raw CPU usage values to keep them in an appropriate context, such as the CPU architecture, number of available cores, memory and cache speeds, and the characteristics of the storage device being used. This metric is typically expressed as the percentage of time that a CPU is executing code from a certain area of the system (and is not idle), during a given sample period.

NUMBER OF DEVICE COMMANDS The number of individual commands sent to a storage device during a transfer (and whether they are directly caused by that transfer) can give a good indication of overall software predictability, including all layers of the application and operating system. This metric only gives an indication of consistency in a single area, however, as the time and effort taken to set-up and construct each command may vary considerably. The time taken by storage hardware to complete each request may also vary, even if requests are identical, however this is often beyond the control of the rest of the system.

NUMBER OF DEVICE INTERRUPTS The number of interrupts from a storage device gives a similar measure of predictability to the number of commands sent, and the two values are often heavily correlated, however measuring interrupts shows a more direct indication of how the device is affecting the system, rather than how the system is interacting with the device. The difference in time between a command and associated interrupt can also give an idea of both efficiency and predictability. Additionally, for a real-time system, context-switching to an interrupt service routine during execution of another task can be a significant consideration, meaning the frequency and consistency of interrupts is an important metric for predictability.

I/O OPERATION SPEEDS This metric is a more standardised way of measuring the number of operations that a storage device can service over a given period of time, effectively combining the number of commands sent to a device and interrupts received from it, but often specified at a higher level. Measuring the time taken for I/O operations to complete can give a strong indication of both efficiency and predictability, when comparing different systems or when measuring variation over time. The specific operations being performed must be taken into account when using this metric. This metric is typically expressed as the number of Input/output Operations Per Second, or IOPS.

Higher-order metrics

RATIO OF ONE METRIC TO ANOTHER Combining two related metrics into a single ratio can give more information about a system than either individual number, while still retaining a single value for comparison. For example, a ratio of speed to CPU usage combines the two values to give a broader measure of transfer efficiency than either individual value provides, showing an indication of how much CPU effort it takes to transfer a given amount of data. While this can often be more useful than using a component metric individually, it should be taken in the context of its constituent parts in order to properly evaluate values based on measured results.

VARIATION OF VALUES OVER TIME The variation in value of a metric over time can give an indication of the predictability of a system, independent of its raw performance. For example, this could be used to differentiate two systems with similar long-term average results but vastly different moment-to-moment performance profiles, or to analyse how accurately the properties of a future transfer can be predicted. Any conclusions drawn from this data should take into account the specific source metrics and how they are measured over time. This metric can be measured statistically using variance and standard deviation values.

DISTRIBUTION OF VALUES OVER TIME Similar to the variation of values over time, the distribution of these values and any temporal patterns identified through their measurement can show useful indications of predictability, as well as potentially helping to provide explanations for individual observed measurements. This metric can be particularly useful for events that happen instantaneously, such as command submission and device interrupts, but can also be applied to values measured across time.

2.11.2 Benchmarking metrics

In order to set requirements and measure quantitative results of work in an appropriate context, and for the motivation of analysis and development work, a number of potential applications are identified for targeting. These involve a mixture of data-intensive embedded and high-performance computing applications, chosen in order to investigate a variety of scenarios that could benefit from improvements to the storage stack. It is important for benchmarks to be (to an extent) representative of real-world system use, but also not to entirely be fitted too strongly to the targets of the work, so tests in general focus on general patterns that could be applied to a large number of typical application scenarios.

Beyond traditional performance-oriented benchmarking, measuring the predictability of a system in a real-time context is a valuable metric. While performance benchmarks may acknowledge variability of results in their measurements, this may not be enough to properly show the full extent of system predictability. In this regard, new benchmarks are required that take into account the characteristics of a real-time system, and measure results based on these.

2.12 Summary

In summary, there is a definite consensus in both academic literature and industrial contexts that operating system storage stacks require changes in order to fully take advantage of emerging high-speed

storage technologies. Currently, a small amount practical work has been carried out to address these issues, with some experimental platforms being developed, as well as new interfaces such as NVMe being introduced to standardise an efficient means of communicating with high-speed solid-state storage. However, limited work has been done to address some of the more fundamental areas of the storage stack that are causing overheads in software, such as file systems and the VFS, particularly in the context of real-time and embedded domains.

The hardware acceleration of operating systems is a topic that has been considered previously, with some work being carried out on the acceleration of file systems and other areas of the storage stack, however this has not been fully combined with the need for predictability and efficiency required by real-time embedded systems. The increasingly high relative software latencies caused by faster storage motivates this further with a need to alleviate the additional stress currently being put on CPUs, while modern programmable hardware platforms and synthesis tools make the introduction of accelerator cores into a system more practical than ever.

In order to effectively evaluate any further research into storage systems, strong attention must be given to appropriate benchmarking and profiling approaches, which should be informed by literature and standards established within the community. The broad range of technologies, both new and old, relevant to real-time systems, embedded systems, operating systems and storage systems, must also be understood and considered in order to contextualise and inform future development of engineering and research outputs.

3

Analysis of Storage Operations in Embedded Linux Systems

In order to potentially improve upon methods of accessing storage within real-time embedded systems, it is necessary to thoroughly understand and measure how storage access is currently performed within a common general-purpose operating system. This chapter provides an analysis of storage operations in GNU/Linux[1], with a focus on areas relevant to real-time embedded systems. The internals of the operating system are examined both empirically and analytically in how they create a bridge between an application and various storage devices, in order to both motivate and provide context for any necessary improvements.

3.1 Introduction

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 and interfaces can potentially still operate at full speed, causing this balance to shift, and leading to the observation of performance bottlenecks caused by the operating system software rather than the speed of storage devices themselves. This increased relative time spent in software also emphasises any timing variance caused by software routines, especially with faster and more predictable storage devices, such as modern SSDs.

This chapter 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

1 Hereafter referred to as simply 'Linux', as this work focuses mainly on the operating system kernel and less on any GNU project software running on top of it.

limitations and into software operations. Performance and profiling results are presented 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. The predictability of storage operations is also considered, through observing and measuring the variability of their execution.

3.1.1 Chapter Outline

Initial results from profiling I/O operations are presented in Section 3.3, demonstrating the complexity of the block storage and file system layers in Linux. These are followed by a more detailed analysis though further benchmarking and profiling experiments on a general-purpose server platform in Section 3.4, and an embedded platform in Section 3.5, showing that storage operations experience bottlenecks caused by CPU limitations rather than the speed of the storage hardware when 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, however speeds are still ultimately limited by CPU capability.

Following this, the implementation of a custom timing and profiling hardware component is outlined in Section 3.6, which is used for further experimentation on low-level storage timing on an embedded platform. Experiments focussing on the speed and variability of storage operation timings in periodic tasks are presented in Section 3.7, demonstrating the differences that file systems and storage access methods can have on these metrics. Finally, results from low-level storage operation profiling are presented in Section 3.8, showing in greater detail the breakdown of where time is spent during various methods of accessing storage.

Potential solutions presented in Section 3.10 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 specific hardware acceleration for storage, may reduce the negative effects of CPU limitations on storage speeds. A number of these solutions are explored further in Chapter 4, or discussed as potential future work in Chapter 5.

3.2 Background and Motivation

Traditionally, access to persistent storage has been orders of magnitude slower than volatile system memory, especially when performing ran-

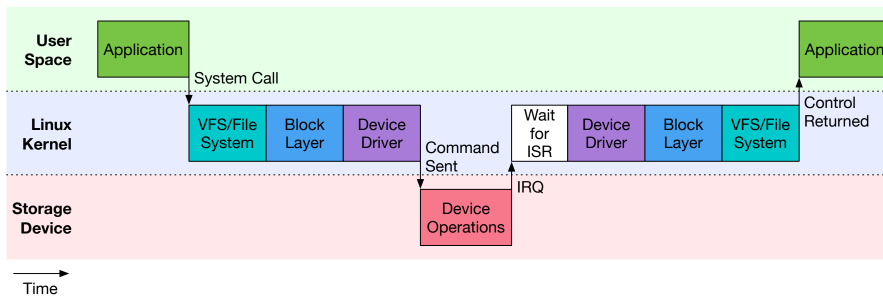


Figure 3.1: Basic control flow of standard file operations in Linux

dom 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 (as discussed in Section 1.3), the main bottleneck associated with accessing non-volatile storage in a general-purpose system is still typically the storage device itself.

3.2.1 Linux Storage Access

Linux (similar to many other operating systems) uses a number of methods to reduce the impact that slow storage devices cause on overall system performance, through its VFS, block layer and scheduler, and low-level device driver framework. 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 at an operating system level 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. Figure 3.1 shows the basic control flow of accessing a file in Linux, from a user-space application, through the VFS, file system and block layer in the kernel, to the storage hardware device.

The Linux attitude to storage access differs from some other, otherwise quite similar, operating systems, such as FreeBSD, which treats all storage devices as raw, uncached character devices, and whose documentation describes the use of cached block devices in other UNIX systems to be "almost unusable, or at least dangerously unreliable" [117].

For most general-purpose Linux systems, 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. A number of examples exist where fast and reliable access to storage is required by a Linux-based embedded system, which may be limited by CPU or memory resources when standard file system operations are used. 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. Such applications can be found in areas including autonomous vehicles, where a large amount of data must be processed and stored reliably and with real-time constraints.

3.2.2 Buffered I/O versus 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 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 [118], 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 3.2 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.

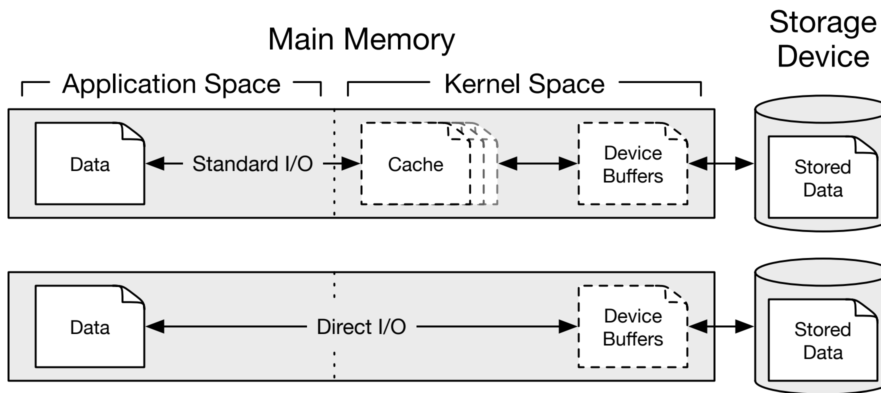


Figure 3.2: The operation of direct I/O in the Linux kernel, bypassing the page cache

A further method of accessing a block device in Linux is to bind it to a 'raw device' node, creating a character device that bypasses the block layer caches and buffers for zero-copy I/O. Raw devices are generally considered to be superseded by the `O_DIRECT` file open flag, which performs essentially the same function, and at one point were considered for removal from the kernel [119], however later had their deprecated status reversed [120].

3.2.3 Linux Storage Stack Complexity

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 typically be copied a minimum of four times on its path from network to disk:

1. From the network device to a buffer in the kernel's networking subsystem
2. From the kernel buffer to the application's memory space
3. From the application's memory space to a kernel buffer
4. From the kernel 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 processing and 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. Limited system memory can also contribute to this effect, as cache sizes will be more restricted.

3.2.4 Issues with Direct I/O

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" [121], which may be interpreted differently (or not at all) by various file systems and kernel versions.

3.3 Preliminary Profiling Tests

In order to identify areas of the Linux kernel that may cause latency during file system operations, and to test the basic capabilities and limitations of profiling tools, a preliminary investigation was conducted into the operations involved in a basic file copy.

A simple C program was written in order to create profiling information for a file copy operation, which involves repeatedly reading data from a file into a buffer, 4 KiB[2] at a time, then appending the contents of this buffer into a new file. Source code for the test program can be found in Appendix C.1.

2 Throughout this thesis, the units KiB, MiB and GiB are used to refer to 2^{10}, 2^{20} and 2^{30} bytes respectively, and KB, MB and GB are used for 10^3, 10^6 and 10^9 bytes.

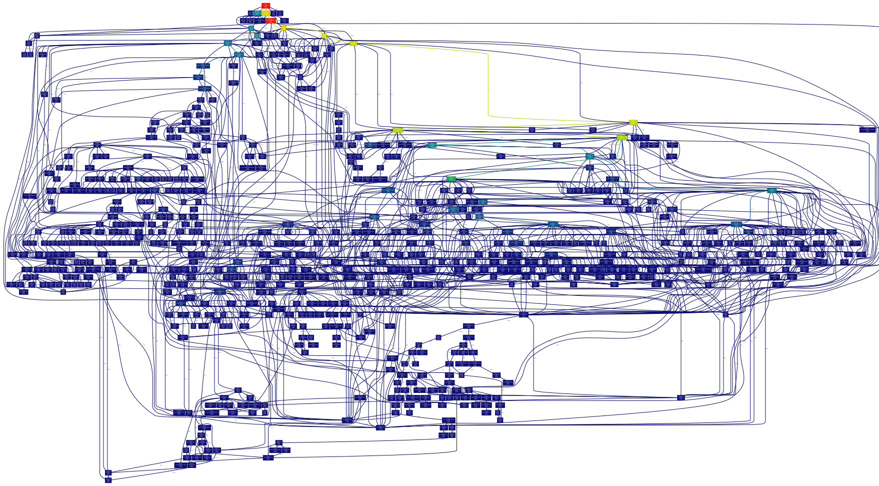


Figure 3.3: Full call graph of the file copy test – nodes represent functions and edges represent calls

3.3.1 Experimental Set-Up

Experiments were performed on the server platform detailed in Appendix B.1 – containing an Intel Core 2 Quad Q9450 CPU at 2.66GHz, 4 GiB main memory, and running Debian Linux 7.6 with a custom-built 3.17 kernel (with profiling support enabled and relevant file systems compiled into the kernel image instead of included as loadable modules). The test application was compiled using *EGLIBC* version 2.13 and *GCC* version 4.7.2, with symbol information retained and compile-time optimisation disabled (using GCC flags `-g -O0`).

3.3.2 Profiling Results

The copy program was run multiple times on a 2GiB file containing random data, with all operating system caches being cleared between runs. During execution, CPU cycle count in each function of the program and operating system was monitored by the full-system profiler, `operf`, part of the *OProfile* suite of tools [122]. The output from the tools showed around 99% of execution time was spent in kernel code (including file system code). A sample of the raw profiling tool output can be seen in Appendix D.

Within the kernel, just over 10% of total execution time was spent in the 'copy_user_generic_string' function, which is used to copy data to and from user-space memory. This shows one potential inefficiency of the traditional storage stack, as data read from storage must be copied between kernel- and user-space memory for an application to use it. As storage device speeds increase, and certain applications demand increasingly large-scale data transfers, this could become a potential bottleneck when dealing with storage.

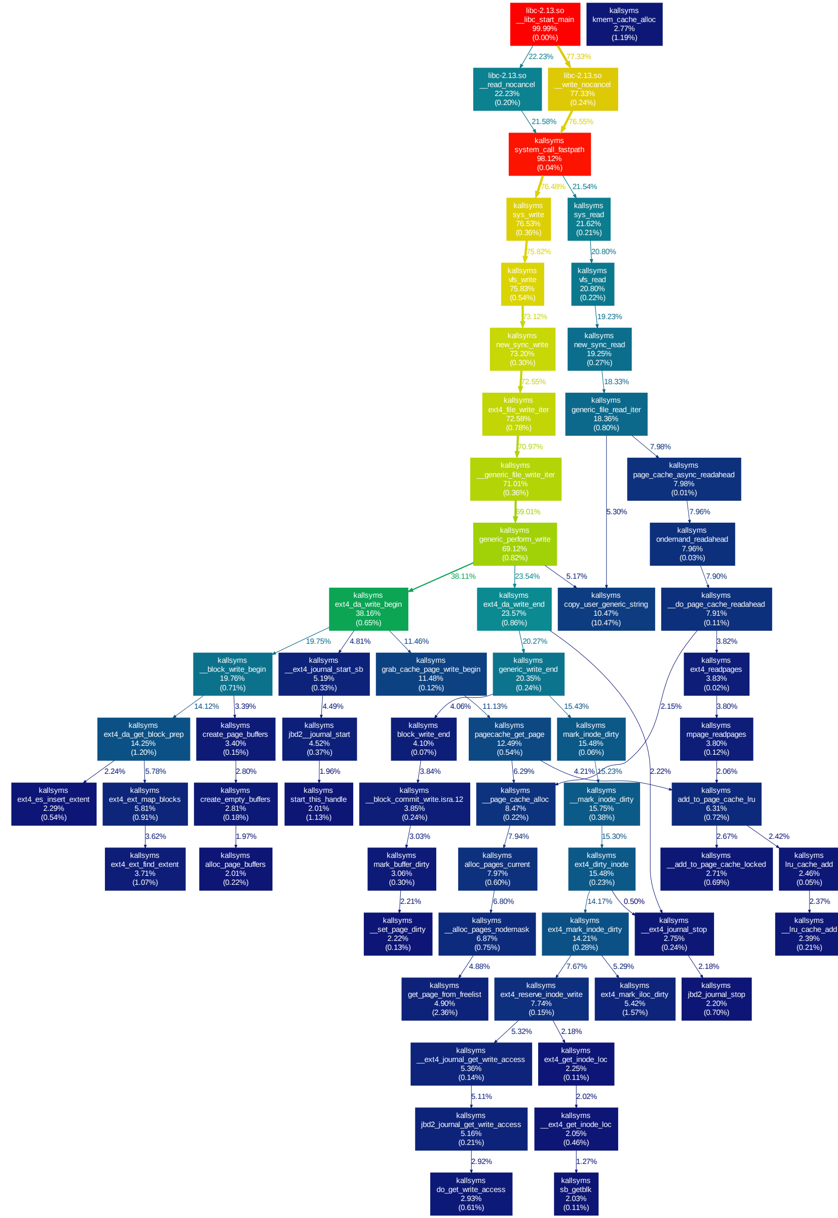


Figure 3.4: Limited call graph of the file copy test (with only nodes accounting for greater than 2% of total execution time shown)

The call graph recorded during execution shows the overall complexity of the storage stack within the kernel and system libraries. Figure 3.3 shows the entire call graph of the file copy program, demonstrating the extreme extent of the code that is actually executed, while Figure 3.4 highlights how little of the executed code performs any serious processing, displaying only the functions that are executed for more than 2% of the total execution time[3].

These initial results motivate a further, more-detailed investigation of the operation of the Linux storage stack in order to examine these complexities further, which is detailed in the remainder of this chapter.

3.4 Measuring Storage Performance

In order to examine the performance of storage devices within Linux at a greater level of detail, and to identify potential bottlenecks present in the Linux storage stack, a number of experiments were performed with storage operations while collecting profiling and system performance information. The first set of detailed experiments were performed on the same standard server platform as the preliminary tests (see Appendix B.1).

To provide a range of results, three storage devices were tested with the system: a Western Digital Blue 500GB hard disk drive [123], and an Intel SSD 535 240GB [124], both connected though a PCIe StarTech SATA III RAID controller card [125]; and an Intel SSD 750 400GB [16] connected directly through PCIe. While both the RAID controller card and Intel SSD 750 use the same PCIe interface for their physical connection to the system, the RAID controller uses SATA III to connect to the storage devices themselves, and AHCI for the logical storage interface, whereas the Intel SSD 750 uses the more recent NVMe standard.

Due to limitations of the PCIe controller on the server motherboard, the speed of the SSD interface is limited to PCIe Gen. 2 x4 (from its native Gen. 3 x4), reducing the theoretical maximum four-lane bandwidth from 3940 MB/s to 2000 MB/s. While this is still significantly faster than the 600 MB/s theoretical maximum of the SATA III interface used by the HDD and SATA SSD, it means the SSD will never achieve its advertised maximum capable speed of 2200 MB/s in this hardware set-up, although this speed is likely to be calculated theoretically, and unlikely to be reached in any practical situation.

3 These call graphs are not intended to be fully readable, and are just used as a visual representation of call stack complexity.

3.4.1 Read and Write Benchmarking

To determine an indication of the operating speeds of the storage devices at various block sizes with minimal external overhead, the Flexible I/O Tester (FIO) benchmarking utility [126] was used to perform basic sequential and random read and write tests. Sequential read and write results from FIO were validated using the standard Linux `dd` utility (using `/dev/zero` as a source file for writes and `/dev/null` as a destination for reads), which demonstrated that the benchmarking results closely matched the real-world tool. Both storage devices were freshly formatted with an *ext4* file system before each test, using the default options to best reflect common usage (other than disabling lazy inode table and journal initialisation).

For each block size, storage device and access type (sequential or random), four tests were performed: reading from a file on the device, writing to a file on the device, reading and writing with direct I/O enabled. Each test was performed with and without the collection of profiling data, so results could be gathered without any additional overheads caused by these measurements.

For each benchmark, speed and resource usage data was recorded for 20 GiB transfers, along with the variability across the duration of the transfer. 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 recording data for different access types and transfer block sizes is necessary to gain a full understanding of performance.

3.4.2 Benchmark Performance Results

Read and write speeds generally correlate with what is expected from the storage devices, with the HDD being slowest, followed by the SATA SSD, then the NVMe SSD performing fastest, as shown in Figures 3.5 and 3.6. Small block sizes decrease performance, especially when writing, and for random accesses. Larger block sizes of random accesses begin to align themselves with sequential access speeds, as the sequential nature of the blocks outweigh the overheads of random access. This effect begins with larger block sizes on the HDD than those on the SSDs, showing how its random access performance differs more significantly from sequential access. This is a broadly expected result, due to the physical limitations of mechanical storage preventing efficient random access to data, and solid-state storage operating in a fundamentally different manner. An interesting observation of the SATA SSD is the reduction in write speed as block sizes get large (for those measured, 128MiB and above), suggesting that operating system or device buffers are becoming overwhelmed, showing that

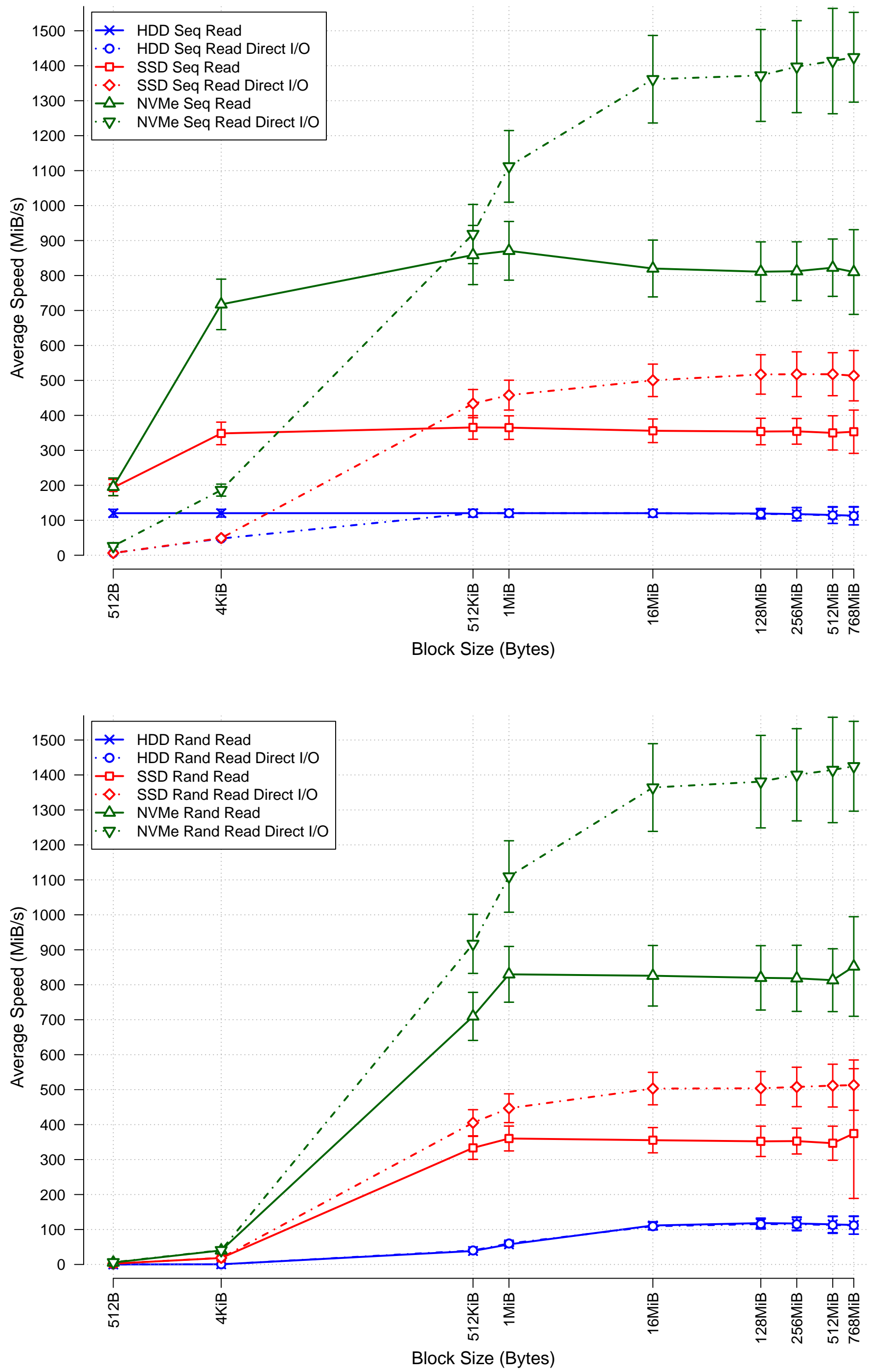


Figure 3.5: Read speeds for HDD, SSD and NVMe SSD on the server platform (x-axis log scale, error bars show standard deviation)

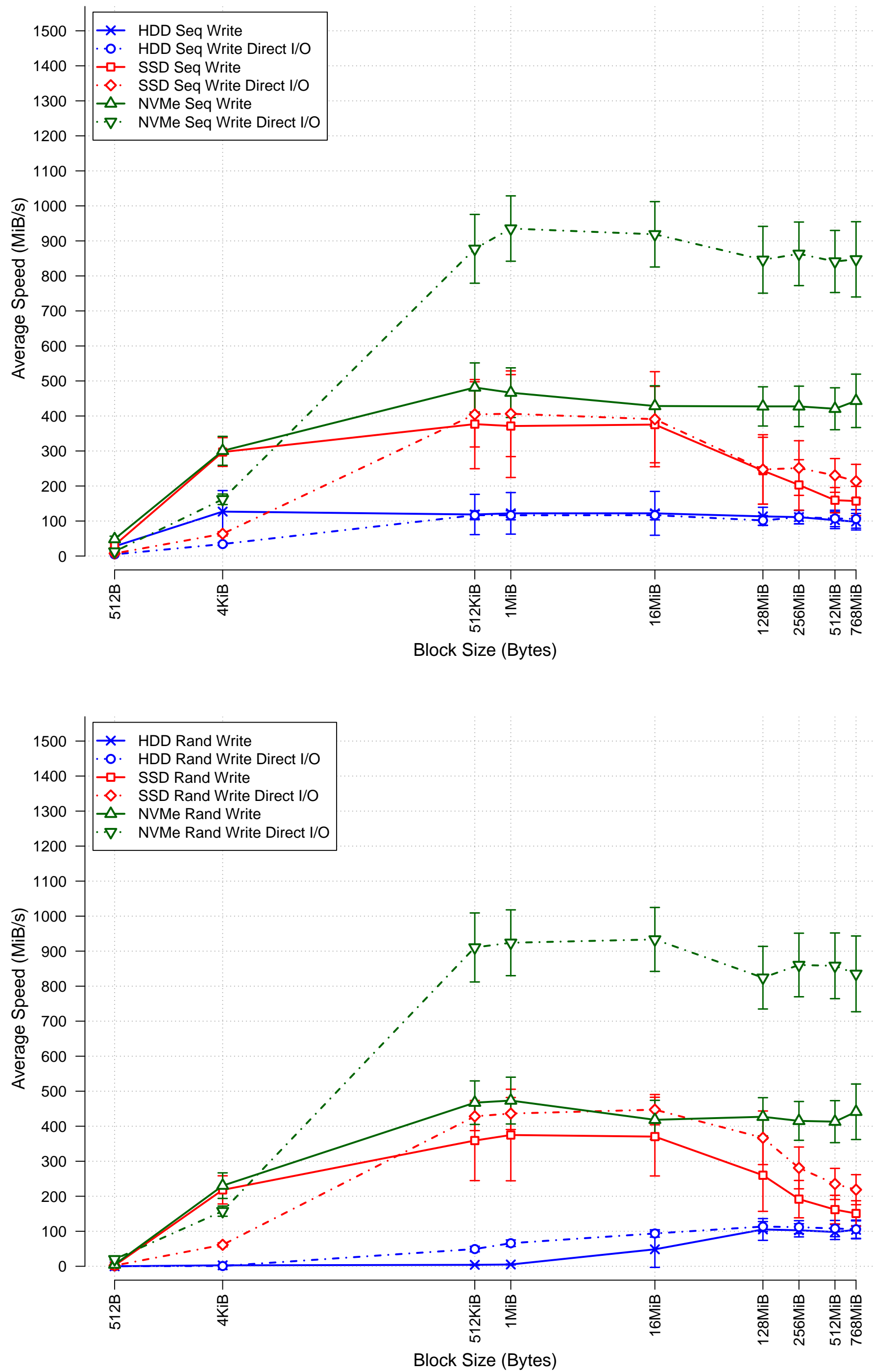


Figure 3.6: Write speeds for HDD, SSD and NVMe SSD on the server platform (x-axis log scale, error bars show standard deviation)

increasingly large transfer block sizes do not necessarily mean higher performance.

When direct I/O is enabled, speeds are generally higher when the block size is sufficiently large to overcome the overheads involved, such as increased communication with hardware due to the lack of buffering and access-coalescing. 512 B and 4 KiB block sizes are significantly slower than the standard write tests for sequential accesses, as the kernel cannot cache data and write it to the device in larger blocks, but this effect is reduced for random access.

Large block sizes show significant speed increases when using direct I/O on the NVMe SSD, suggesting that the bottleneck in performance encountered when performing standard I/O on this device is due to inefficiencies in the operating system, rather than the storage device itself. This contrasts with the HDD, where the storage device is clearly almost always the performance bottleneck, as speeds change very little with large block sizes or direct I/O. The large increase in potential speed when using direct I/O on a fast storage device motivates further investigation into this area, as software efficiency becomes increasingly relevant in storage access performance, even in a relatively capable computer system.

3.4.3 CPU Utilisation

The CPU utilisation measurements taken during the benchmarking process give a further indication on the location of bottlenecks when accessing storage, and are shown in Figures 3.7 and 3.8 (note that error bars are not displayed in CPU measurements as FIO only reports the average CPU usage measured across the entire length of a transfer). Small block size sequential operations are clearly very inefficient for the file system and block layer, showing high CPU usage despite a slow transfer speed. The lower CPU utilisation than this for small random transfers is likely due to the even slower transfer speeds. There is a general trend in CPU utilisation increasing as random read block size increases, in line with the faster performance of storage devices, and therefore less time spent idle waiting for the device to return data.

Direct I/O CPU utilisation is almost always less than the standard I/O equivalent, however some counterexamples to this are seen in various small block size transfers, highlighting why direct I/O is not always a good choice for disk access, even when just considering CPU usage and not speed.

3.4.4 Storage Access Efficiency

To obtain a better overall view of the CPU efficiency for accessing storage, the speed per percentage unit of CPU utilisation can be used,

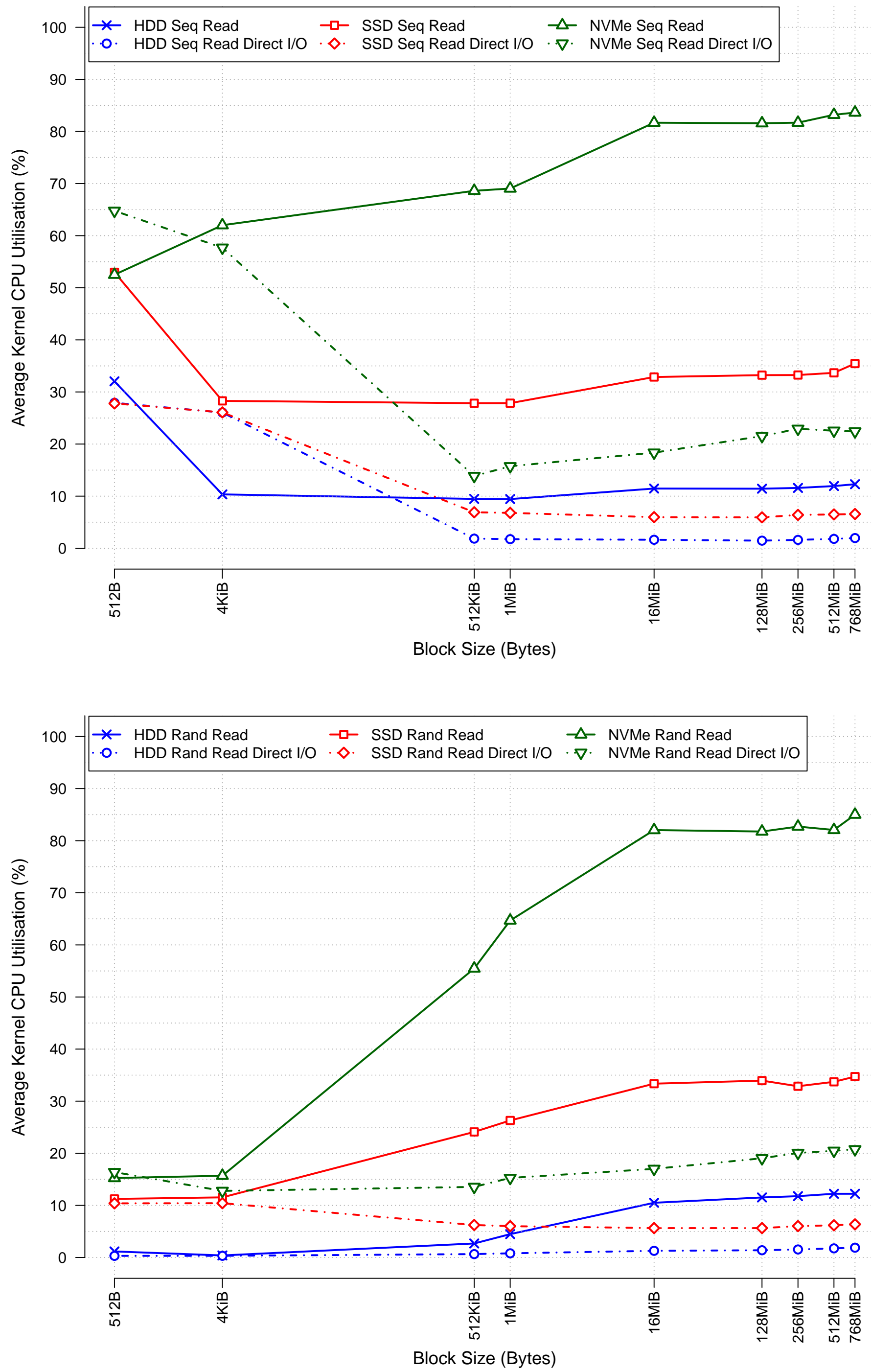
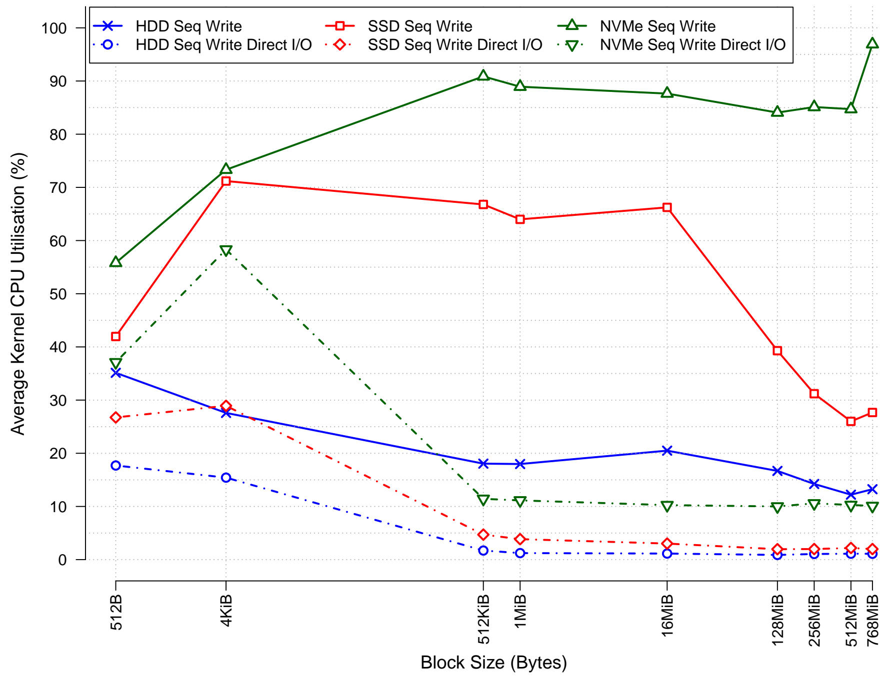


Figure 3.7: Kernel CPU utilisation for HDD, SSD and NVMe SSD read operations on the server platform (x-axis log scale)



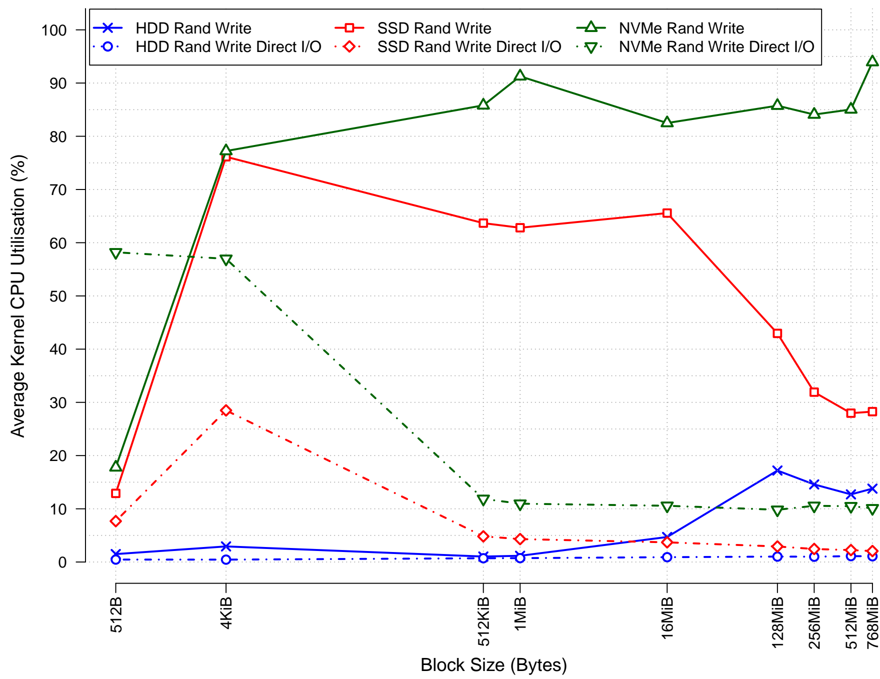


Figure 3.8: Kernel CPU utilisation for HDD, SSD and NVMe SSD write operations on the server platform (x-axis log scale)

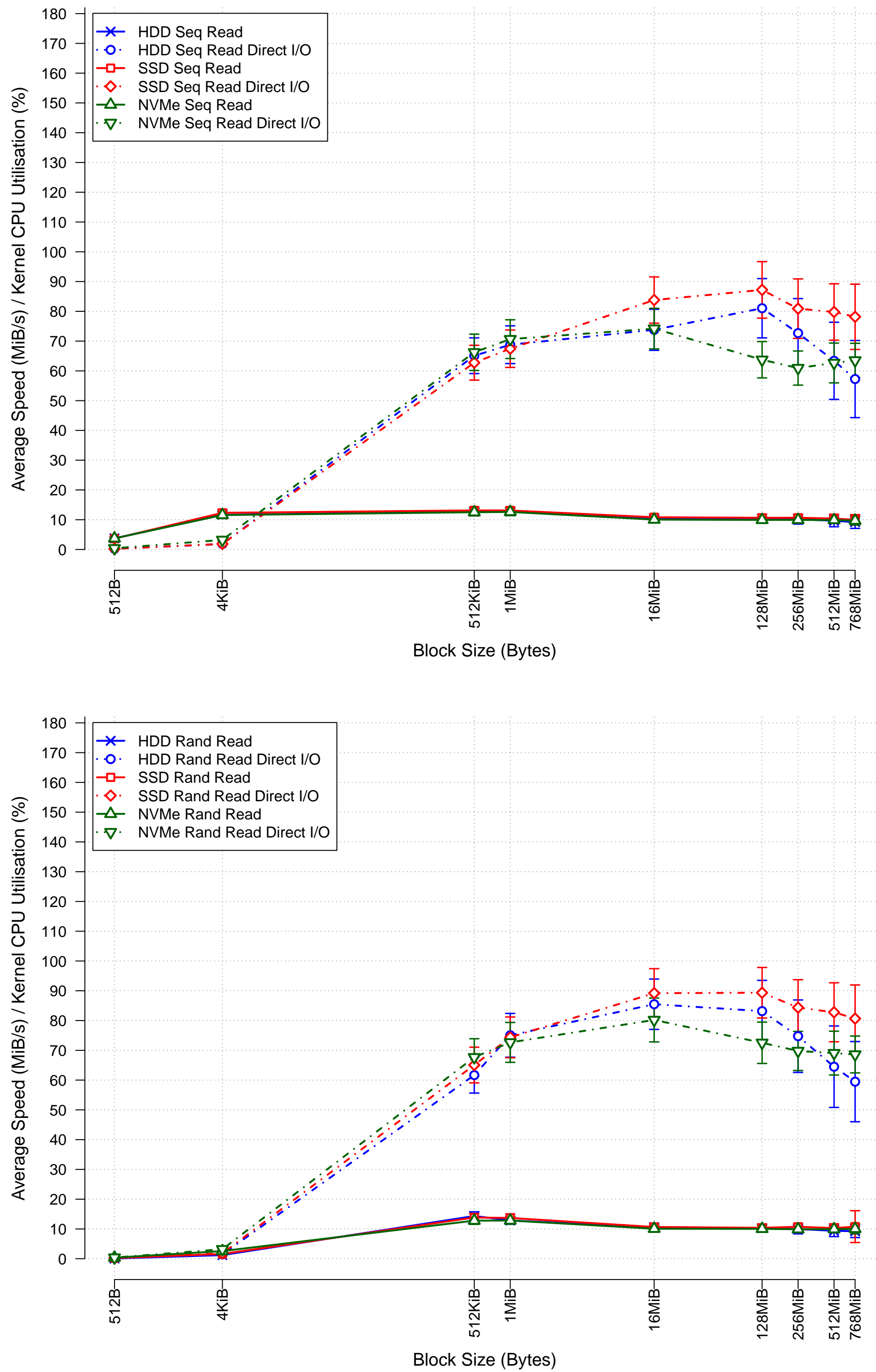


Figure 3.9: Read speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the server platform (x-axis log scale, error bars show speed standard deviation)

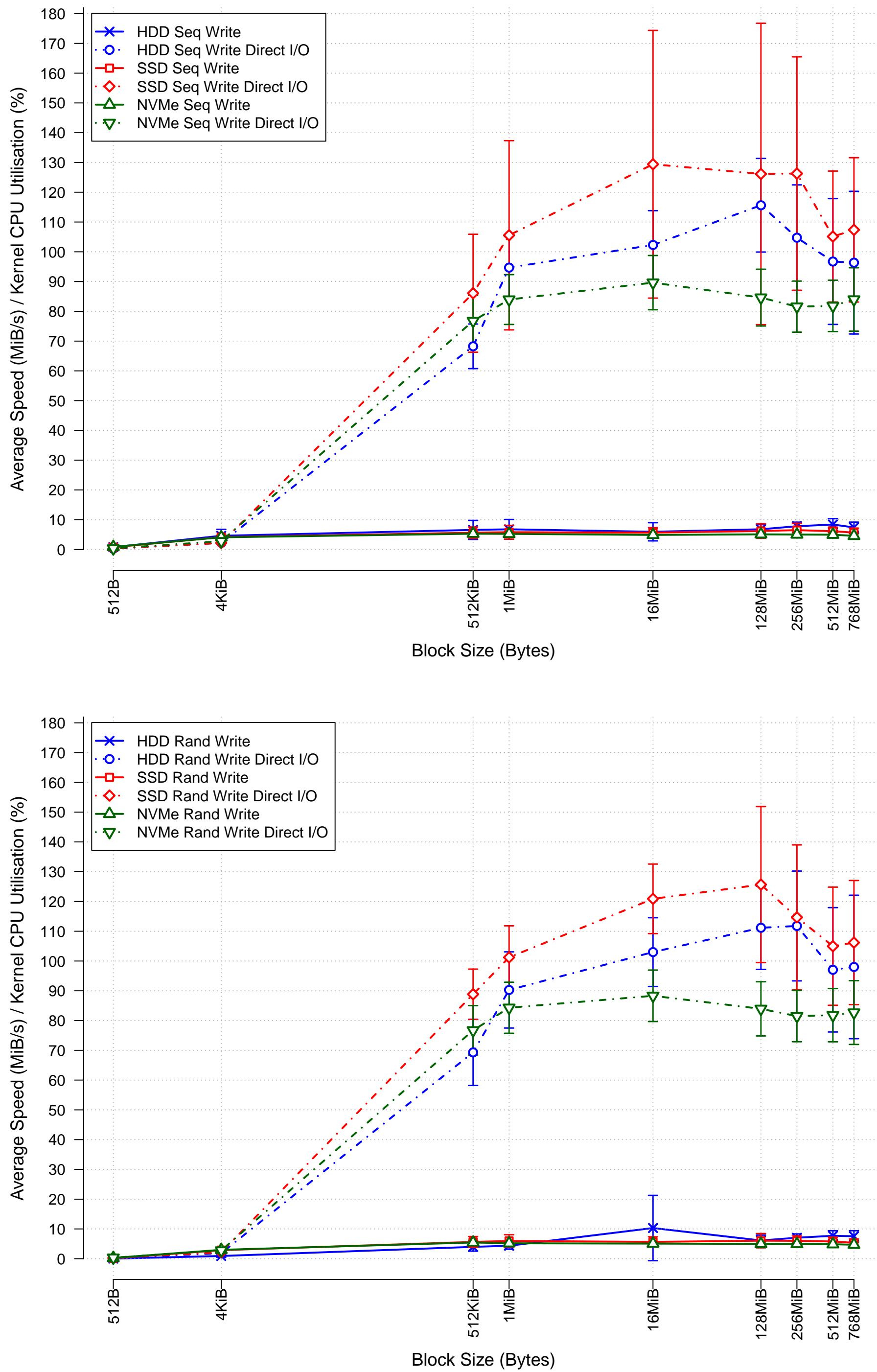


Figure 3.10: Write speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the server platform (x-axis log scale, error bars show speed standard deviation)

giving an indication of the effort required to achieve a given transfer speed. These results are shown in Figures 3.9 and 3.10, for the tests detailed previously.

These plots clearly show the large increase in CPU efficiency offered by direct I/O when transferring blocks of data as small as 512 KiB, as well as the fairly constant CPU cost for standard *ext4* I/O transfers, varying very little with block size, storage interface or storage device type. The general inefficiency of performing many small block size transfers is also further highlighted.

Another trend clear in these plots is that the CPU efficiency of transfers can decrease for large block sizes when using direct I/O. This is likely due to the increased amount of buffer allocation and DMA mapping being forced onto the kernel for the larger blocks, while already being at an optimal speed for the device at lower block sizes.

3.5 Embedded System Storage Performance

In order to examine the effects that a resource-constrained system can have on the performance of storage devices, storage experiments were repeated on an embedded platform, along with a number of additional experiments to further test the embedded hardware.

3.5.1 Experimental Set-Up

Embedded experiments were carried out on an Avnet ZedBoard Mini-ITX development board [43], communicating with storage devices through its PCI Express connector and an AXI-to-PCIe bridge design programmed on the FPGA. The ZedBoard Mini-ITX provides a Xilinx Zynq-7000 system-on-chip, which combines a dual-core Arm Cortex-A9 processor (with a variable clock, up to 800 MHz) with a large amount of FPGA fabric, alongside 1 GiB of DDR3 RAM and many other on-board peripherals. Experiments were run using Linux kernel 3.18 (based on the Xilinx 2015.2 branch). A detailed specification of the experimental platform can be found in Appendix B.2.

3.5.2 Benchmark Performance Results

Figures 3.11 and 3.12 show the average sequential and random read and write speeds for a number of block sizes when transferring data to and from the storage devices.

The standard read and write speeds for all storage devices are clearly slower in general than on the server platform, with the SSDs also only performing slightly faster than the HDD for all block sizes tested. While random access results are slower for the HDD than the

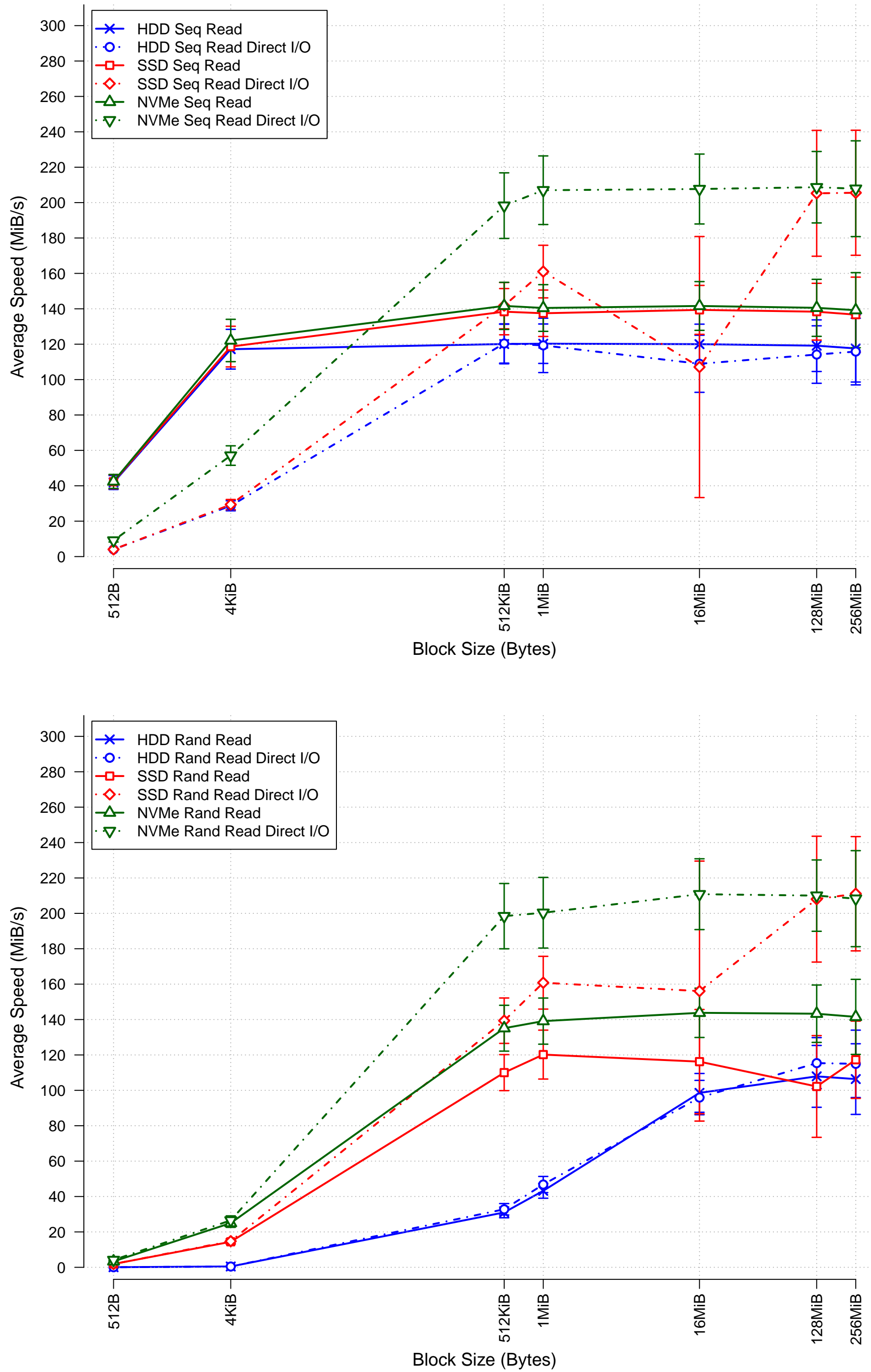


Figure 3.11: Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz (x-axis log scale, error bars show standard deviation)

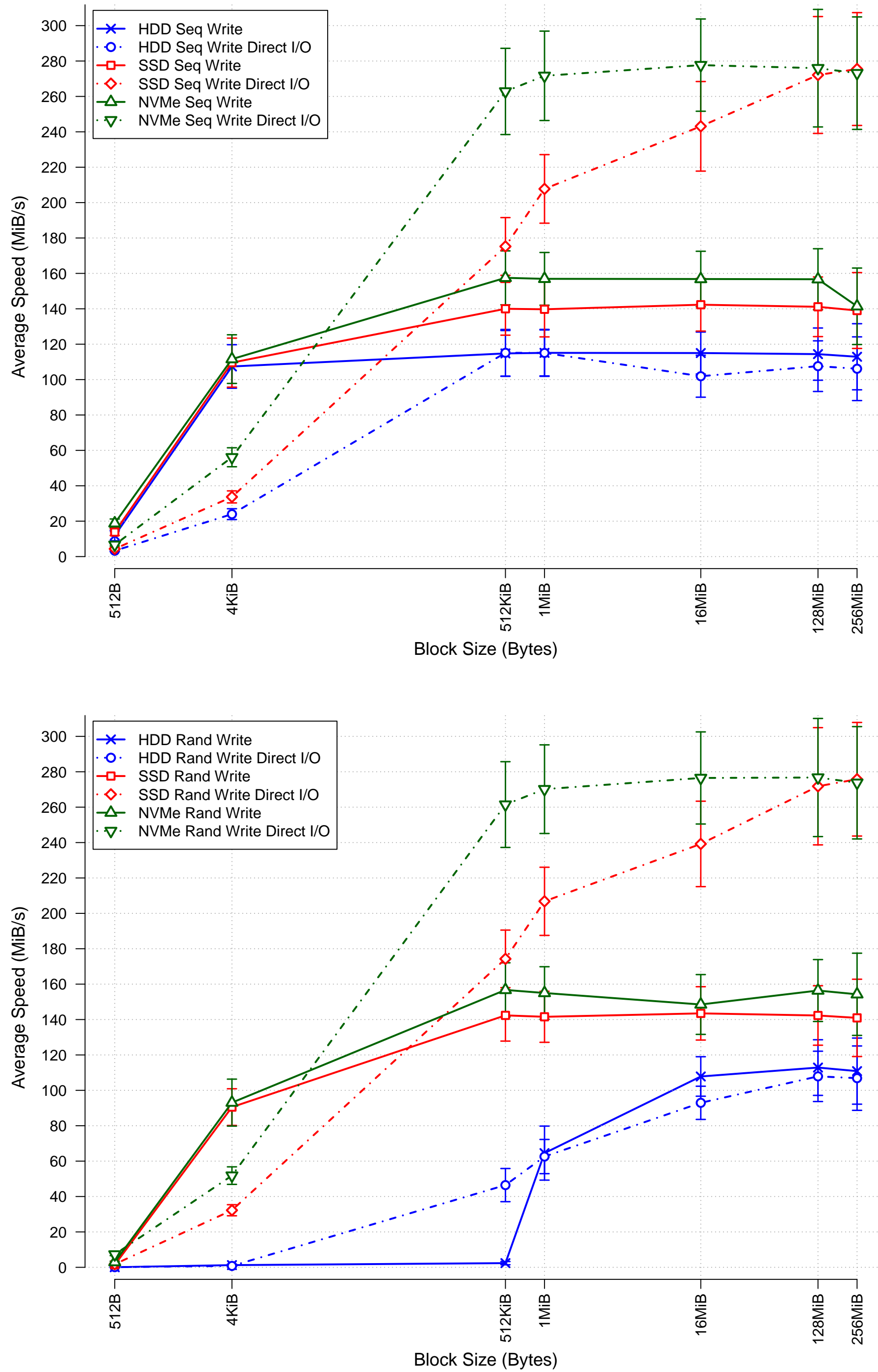


Figure 3.12: Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz (x-axis log scale, error bars show standard deviation)

SSDs, sequential speeds do not vary significantly between the different storage types, with the two SSDs performing almost identically in sequential reads, and the HDD only slightly slower than this. 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, and that the NVMe SSD should perform better than the SATA device, as seen with the server platform.

The overheads of small block transfers are clearly visible, even in sequential tests, with results significantly slower than small block transfers on the server platform, suggesting that these are caused (at least in part) by areas outside the limits of the storage devices themselves. Maximum HDD speeds on the embedded platform are almost identical to those on the server, however, suggesting that the physical limits of the hardware are reached on both when using a suitably large block size, unlike with the SSDs.

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

Additional speed tests performed on a *tmpfs* RAM disk (through a `/dev/shm` device) show far higher read and write performance than all three non-volatile storage devices. This is expected behaviour, 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 giving a more direct route from application buffers to the (virtual) storage device with fewer memory copy operations.

For the SSDs, maximum direct I/O speeds are almost double those of standard I/O, however these are both still far lower than the rated speeds of the device, and than the same tests performed on the server platform. A further bottleneck appears to be encountered before the 16 MiB direct I/O block size on the NVMe SSD, and before 128 MiB on the SATA SSD, suggesting that at this point 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 210 MiB/s read and 280 MiB/s write) also matches the write speed limit of the RAM disk when using block sizes between 512 KiB and 16 MiB, suggesting that both the SSD and RAM disk may be experiencing the same bottleneck here.

There are several points on the plots that go against the general trends of increasing block size improving performance – while these do not affect the validity of the results or suggested conclusions, they

are interesting to consider nonetheless. One example is the falling speed of 16 MiB direct I/O SSD reads in Figure 3.11, which may be caused by specific DMA buffer sizes being problematic for the kernel to allocate, or interactions with the device driver or SSD hardware controller. These results also correspond with a much greater standard deviation, suggesting the decrease in performance is actually attributable to a general increase in variability for that specific test. Results for the 16 MiB block size across different CPU frequencies (available in Figure E.4 in Appendix E.1) show a similar trend, suggesting the anomaly is related to the specific block size rather than other factors. A further unusual result is the low speed of 512 KiB HDD random writes shown in Figure 3.12. Again, this result appears to be inherent to this block size on this device, as the same pattern is present across multiple experiments with various CPU frequencies (shown in Figure E.3 in Appendix E.1). Due to the HDD direct I/O result having a higher speed than this, it can be assumed that this result is caused by an interaction between the kernel and storage device.

3.5.3 CPU Utilisation

Figures 3.13 and 3.14 show the mean system CPU utilisation across block sizes for each test, measured as the sum of the utilisation of each of the two CPU cores (so 100% corresponds to utilisation equivalent to that of an entire single core, and values between 100% and 200% show some utilisation of a second core). 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 utilisation 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.

Peak CPU utilisation generally correlates with peak performance on the SSD direct I/O tests, suggesting that speed bottlenecks are caused by processing on the CPU, especially as this happens as one core is completely saturated by the I/O operation. One clear example of this is the SATA SSD direct I/O write tests, where speed increases with block size until 128 MiB, and CPU utilisation also peaks here at around 100%. Faster results from the server platform, using a faster CPU, also suggest this correlation.

3.5.4 Storage Access Efficiency

The speed per percentage unit of CPU utilisation results are shown in Figures 3.15 and 3.16, giving an indication of the effort required to achieve a given transfer speed.

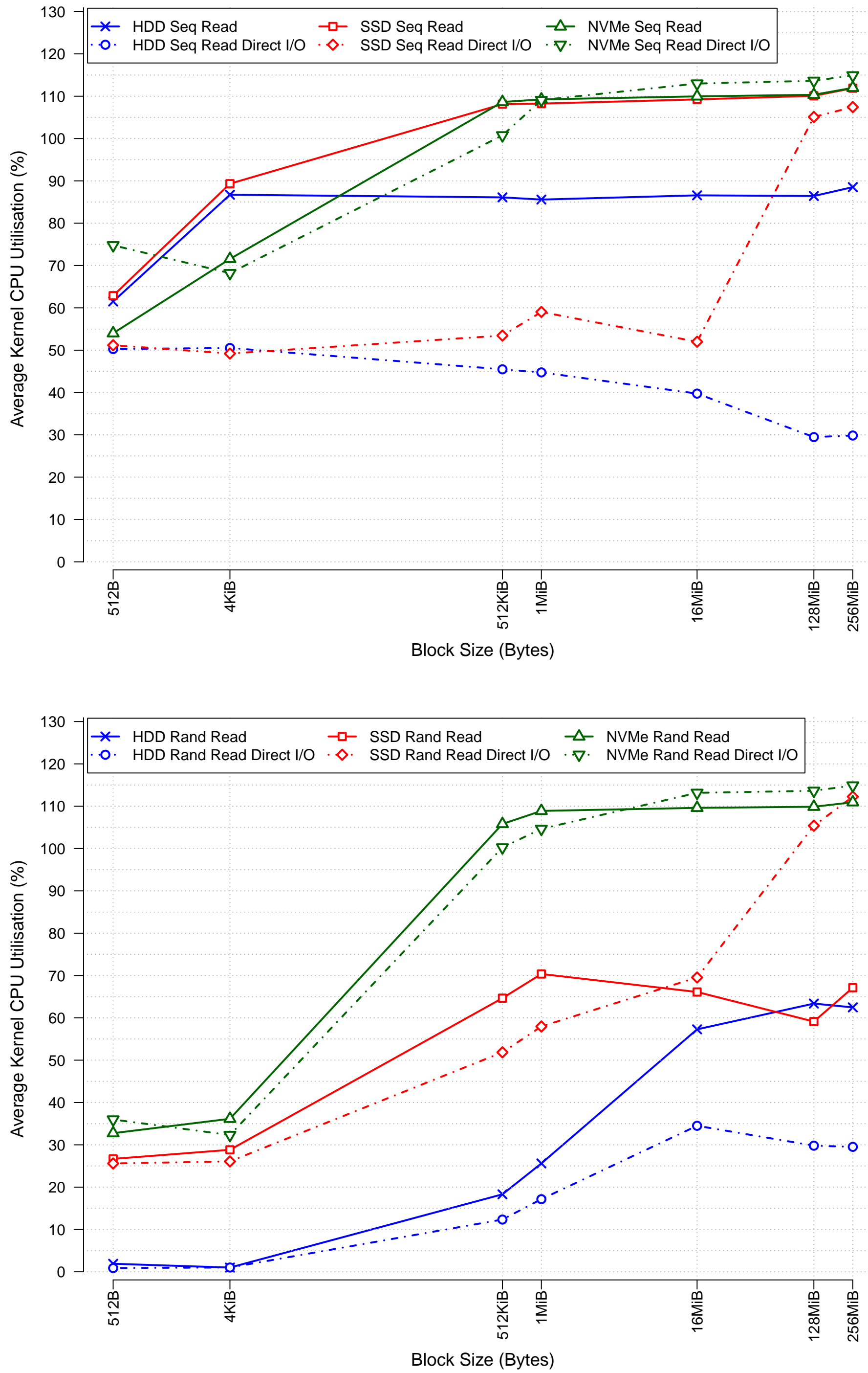


Figure 3.13: Kernel CPU utilisation for HDD, SSD and NVMe SSD read operations on the Zynq-7000 platform at 800 MHz (x-axis log scale; y-axis is sum of per-core values, up to 200%)

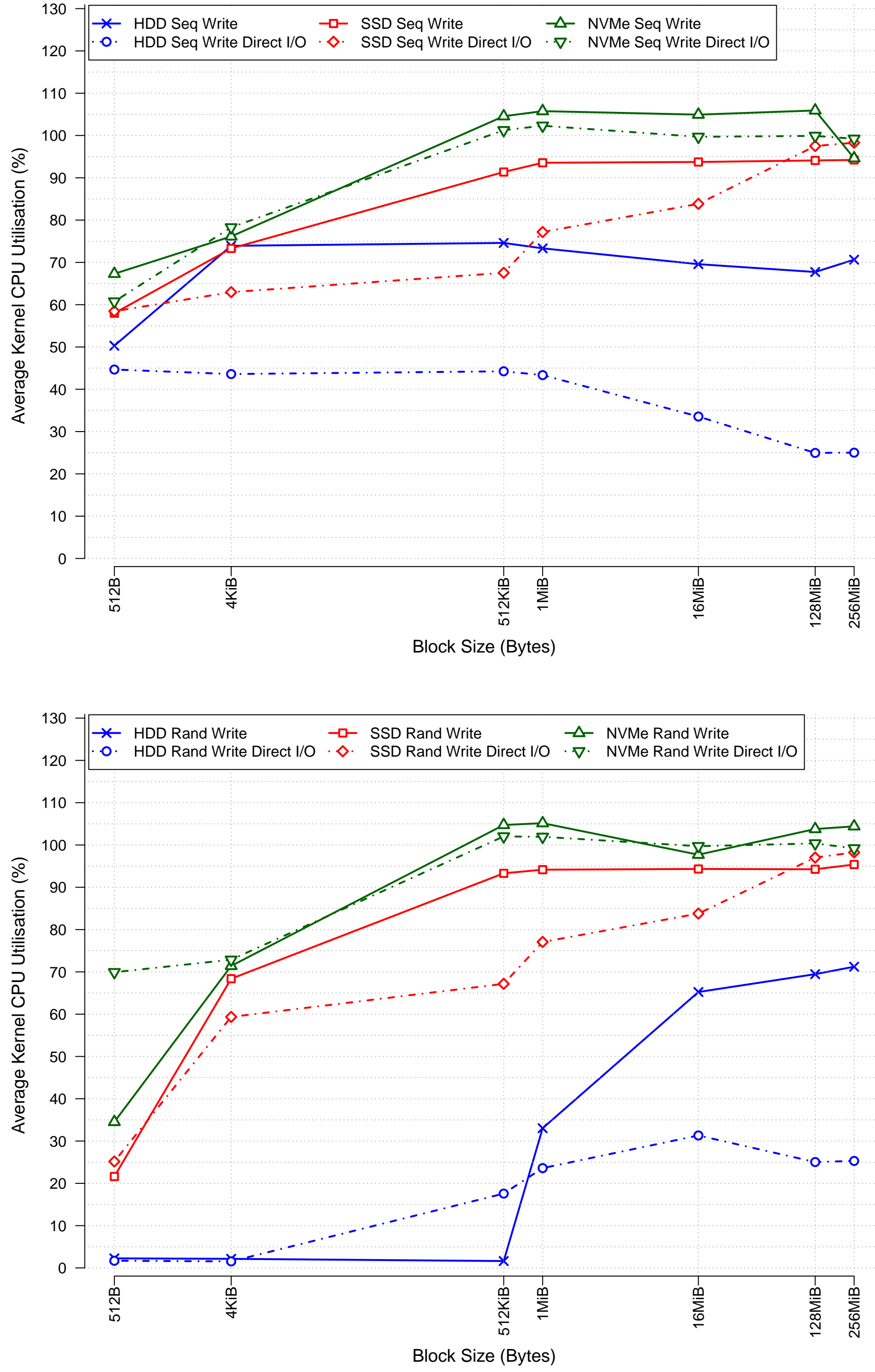


Figure 3.14: Kernel CPU utilisation for HDD, SSD and NVMe SSD write operations on the Zynq-7000 platform at 800 MHz (x-axis log scale; y-axis is sum of per-core values, up to 200%)

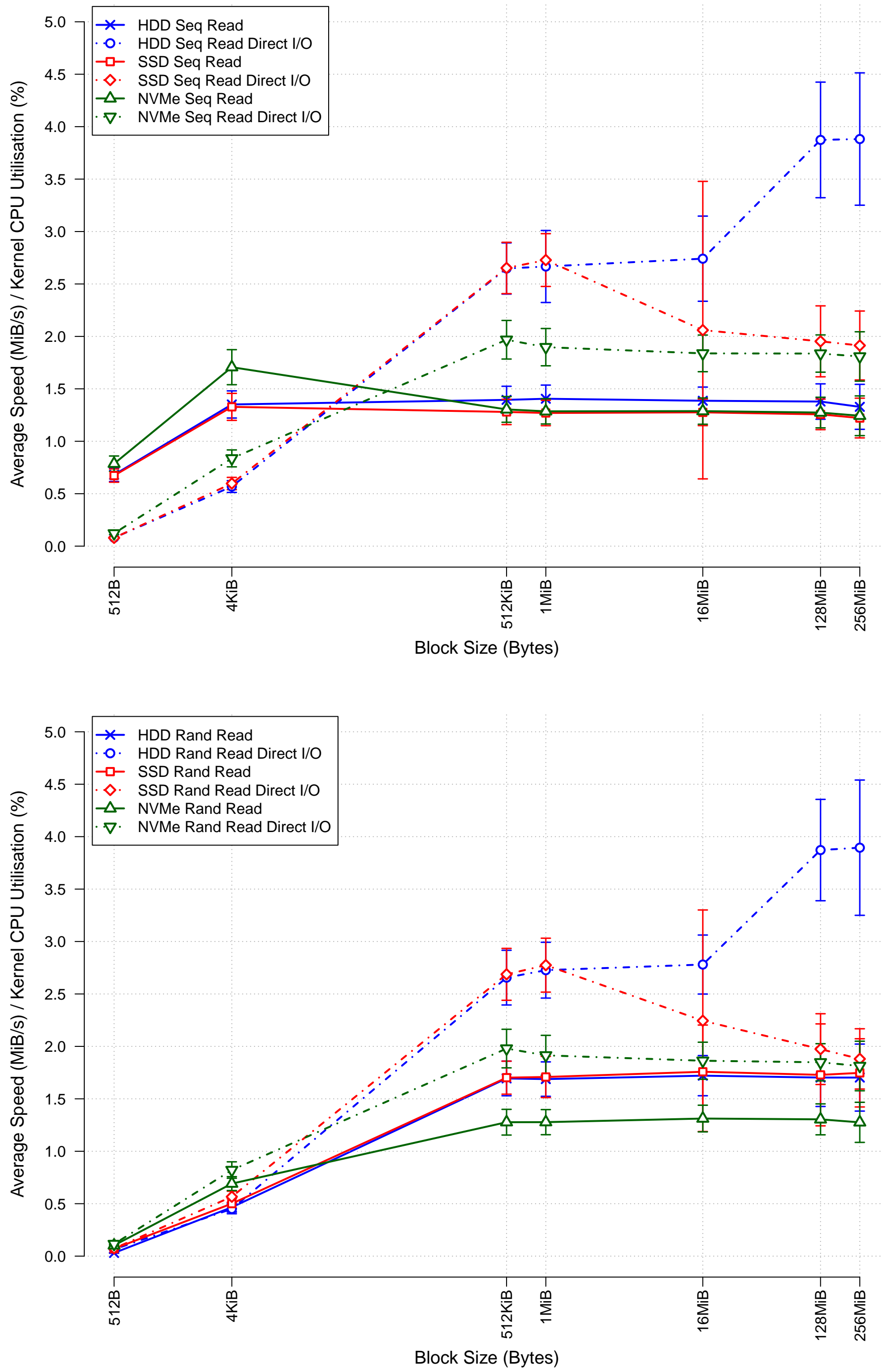


Figure 3.15: Read speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz (x-axis log scale, error bars show speed standard deviation)

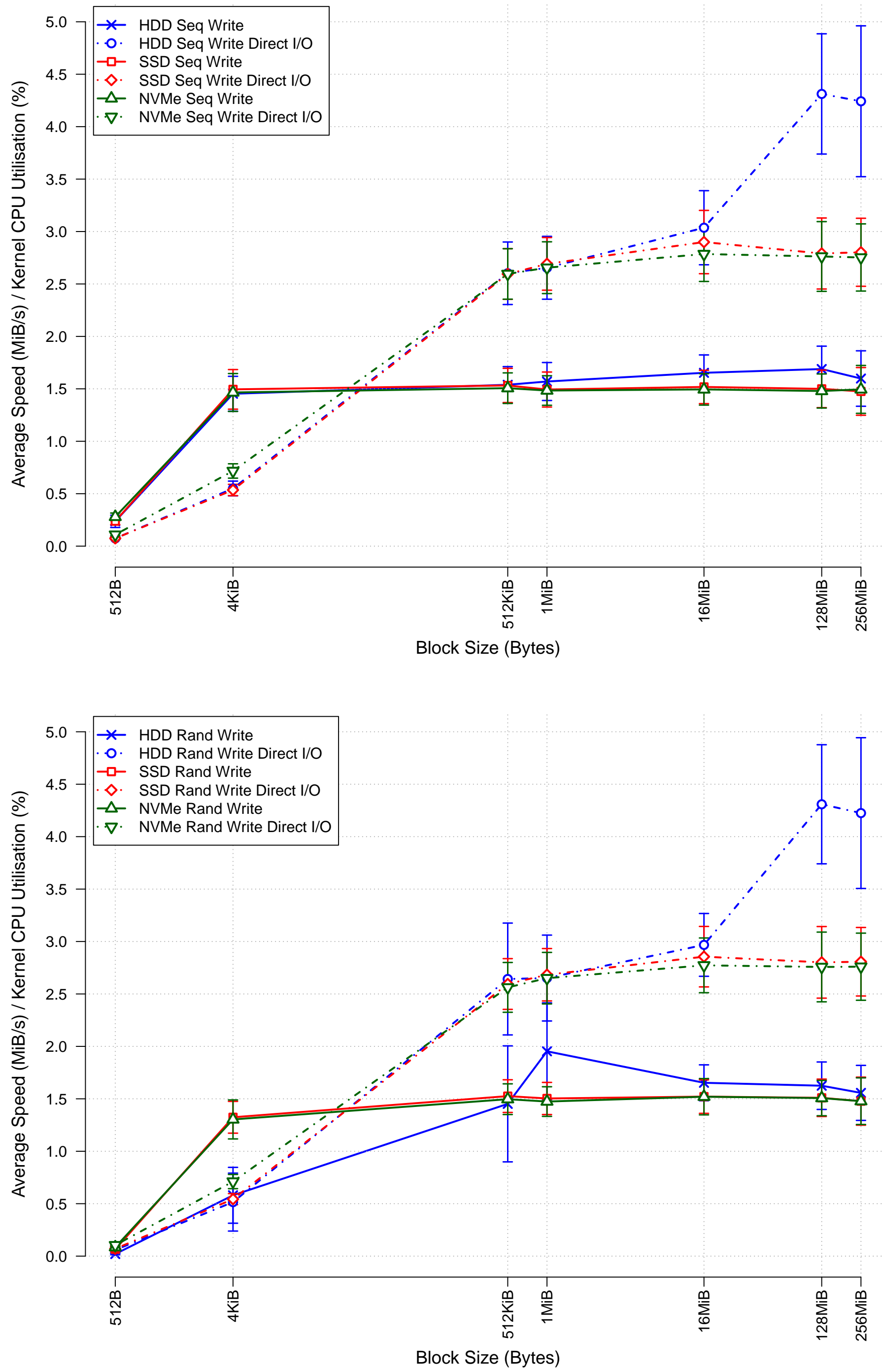


Figure 3.16: Write speed per percent of CPU utilisation for HDD, SSD and NVMe SSD on the Zynq-7000 platform at 800 MHz (x-axis log scale, error bars show speed standard deviation)

Similar to the server experiments, plots clearly show the large increase in CPU efficiency offered by direct I/O when transferring blocks of data as small as 512 KiB, but show standard I/O being more efficient with lower block sizes. The CPU cost for standard *ext4* I/O transfers is fairly constant when compared to direct I/O, but varies more than on the more cable server platform. The inefficiency of performing many small block size transfers is also further highlighted, especially in the random transfer results.

Like with the speed results, there are some results in the CPU data that do not fit the general trend – for example, the unexpectedly low CPU utilisation (and subsequently high efficiency) of large-block direct I/O HDD transfers shown in Figures 3.13 to 3.15. The fact that this efficient CPU usage is consistent across all experiments with these block sizes on the HDD, but differs from the SATA SSD results, means it is likely due to an interaction between the slow speed of the disk combined with the efficiency of large block size transfers, potentially with further improvements facilitated by the I/O scheduler. Note that error bars are quite large for these outlying results, showing a larger variation in speed across the transfers compared to the less efficient, smaller block sizes.

3.5.5 System Profiling

To compliment the data collected by FIO, experiments were repeated while using `operf`, enabling the execution time spent in each function within the user application, kernel and associated libraries to be measured. 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.

Results from profiling show that for both standard read and write tests, a large amount of CPU time is spent copying data between user and kernel areas of memory. Figure 3.17 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.

Standard read, and both read and write 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.

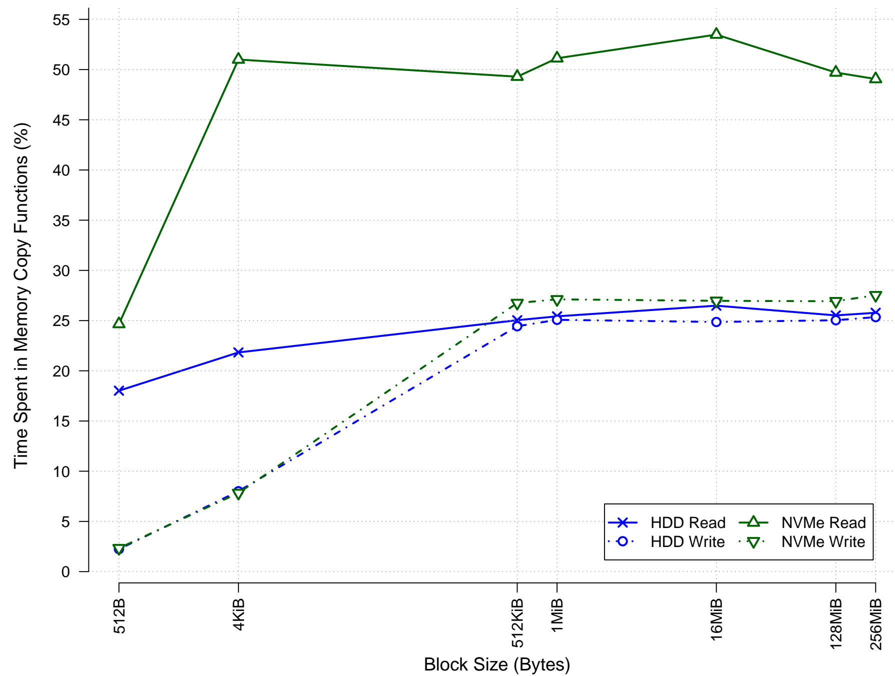


Figure 3.17: Proportion of time spent in memory copy functions for HDD and NVMe SSD with each block size tested (x-axis log scale)

3.5.6 Varying CPU Frequency

To further examine the direct effects of CPU speed on storage bandwidth in Linux, experiments were repeated on the embedded platform using three different fixed CPU frequencies – 200 MHz, 400 MHz and 667 MHz. This work validates the idea that CPU speed has a significant and direct effect on storage speeds, as suggested by the original embedded experiments (which use a fixed 800 MHz clock speed), showing a clear relationship between CPU and storage speed scaling for all storage devices.

Figures 3.18 and 3.19 show a summary of results from the repeated experiments, with the main points indicating the mean speed across block sizes, and the vertical bars indicating the maximum and minimum average speed values from the block sizes tested. It is clear from these results that the faster solid-state storage devices are being fully limited by the CPU, especially when using direct I/O, as their maximum and mean speeds increase approximately linearly with CPU frequency. In contract, for the slower HDD, the bottleneck of the storage device itself can clearly be seen in the maximum speed of both reads and writes at 400MHz and higher.

The minimum speed values for each device also show an increase relative to the CPU frequency, however these are much smaller absolute values than the change across maximums, leading to a far larger absolute range of speeds as frequency increases (as demonstrated by the long vertical bars).

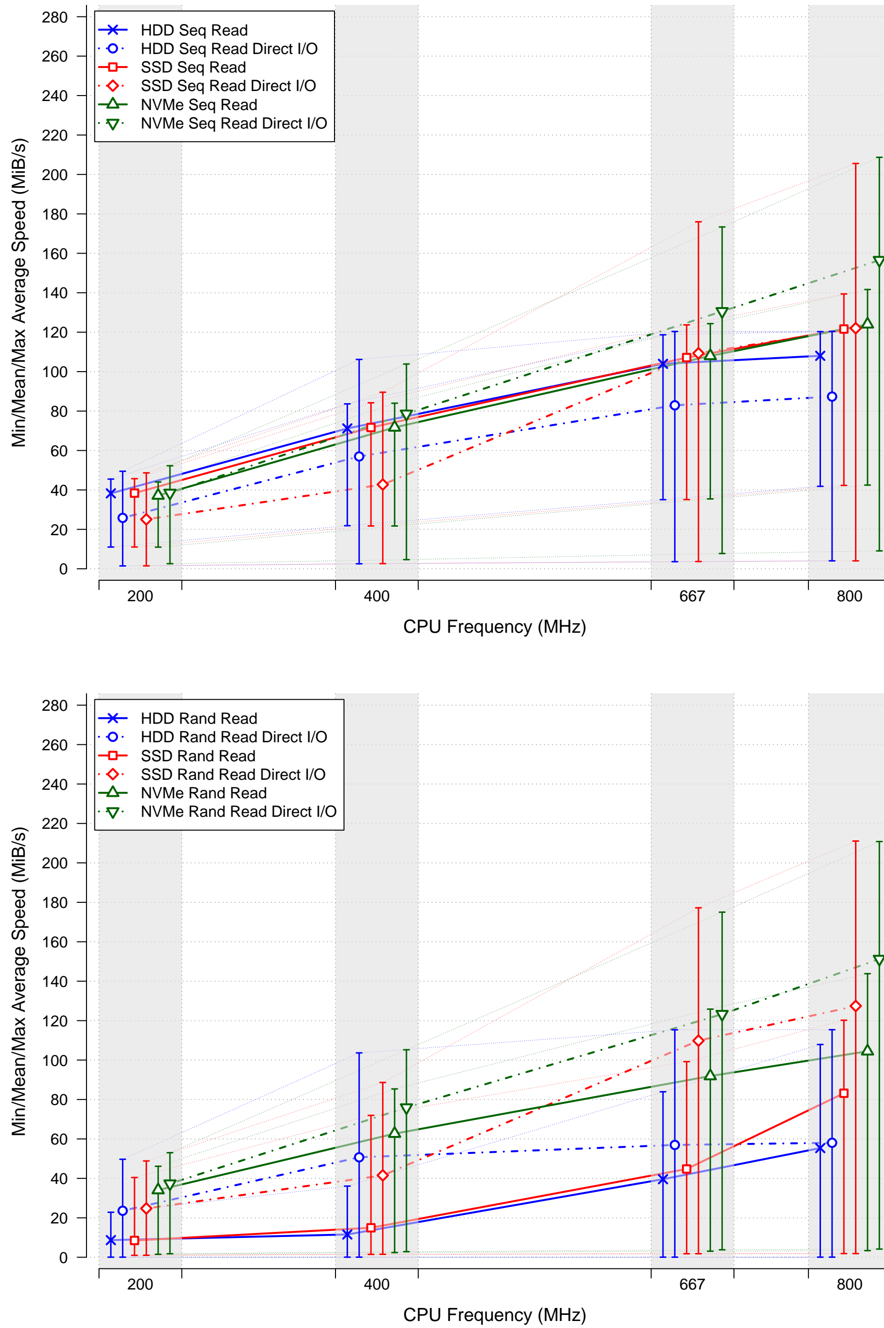


Figure 3.18: Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at different frequencies (points show mean value, bars show minimum and maximum values

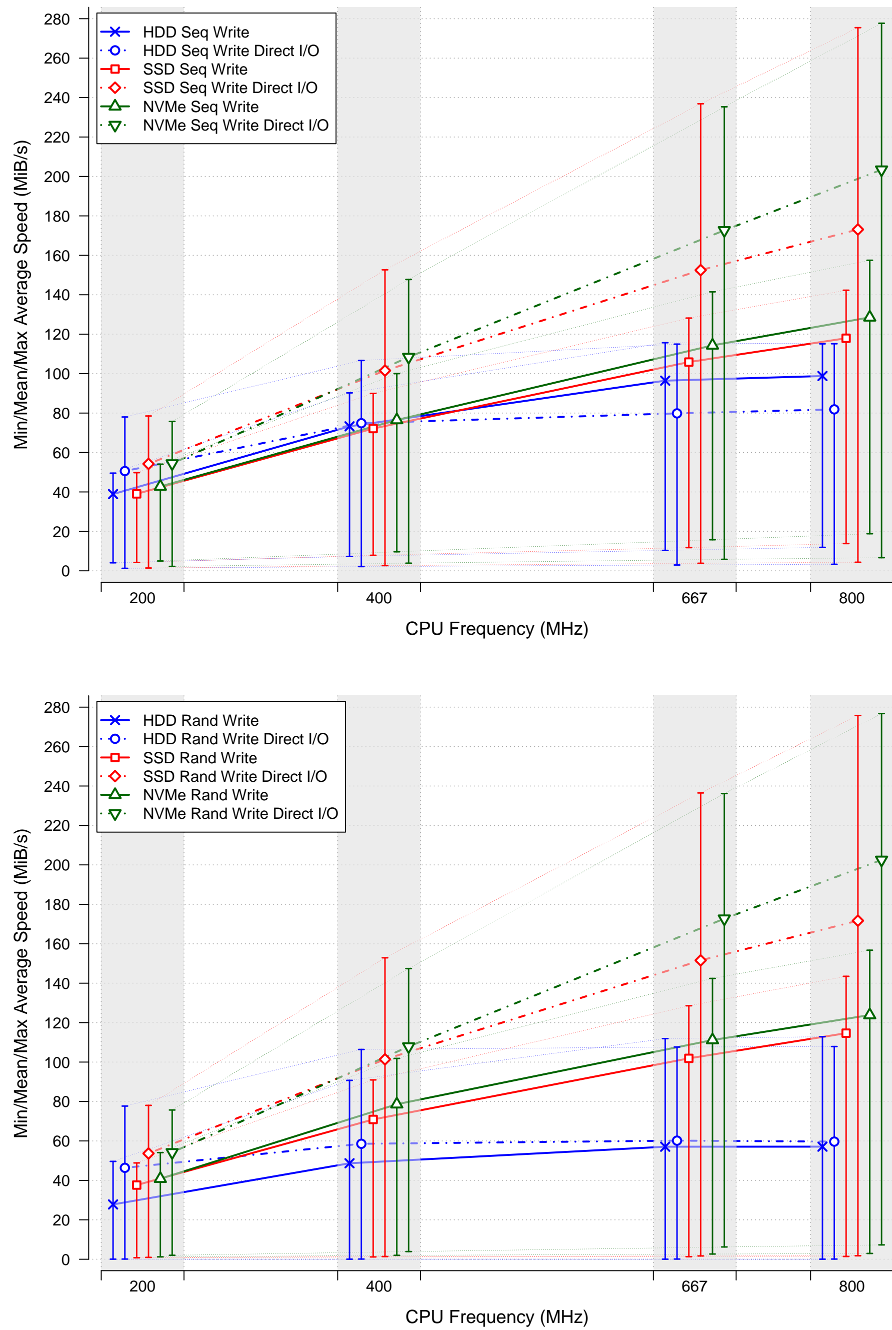


Figure 3.19: Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform at different frequencies (points show mean value, bars show minimum and maximum values)

A further selection of specific block sizes at a range of frequencies can be seen in Appendix E.1, showing well-defined examples of SSD results scaling with CPU frequency, while HDD results are often limited by the storage device itself.

3.6 Custom Low-overhead Profiling Component

While the timing measurement and profiling functionality of Linux (using tools such as *OProfile*) gives a good impression of where time is spent during application execution, both within user-space code and the kernel, a more specialised, hardware-based profiling timer component was developed to address the following requirements.

- Minimise CPU and memory overhead of measuring an event
- High-accuracy timing without relying on kernel timers
- Predictable and consistent overhead when an event is measured
- Simple to incorporate into tests and kernel code
- Ability to accurately measure hardware interrupts

This specification makes the timer far more suitable for taking measurements in real-time systems than standard profiling methods, as its use is minimally intrusive and relatively predictable. The minimal CPU and memory overhead is also advantageous for embedded systems, where resources are typically limited. The addition of hardware interrupt triggering allows for full software latency to be measured when performing I/O operations, such as accessing a storage device.

The profiling timer component operates by 'tagging' an event when software issues it a command, or when a specific hardware interrupt occurs. Events are recorded to an internal Block RAM (BRAM) buffer, and are comprised of a 32-bit tag value (or interrupt number) along with the trigger time. Each timing event is recorded using two sequential 32-bit register writes to the component – one to set the event tag and one to trigger the event. After an experiment is complete, all timing events can be read back from the internal buffer at once, ensuring measurements can be taken consistently and without interruption during the test.

An example of a plot generated from the profiling timer component output is shown in Figure 3.20. A number of software timing events are shown from both user-space and kernel code, along with interrupts recorded from within the FPGA fabric, sampled during a 4 KiB write to an ext4 file system on an NVMe SSD.

3.6.1 Hardware Implementation

The FPGA hardware core was implemented and tested using Xilinx Vivado HLS, allowing a high-level C++ specification of functionality to be compiled down to cycle-accurate HDL code. The hardware

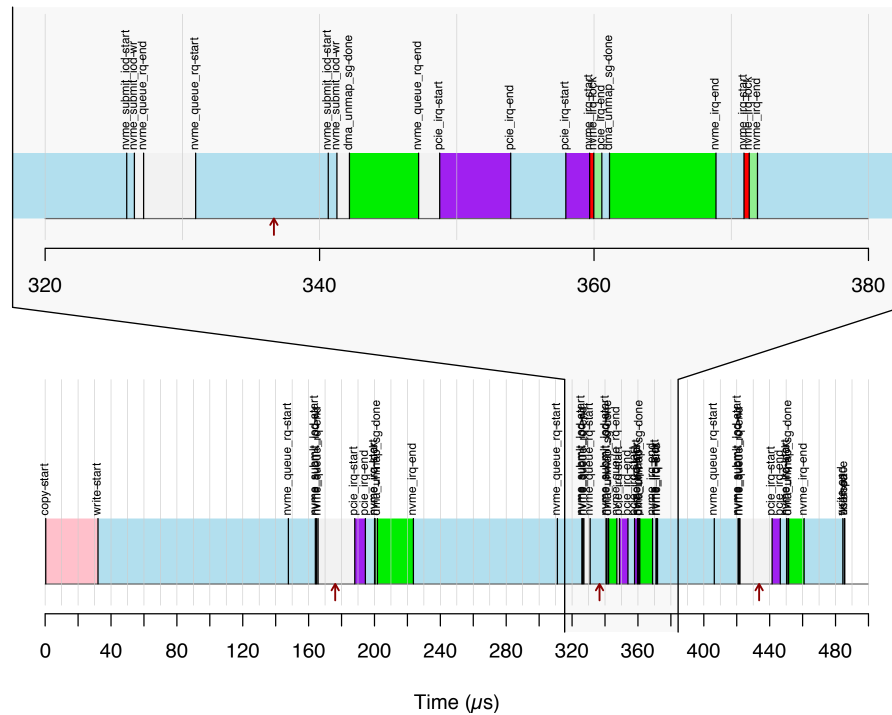


Figure 3.20: Example output of the profiling timer – vertical lines and labels show software events, arrows show interrupts; top section is a zoom of 320-380 µs

presents a simple set of memory-mapped registers for control and reading stored data, as well as a vector input for a timer counter value, and an interrupt input for triggering timing events from hardware. Source code for the Vivado HLS hardware components of the profiling timer is available at [4].

The component is designed to be used in conjunction with a number of standard Xilinx peripherals to allow for design flexibility and full control from software - *Binary Counter* IP cores to generate an incrementing time value and to count raw interrupt events, various glue logic (using *Utility Vector Logic* IP), and an *AXI GPIO* IP to enable control from the CPU over the main peripheral AXI bus.

3.6.2 Software Implementation

A basic user-space library was developed to interface with the profiling timer by accessing the registers to control timing, trigger events, and read back stored events. To minimise the size and complexity of implementation, this simply maps the Linux `/dev/mem` device using the `mmap` system call, then reads and writes the appropriate physical memory locations for the mapped registers. An example of the user-space library for accessing functions of the profiling timer can be found at [7].

Modifications to the Linux kernel code allow events to be timed within the kernel itself, also through a simple set of interface functions. These are implemented using `ioremap_nocache` on the address space of the timer to map it into virtual memory with no caching, followed by `iowrite32` calls for triggering events. To ensure measurements are only taken when required, a custom system call is used to dynamically enable and disable kernel event tagging globally. Full kernel modifications for using the profiling timer can be found at [5].

3.7 Timing Periodic Storage Access

In order to gain a deeper insight into how the CPU bottlenecks identified previously might affect the performance and variability of storage operations performed by a typical real-time task in an embedded system, a set of experiments were performed using the custom profiling timer component to measure low-level timings within the kernel. The experiments consist of a task periodically performing a fixed-size storage operation (either read or write), for various task priorities, system loads and storage interfaces types. The effects of varying a task's period on the resulting timings of its storage operations were also examined, in order to investigate how the frequency of a task performing such operations can influence variability.

3.7.1 Experimental Set-Up

Experiments were performed using the same basic set-up as the embedded experiments detailed in Section 3.5, but using a customised Linux kernel based on version 4.1.15, with `RT_PREEMPT` patches, various Xilinx patches for correct operation on the Zynq-7000 platform, and additional code included to support the profiling timer and measurement of VFS functions. The same Intel SSD 750 [16] NVMe SSD was used for all experiments, as it offers fast storage that has the greatest possibility of stressing other areas of the system.

Throughout the experiments, 'high load' was simulated by pushing all CPUs to maximum utilisation though several separate processor-intensive (but not memory- or disk-intensive) processes[4] running at real-time priority 0 (Linux priority 20, *nice* 0), 'medium load' used the same method but only running a single secondary process so only a single core was occupied, and 'low load' left the system in its default idle state with only basic background tasks running. High priority is achieved through setting the `SCHED_FIFO` priority of the test process to real-time priority 10 (Linux priority 10, *nice* -10), while low priority uses the default non-real-time system scheduling and priority.

4 Specifically, each 'system load' command was: `nice -n 0 yes > /dev/null &`

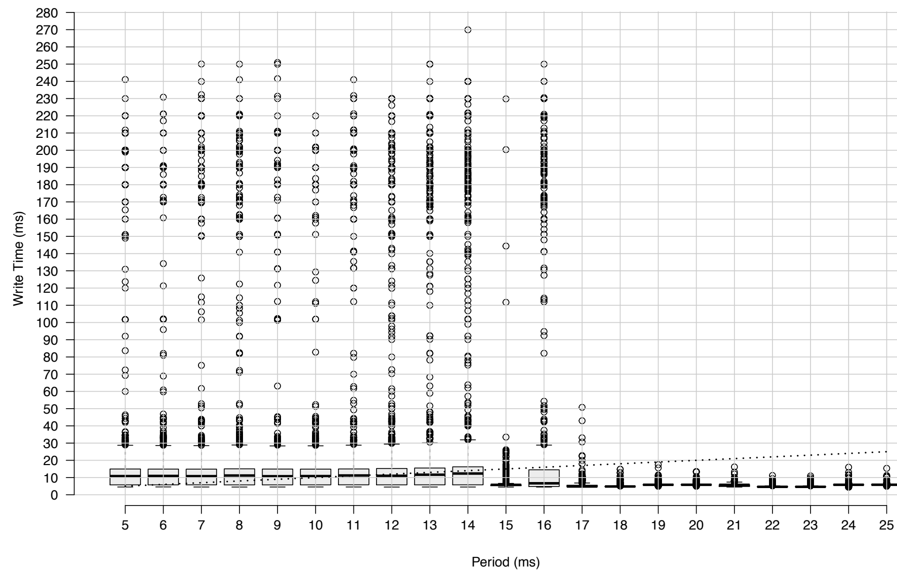


Figure 3.21: Timing of periodic high-load, low-priority, synchronous block device writes; dotted line shows task period

To simulate a periodic data streaming task transferring data to or from a device buffer, data is read from the SSD and written to the Zynq-7000's On-Chip Memory (OCM), an area of fast SRAM on the chip itself, or vice-versa. Each read or write operation transfers the full 256 KiB of the OCM, via a buffer within the application.

3.7.2 Varying Task Period

An initial test was run to determine an appropriate task period for further experiments, to ensure that storage operations had sufficient time to complete, and to measure the effect that period can have on the variability of operations. A low-priority periodic write to an NVMe block device was performed 10,000 times on a system with high load, at periods ranging from 5 ms to 25 ms, with total timing of the write operation measured using the profiling timer component.

Box plots of these results can be seen in Figure 3.21, showing a large number of outliers in quite distinctive bands, until the period is long enough for all write operations to complete before the next firing of the task, giving an observed worst-case response time less than the period from 18 ms upwards. At around 15 ms, a critical percentage of writes complete in less than the task period, freeing time in the system between writes and causing the number of major outliers to fall dramatically. The reversing of this effect for a 16 ms period is likely to be caused by either uncontrolled variation in the system, or scheduling effects caused by specific periods. The write time then stabilises at 17 ms and above as the system slack time increases.

3.7.3 Performance and Variability Results

To determine the timing variability of results in a more empirical manner, specific areas identified during profiling were measured over many runs of periodic storage tasks.

Based on the results from varying a likely worst-case storage task period detailed in Section 3.7.2, tasks for this set of results were run with a period of 25 ms to ensure minimal interference between subsequent firings. Additional results collected with a 10 ms period confirm the patterns observed with shorter periods previously, and so show a much larger difference between the variability of faster and slower storage operations (with response times less than or greater than approximately 10 ms respectively). Full results from experiments run with a 10 ms period are presented in Appendix E.2, with results and discussion here restricted to the more stable 25 ms experiments.

As may be predicted, higher system load and lower task priority generally increase timing variability as other processes on the system cause more interference, however the extent that these affect the storage task varies across access methods.

For both reads and writes, using direct I/O greatly reduces the number of extreme timing results when accessing storage through a plain block device, as shown by the outliers in Figure 3.22. Results for an *ext4* file system in Figure 3.23 show a similar pattern, although there are more outliers remaining even when direct I/O is enabled, likely due to the inherent complexity that the file system adds. While direct I/O also improves both observed worst-case and average speed when writing for both block device and file system accesses, the same is not true for reading, with only the observed worst case being faster.

3.7.4 Asynchronous I/O Results

The default for disk I/O in Linux is to use asynchronous requests, where transfers can be completed by the kernel after the system call initiating the transfer in the user-space application has returned. Results presented to this point force I/O requests to be serviced synchronously, giving a more accurate (and theoretically more consistent) measurement of the response time. If it is not necessary to enforce transfers to be complete before the next firing of a task then asynchronous I/O can be used, but this can cause consecutive requests to queue up and affect future transfers.

To investigate differences between asynchronous and synchronous I/O on a periodic storage access task, the same set of experiments was performed without the `O_SYNC` flag specified on file open.

In general, median write speeds are faster than when using synchronous transfers, and distribution of results is flatter, however there are some more extreme outliers present when using ext4 with stand-

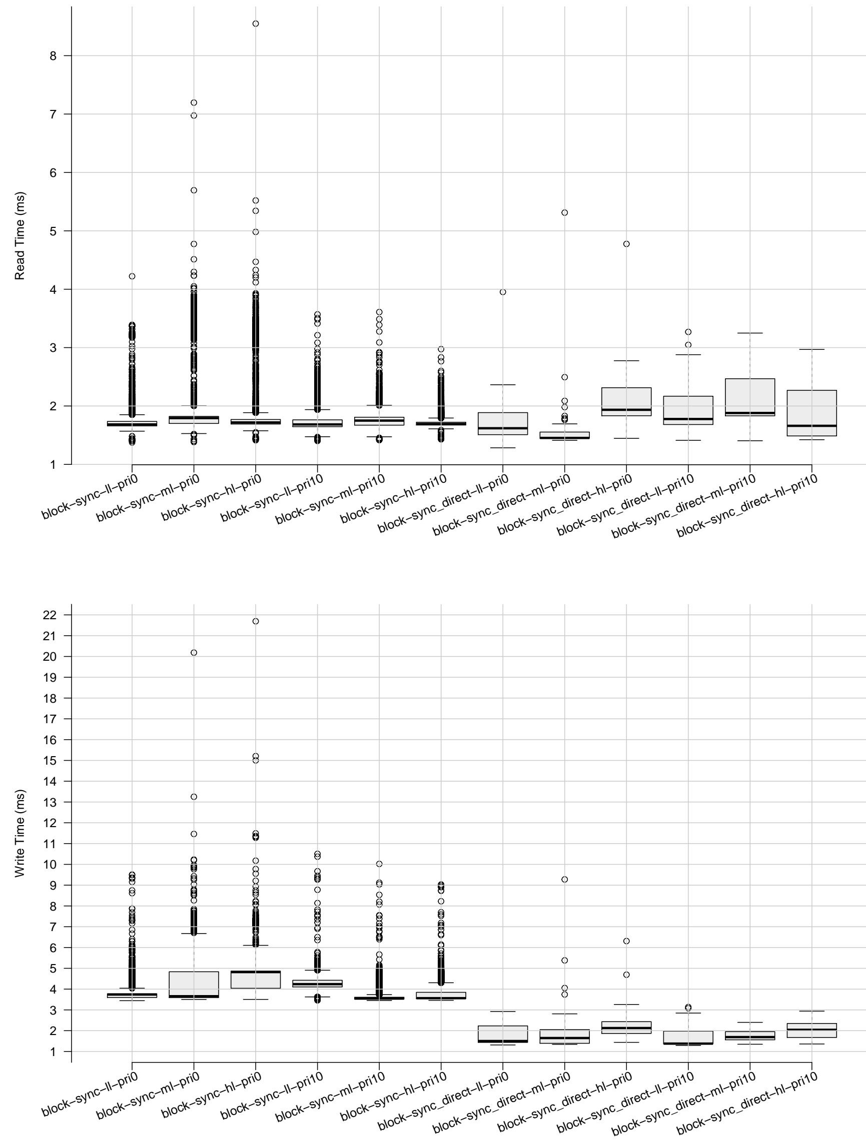


Figure 3.22: Standard and direct I/O speeds for periodic block device synchronous reads and writes, at low, medium and high load, and low and high priority

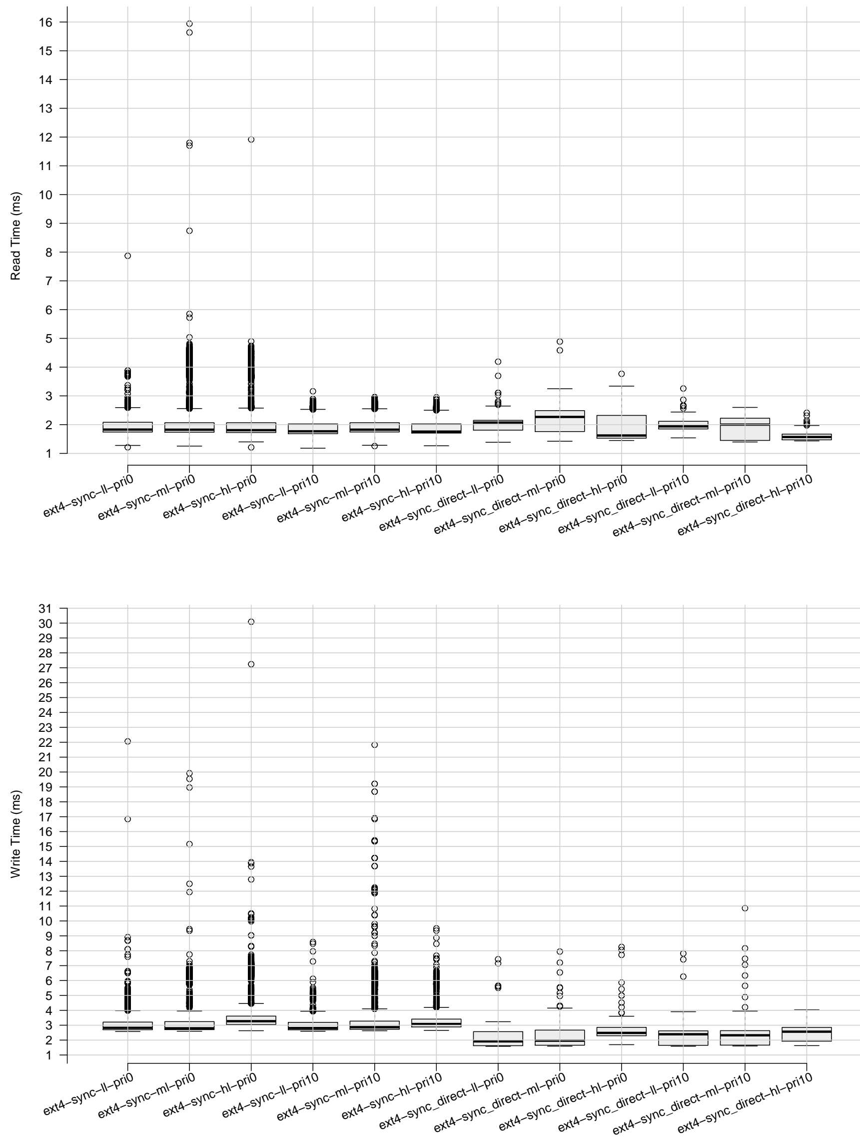


Figure 3.23: Standard and direct I/O speeds for periodic ext4 file system synchronous reads and writes, at low, medium and high load, and low and high priority

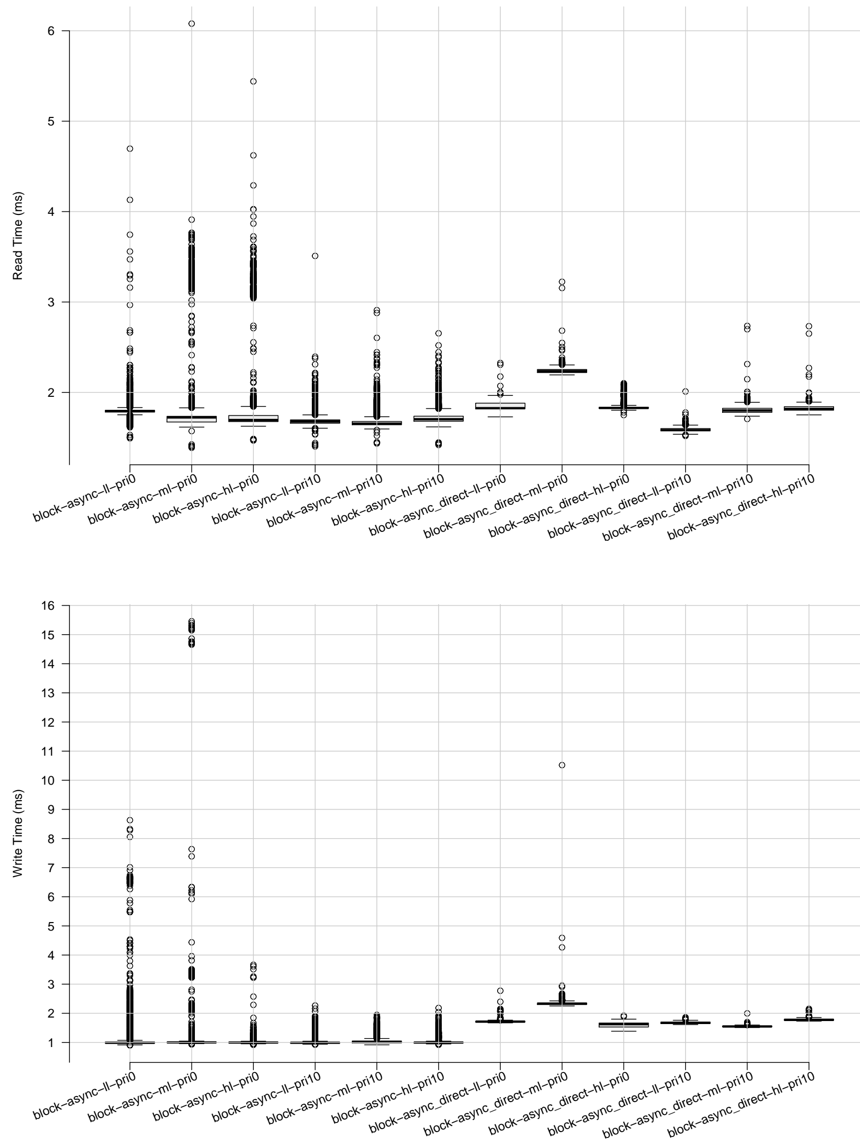


Figure 3.24: Standard and direct I/O speeds for periodic block device asynchronous reads and writes, at low, medium and high load, and low and high priority

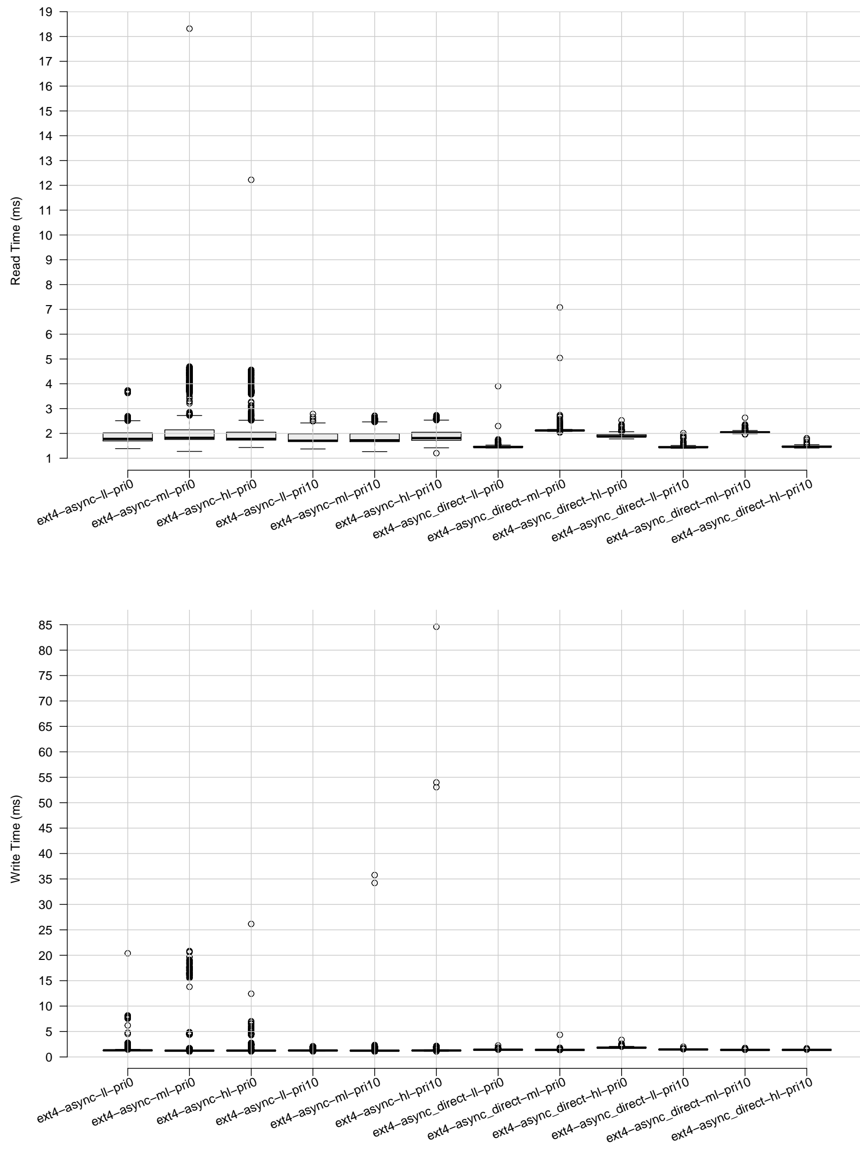


Figure 3.25: Standard and direct I/O speeds for periodic ext4 file system asynchronous reads and writes, at low, medium and high load, and low and high priority

ard I/O, even when running at a high priority. For both reads and writes, the distribution of direct I/O results is much flatter when using asynchronous transfers. This is likely due to any extreme variations in disk access time being absorbed in the time between tasks firing, as the average response time for complete requests is still far below the period.

3.7.5 Linux Raw Device Results

For completeness, and since raw devices are not officially considered deprecated in the Linux kernel [120], block device experiments were repeated using the same device bound to a `/dev/raw/rawN` virtual device.

Observed raw device results for all run types are shown in Appendix E.3, and show some variation from block device results. Most noticeably, writes are generally faster and less variable than block devices, even when using direct I/O, however there are still some significant outliers across the results, especially with read timings. Results vary more between different priorities than block device results, even with a low system load, and some high priority read results are slower than their low priority counterparts.

While these results could be an indication that raw devices can provide a better interface than block devices in some circumstances, they also may simply highlight a greater issue with the difficulty of drawing coherent conclusions from a set of storage benchmarks. Given that raw devices are supposedly interchangeable with using direct I/O on block devices, and both operate using the same methods at a low level within the kernel, any differences between results may just reinforce the inherent variability and unpredictability of storage access in a Linux system, and of storage devices themselves. Also note that while raw devices are essentially direct I/O block devices at their core, raw devices themselves can be opened using the `O_DIRECT` flag, giving a further option for accessing storage that theoretically has no practical meaning, reinforcing ideas that the Linux direct I/O implementation can be problematic [118].

3.8 Low-level Storage Access Profiling

While the timing results presented as box plots in Section 3.7 give an idea of the overall performance and variability of repeated storage accesses, they do not break down each request into where time is spent, whether in the application, within the kernel, or waiting for hardware. To this end, the custom profiling timer component (detailed in Section 3.6) was used to measure the low-level timings of various-sized read and write requests to file systems and plain block devices,

while enabling and disabling asynchronous transfer and direct I/O flags.

Key functions were identified and coalesced into higher-level areas of interest (such as kernel time, storage device time, and interrupt delays), in order to create timings appropriate for averaging across many runs, giving a suitable set of results from which to draw comparisons.

3.8.1 Individual Profiling Results

A number of profiling tests were used to observe the different patterns of accessing storage through different means (using a file system or low-level block device, with and without direct I/O) as well as the variability between different runs of the same task. These tests were also used to determine appropriate points of measurement for future timing tests, where a reliable number of appearances was needed in order to perform variability analysis over many sets of timing results.

One key observation from these experiments is the difficulty in predicting the exact pattern of low-level storage access for a specific request, where repeated runs can have varying numbers of physical storage device accesses, and follow different paths through the kernel based on decisions made by the file system and block layer. This is especially a problem for file system accesses, where additional information about a file may need reading to find its physical location, or extra writes may be required to update journals or save evicted areas of cache.

The read profiling results in Figure 3.26 show that for the small transfers of 4 KiB and 8 KiB, the majority of timing across all runs is spent processing and copying data in the kernel, rather than waiting for the storage device itself. While some variability comes from the storage device, most noticeably in a few outliers, the majority of timing differences between runs is clearly in software, especially when using ext4 rather than the plain block device.

One observation from both the ext4 and block device results is how a 4 KiB read when not using direct I/O actually reads the same amount of data from the storage device as a 8 KiB read, but only copies the first 4 KiB into the application's memory. This is due to the block device layer prefetching the next block of data from storage into the page cache, which can increase throughput if this data is required, but wastes time if it is not, and is difficult to predict from an application due to the process being entirely within the kernel. Direct I/O results do not exhibit this behaviour, so is a desirable option when exact storage access behaviour is required.

The write profiling results in Figure 3.27 show a more major difference between ext4 file system and plain NVMe block device accesses than when reading, with ext4 accessing the storage device multiple

Figure 3.26: Profiling results for 20 runs each of 4 KiB and 8 KiB reads, with and without direct I/O, from ext4 (top) and NVMe block device (bottom)

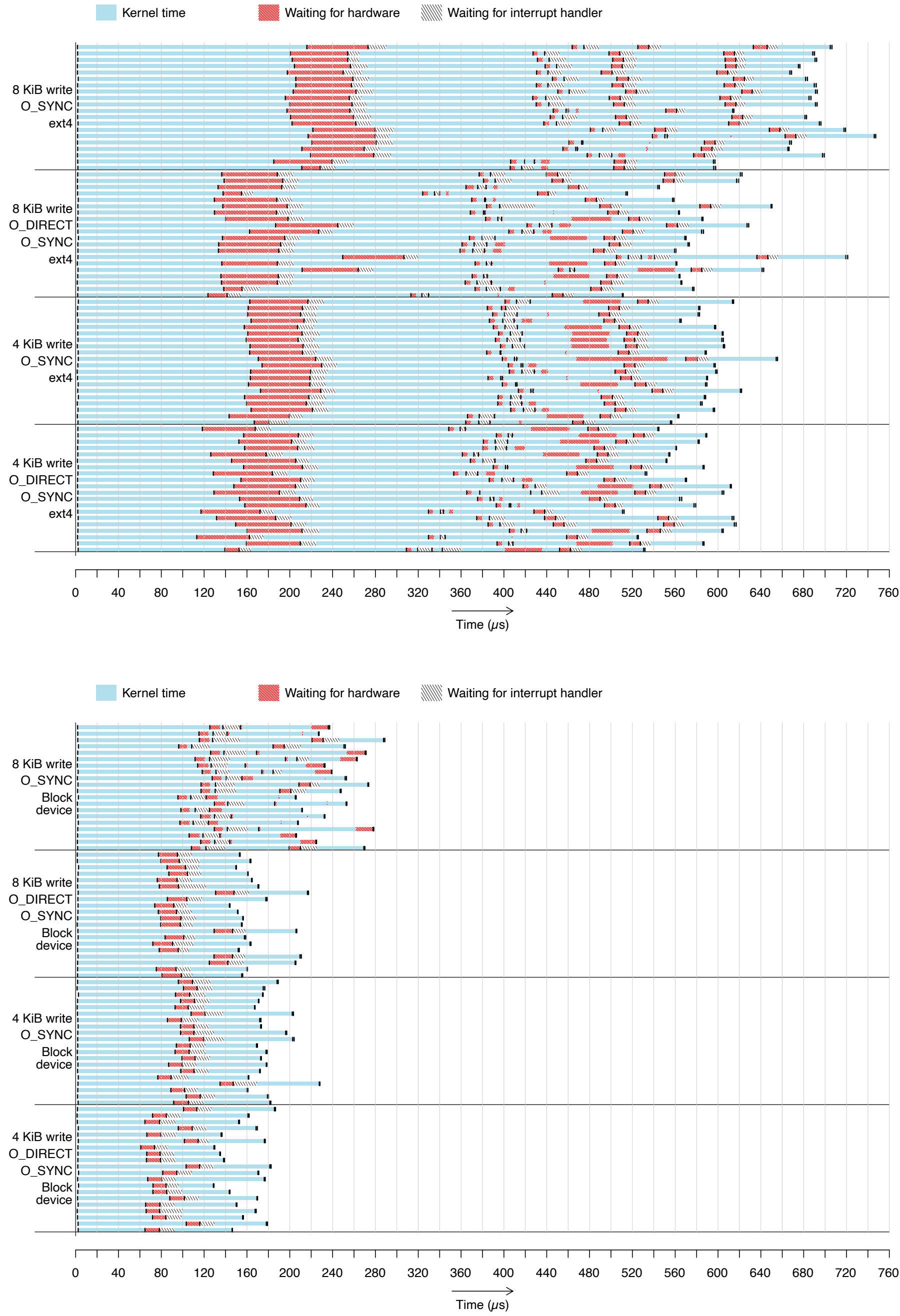


Figure 3.27: Profiling results for 20 runs each of 4 KiB and 8 KiB writes, with and without direct I/O, to ext4 (top) and NVMe block device (bottom)

times after the main data write, and taking overall far longer to return to the application from the system call as a result. Even more so than with reads, the majority of time is spent in software rather than waiting for the storage device, however the software overhead does not change significantly when doubling the transfer size.

These small transfer size observations are further reflected when performing 40 KiB and 80 KiB transfers, as shown in Figures 3.28 and 3.29, including prefetching when not using direct I/O, and more time waiting for software than hardware. One 40 KiB ext4 write result demonstrates how unpredictable storage operations can be, clearly showing a long, unrelated storage device access preventing the main write from taking place, possibly triggered by the requested operation, before the data is eventually written to disk.

A further significant difference made by using direct I/O can be seen in block device writes, where storage requests are made for the entire block of data in a single transfer, compared to the interleaving of transfers and software processing shown in the requests without direct I/O. While there is significant variation in the time taken for direct I/O requests to be processed, they are consistently faster and simpler than when not used.

3.8.2 Probability Density Functions

Figures 3.30 to 3.33 show probability density functions, calculated using kernel density estimation, of the total read or write times from the data presented in the profiling plots, but across 1,000 runs per experiment instead of just 20. These plots can be used to visualise variation in the transfer times, along with identifying patterns and outliers.

Most of the experiments show clear peaks in timing probability, corresponding to areas where the majority of runs are placed. Additionally, in many cases the probability distribution function shows multiple peaks, demonstrating that the timings of transfers are likely to be clustered around specific values rather than spread evenly or normally distributed, but usually still with an ordered precedence of probability.

A number of experiments also show clear outliers in their results, with the most extreme of these happening when not using direct I/O. These are potentially due to low-probability interactions with other tasks running on the system, such as caches being flushed, or the storage device taking longer to respond to a specific instance of an operation.

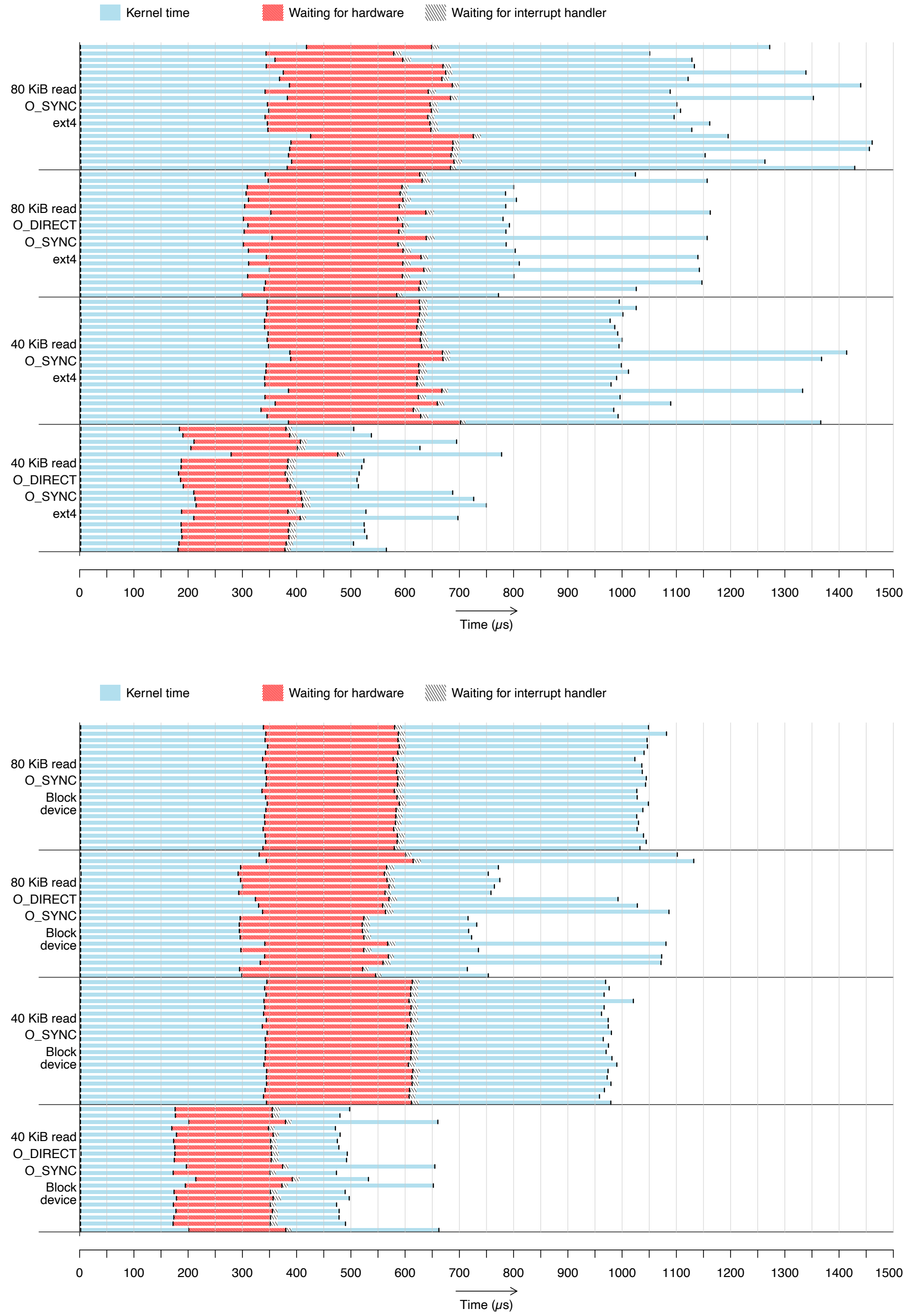


Figure 3.28: Profiling results for 20 runs each of 40 KiB and 80 KiB reads, with and without direct I/O, from ext4 (top) and NVMe block device (bottom)

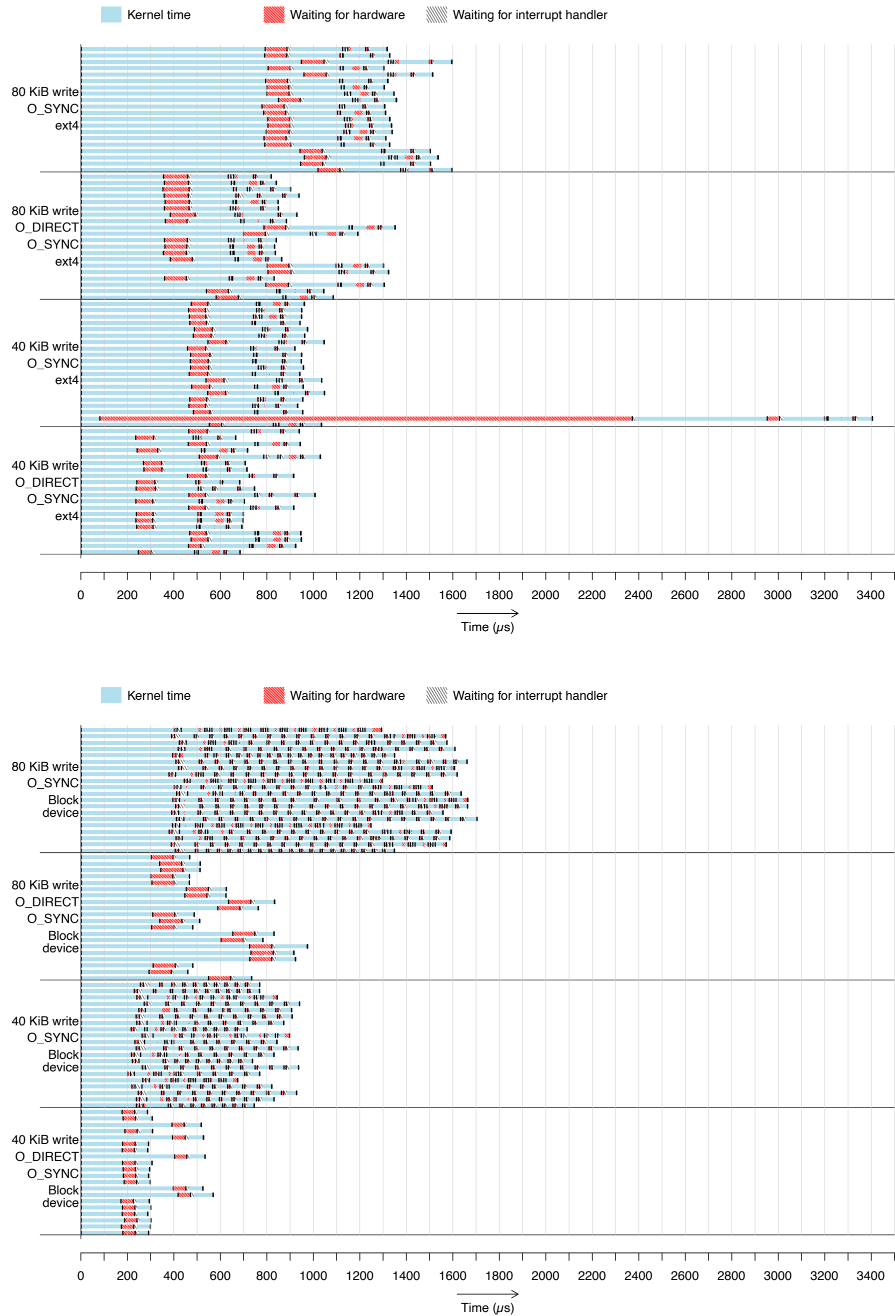


Figure 3.29: Profiling results for 20 runs each of 40 KiB and 80 KiB writes, with and without direct I/O, to ext4 (top) and NVMe block device (bottom)

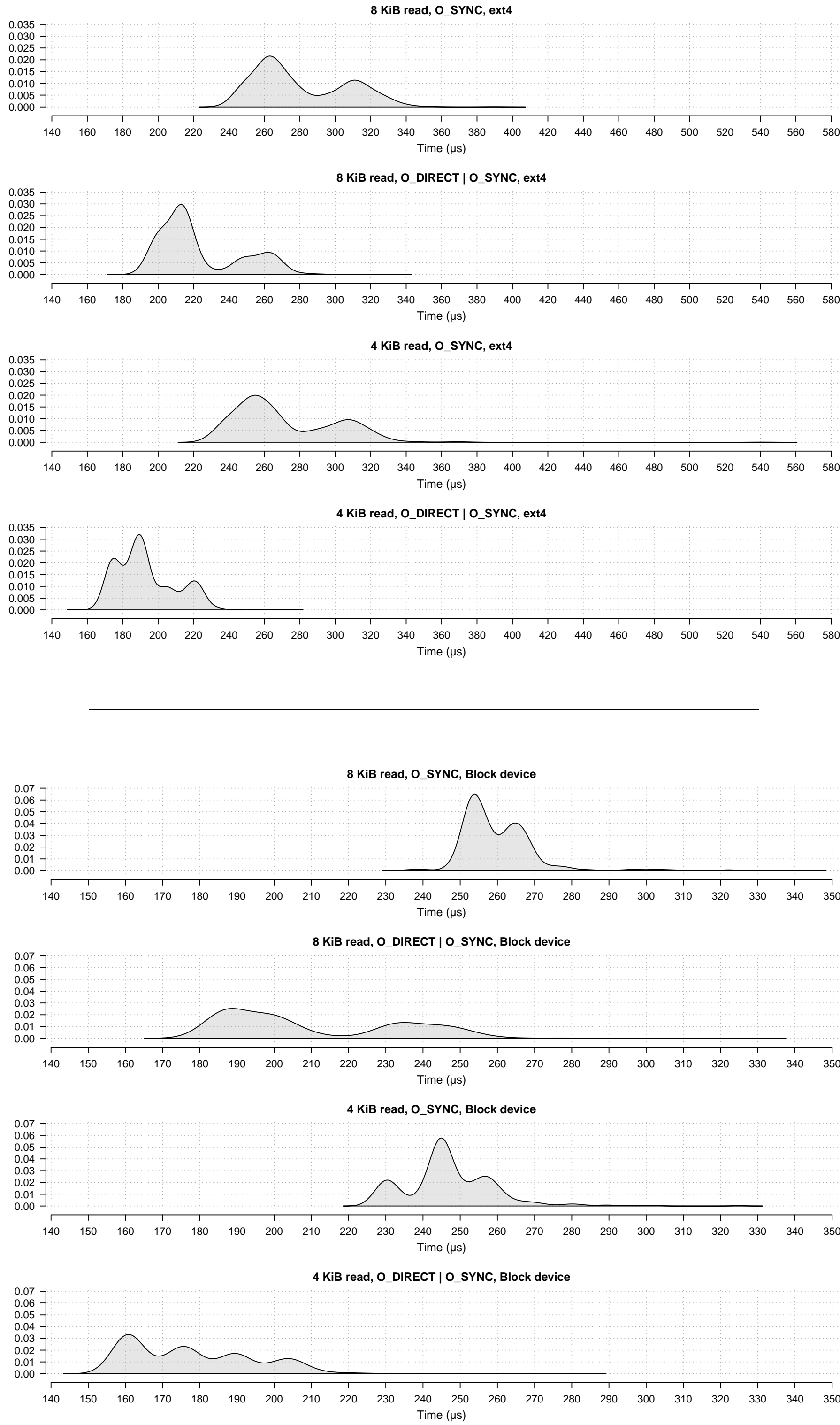


Figure 3.30: Probability density function of 4 KiB and 8 KiB read timings, with and without direct I/O, from ext4 (top) and NVMe block device (bottom)

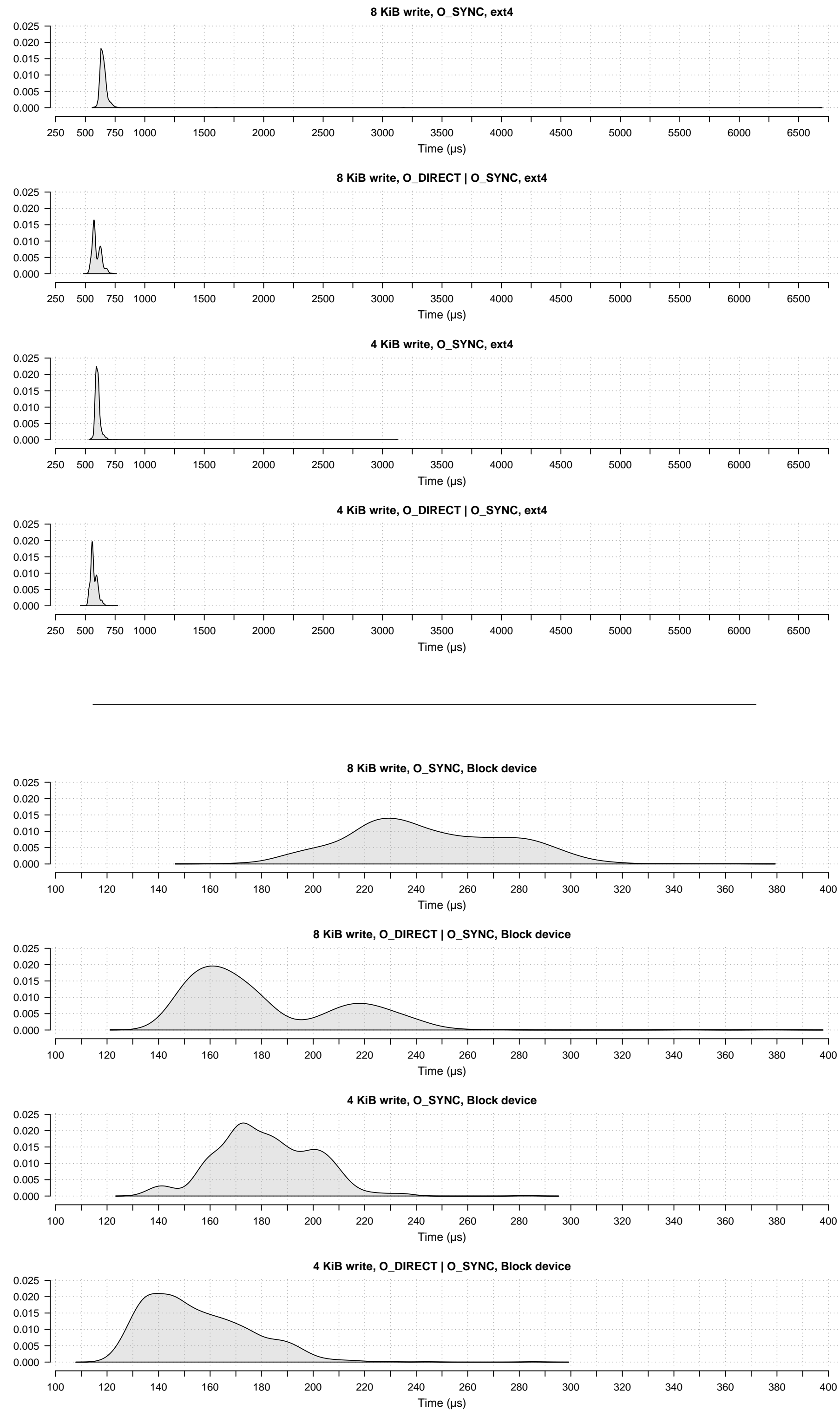


Figure 3.31: Probability density function of 4 KiB and 8 KiB write timings, with and without direct I/O, to ext4 (top) and NVMe block device (bottom)

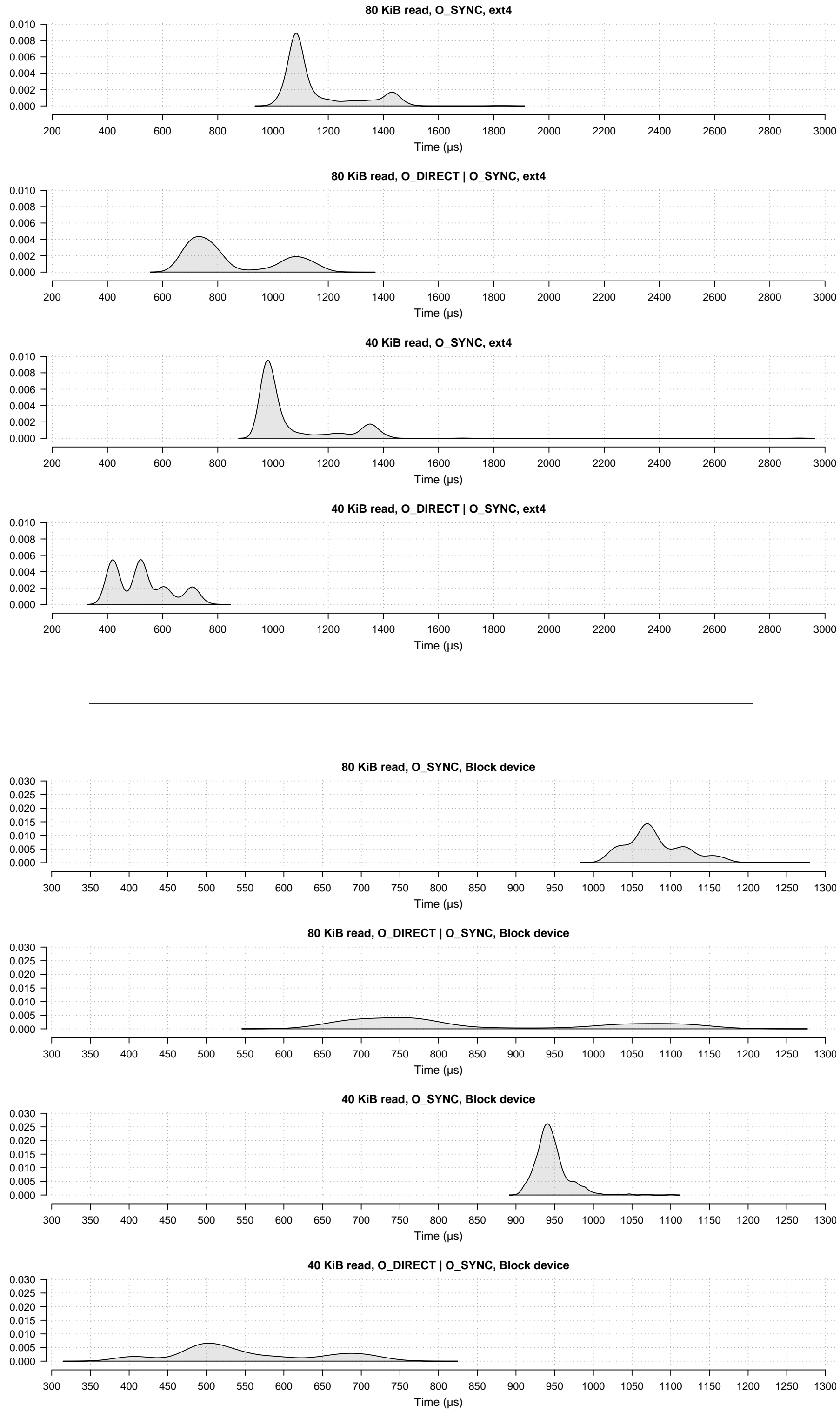


Figure 3.32: Probability density function of 40 KiB and 80 KiB read timings, with and without direct I/O, from ext4 (top) and NVMe block device (bottom)

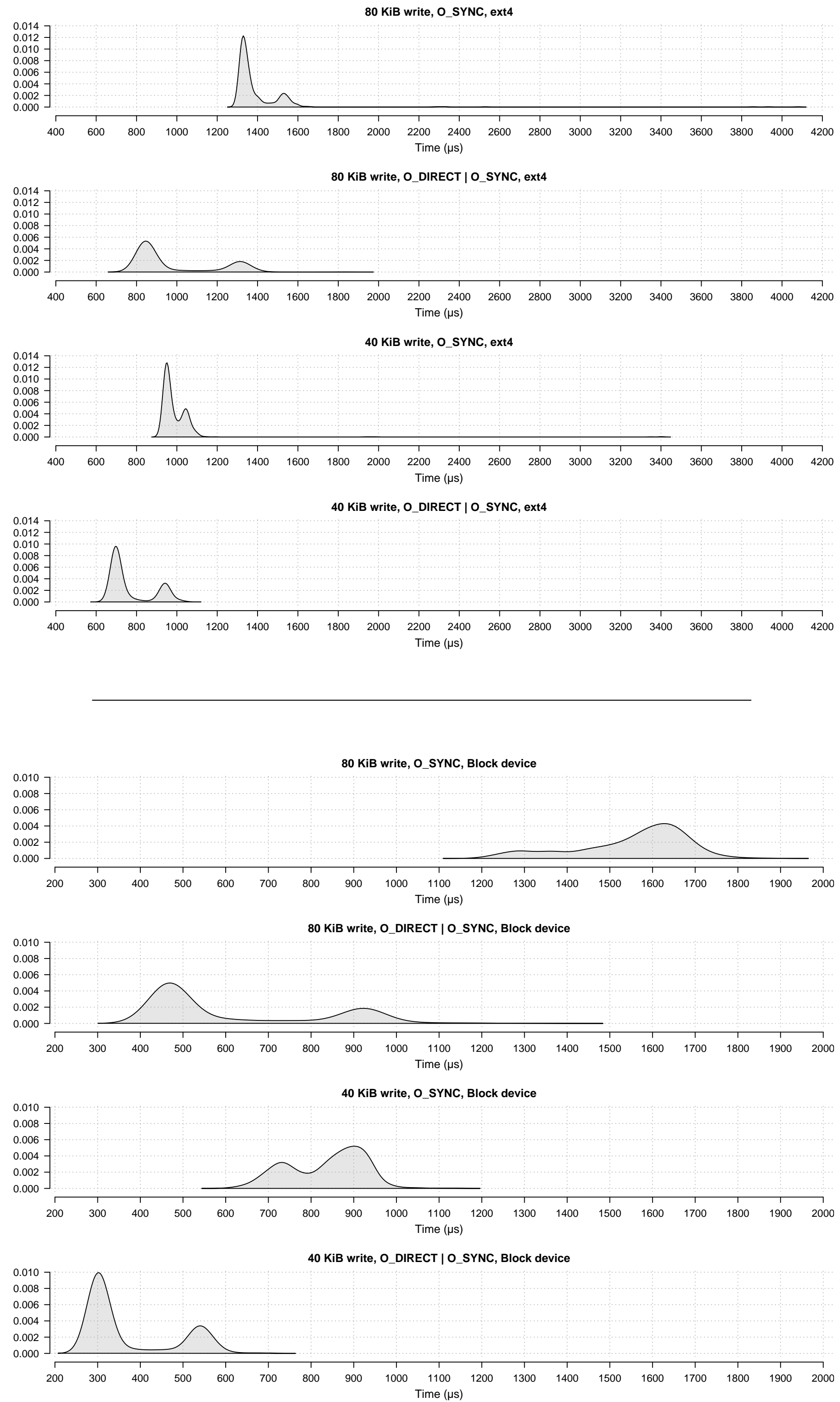


Figure 3.33: Probability density function of 40 KiB and 80 KiB write timings, with and without direct I/O, to ext4 (top) and NVMe block device (bottom)

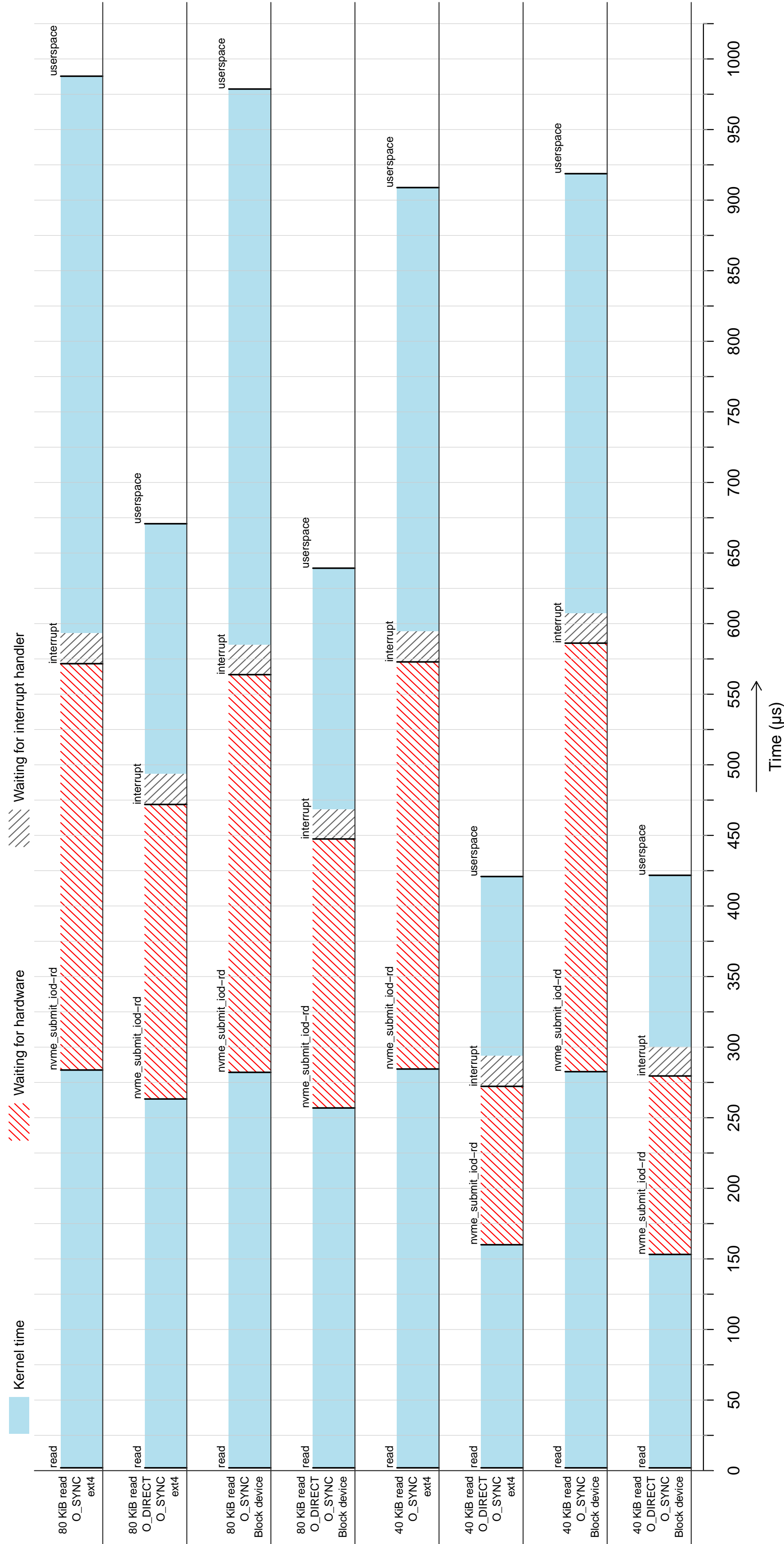


Figure 3.34: Average profiling results for 40 KiB and 80 KiB reads, with and without direct I/O, from ext4 and NVMe block device

3.8.3 Average Profiling Results

Figure 3.34 shows mean timings across 10,000 40 KiB and 80 KiB reads, allowing general trends to be confirmed. This highlights the efficiency of using direct I/O, and the prefetching performed by standard I/O. Producing a similar average plot of write results is infeasible, due to the unpredictable pattern of storage accesses associated with writing using standard I/O.

3.9 Simulated Real-world Benchmarks

In order to provide context for the synthetic benchmarking results presented in this chapter, a number of simulated real-world benchmarks were performed on the NVMe SSD using Filebench [127], an open-source framework for file system and storage benchmarking. Each benchmark was run both with and without direct I/O, to demonstrate situations where it can improve performance, and those that might suffer from avoiding the page cache.

3.9.1 Filebench

Filebench defines its benchmarks in terms of workloads, which emulate real-world usage patterns of an application. Each workload is defined using 'filesets', 'flowops', 'processes' and 'threads'. A full description of how Filebench operates can be found in [127].

Filesets create a set of files in a directory structure for the benchmark to operate on, which can be partially or entirely allocated and committed to disk before the benchmark runs. File distribution and sizes often follow a tuned random distribution in order to emulate a real system.

Flowops represent actions that can be performed on files within a fileset, such as opening, reading, writing or appending data, as well as actions that are not specifically file-related, such as emulating CPU consumption or idling processes. These operations form the main work of the benchmark, attempting to emulate the actions that may be performed by the real-world application being simulated.

Processes and threads provide a mechanism of chaining operations in a manner that emulates real-world application flows, with each thread containing a list of flowops, and each process containing one or more threads. Each process and thread can optionally have a set amount of memory assigned to it, and can be replicated a number of times.

3.9.2 Direct I/O with Filebench

Filebench includes basic support for direct I/O with flowops that allow it (for example, open, create, read, write and append), causing the `O_DIRECT` flag to be passed with the `open` syscall. However, Direct I/O transfers in Linux require additional considerations, such as ensuring file accesses are aligned to the block or sector size of the underlying storage device, as whole blocks are copied directly to or from the memory space of the requesting application. The Filebench source code includes minimal considerations to allow this when using a 512 byte block size device, rounding I/O buffer allocations for direct I/O transactions up to the next multiple of 512 bytes, however this value is hard-coded, and there is no mechanism for enforcing that all random-sized file allocations and accesses are also aligned.

In order to allow the use of 4 KiB block size devices with direct I/O, three changes were made to the Filebench source used in these benchmarks. Firstly, the hard-coded memory alignment of 512 bytes was increased to 4 KiB, which uses more memory for all direct I/O allocations, but aligns correctly for both block sizes. Secondly, the 'appendfilerand' flowop was modified to round its appended data size up to a 4 KiB unit, in order to ensure all files maintain a size that is a multiple of 4 KiB. Thirdly, a new custom random variable, 'cvar-gamma-aligned', was created, which operates in the same way as the commonly used gamma distribution, but rounds all output values to a set alignment, which in this case can be 4 KiB.

Using a combination of these three modifications, along with the workload adjustments detailed below, direct I/O benchmarks can be performed successfully on all devices with a block size that aligns to 4 KiB. The changes made to the Filebench source code, which was subsequently used for the benchmarks in this thesis, is available at [10].

3.9.3 Benchmark Workloads

A number of benchmark workloads were created, each with and without direct I/O enabled, to cover a variety of real-world computing applications. Each workload is based on a standard predefined benchmark personality included with the Filebench source (described at [128]), with customisations to appropriately fit the benchmark to the size of the SSD and memory of the systems under test, along with modifications to allow direct I/O to be used, as detailed below.

Between each benchmark run, all files are cleared and the operating system issues a *trim* command to the SSD in order to start the benchmark from a known state. Additionally, the system page cache is synced and cleared between fileset creation and the benchmark being

run, in order to avoid unfairly starting the benchmark with the most recently created files still in the cache, as recommended by [127].

Specific workload definitions are included in Appendix C.3.

Web server (webserver)

The predefined *webserver* workload emulates a basic web server by spawning a number of threads, each opening and reading an entire file. Additionally, a small write is performed periodically as if to a log file. File sizes follow a gamma distribution with a fairly small average of 16 KiB.

The following changes were made to the predefined *webserver* workload:

- Increased *nfiles* from 1,000 to 600,000 to increase dataset size to over 8 GiB (double the size of system memory on the server platform)
- Reduced *nthreads* from 100 to 50 to reduce total memory footprint to under 1 GiB (size of system memory on the Zynq-7000 platform), as each thread is allocated 10 MiB
- Direct I/O workload adds 'directio' option to *openfile*, *readwholefile* and *appendfilerand* flowops

File server (fileserver)

The predefined *fileserver* workload emulates a simple file server, with a number of threads acting as virtual users who read, write, append and delete files, then perform a *stat* operation on a file. Files are relatively small at 128 KiB each.

The following changes were made to the predefined *fileserver* workload:

- Increased *nfiles* from 10,000 to 80,000 to increase dataset size to over 8 GiB (double the size of system memory on the server platform)
- Direct I/O workload changes *filesize* random distribution from *cvar-gamma* to *cvar-gamma-aligned* with an alignment of 4 KiB
- Direct I/O workload adds 'directio' option to *openfile*, *createfile*, *readwholefile*, *writewholefile* and *appendfilerand* flowops

Mail server (varmail)

The predefined *varmail* workload emulates a traditional UNIX mail server, using `/var/mail` and one file per message. Emails are an average of 16 KiB each, with files being created, written and read to emulate the receipt and reading of messages.

The following changes were made to the predefined *varmail* workload:

- Increased *nfiles* from 1,000 to 600,000 to increase dataset size to over 8 GiB (double the size of system memory on the server platform)
- Direct I/O workload changes *filesize* random distribution from *cvar-gamma* to *cvar-gamma-aligned* with an alignment of 4 KiB
- Direct I/O workload adds 'directio' option to *openfile*, *createfile*, *readwholefile* and *appendfilerand* flowops

Video server (videoserver)

The predefined *videoserver* workload emulates an application serving video files to clients, with two filesets – one with passive videos that are slowly removed and replaced, and one with active videos that are being 'streamed' to clients. Files are much larger than other workloads, at 10 GiB each.

The following changes were made to the predefined *videoserver* workload:

- Reduced *numactivevids* from 32 to 8 and *numpassivevids* from 194 to 28 to fit the dataset onto the SSD
- Direct I/O workload adds 'directio' option to *createfile*, *read* and *writewholefile* flowops

Network file server (netsfs)

The predefined *netsfs* workload emulates a network file server, similar to the *fileserver* workload but with a different distribution of file sizes and accesses.

The following changes were made to the predefined *netsfs* workload:

- Increased *nfiles* from 100,000 to 400,000 to increase dataset size to over 8 GiB (double the size of system memory on the server platform)
- Direct I/O workload increases 'round' option of *wrtiosize*, *rdiosize* and *filesize* variable random distributions from '1k' to '4k'
- Direct I/O workload increases 'min' option of *wrtiosize* and *filesize* variable random distributions from '1k' to '4k'
- Direct I/O workload adds 'directio' option to *openfile*, *readwholefile* and *appendfilerand* flowops

Web proxy (webproxy)

The predefined *webproxy* workload emulates a basic web proxy server, creating and writing small files, opening and reading files, and deleting files, while occasionally appending to a log.

The following changes were made to the predefined *webproxy* workload:

Table 3.1: Simulated real-world benchmark results on the server platform

	I/O throughput (MiB/s)		Average op latency (ms)	
Benchmark	Standard	Direct I/O	Standard	Direct I/O
webserver	296.2	**535.9**	0.814	**0.551**
fileserver	337.9	**984.4**	3.375	**1.239**
varmail	80.5	**108.2**	0.708	**0.657**
videoserver	343.3	**519.7**	**2.259**	2.51
netsfs	2.7	**3.4**	**0.014**	0.091
webproxy	121.4	**206.9**	1.673	**1.177**
oltp	36.1	**85.7**	**4.729**	8.759

- Increased *nfiles* from 10,000 to 600,000 to increase dataset size to over 8 GiB (double the size of system memory on the server platform)
- Direct I/O workload adds 'directio' option to *openfile*, *createfile*, *readwholefile* and *appendfilerand* flowops

Online transaction processing database (oltp)

The predefined *oltp* workload emulates an Online Transaction Processing (OLTP) database, based on the Oracle9i I/O model. It spawns threads that perform many small, asynchronous random reads and writes, as well as performing larger synchronous writes to a log file.

The following changes were made to the predefined *oltp* workload:

- Increased *nfiles* from 10 to 1,000 to increase dataset size to over 8 GiB (double the size of system memory on the server platform)
- Removed 'useism' shared memory option from thread definitions as it causes issues with Filebench on this system
- Direct I/O workload increases *iosize* from '2k' to '4k' to align it with SSD blocks
- Direct I/O workload sets *directio* option to '1', enabling it on *read* and *aiowrite* flowops

3.9.4 Results

The above workloads were run for 1 hour each on an *ext4* file system created on the Intel SSD 750 Series 400 GB NVMe block device used previously for synthetic benchmark experiments. Results were collected for both the server and Zynq-7000 platforms (see Appendices B.1 and B.2 for details).

Tables 3.1 and 3.2 show benchmark results for the server and Zynq-7000 platforms respectively, with 'I/O throughput' showing the overall data throughput of all I/O flowops across the entire benchmark

Table 3.2: Simulated real-world benchmark results on the Zynq-7000 platform

	I/O throughput (MiB/s)		Average op latency (ms)	
Benchmark	Standard	Direct I/O	Standard	Direct I/O
webserver	**32.8**	16.8	**3.994**	15.666
fileserver	47.8	**77.7**	15.34	**7.945**
varmail	**13.4**	12.1	**4.055**	5.083
videoserver	124.1	**195.3**	8.241	**4.981**
netsfs	2.7	**3.4**	**0.076**	0.225
webproxy	**9.2**	7.3	32.111	**27.255**
oltp	3.3	**5.7**	**1.333**	14.576

runtime, and 'average op latency' showing the mean latency of flowops in the benchmark. Full outputs from the Filebench benchmarks, including per-operation breakdown of speeds and latencies, can be found in Appendices D.3 and D.4.

While total throughput varies greatly with the nature of the benchmarks, the server platform shows consistently higher speeds when using direct I/O transfers. Despite this, in three of the seven benchmarks standard I/O has a lower average operation latency, suggesting that access times can be improved by using the page cache in certain applications.

Throughput results on the Zynq-7000 platform are more mixed, with only four of the seven benchmarks showing higher performance when using direct I/O. These benchmarks are generally those that have larger block sizes or large sequential transfers, such as *videoserver*, which has the highest throughput by a large margin. The Zynq-7000 platform results also show consistently slower speeds and higher latencies than the server results, which is expected due to its slower CPU and smaller memory size.

Throughput of the *netsfs* benchmark is a special case as the overall potential I/O bandwidth is artificially limited by the event generation rate, leading to identical speeds between the two platforms as they both reach this limit. Additionally. the difference between standard and direct I/O benchmark throughput in this case is simply caused by the fact that direct I/O has a larger minimum block size, causing more data to be actually read into the application rather than disappearing unused into the page cache.

For both platforms, comparison with synthetic benchmark results suggest that higher I/O speeds may be possible in ideal circumstances, with sequential transfers of very large block sizes, but overall this confirms these results contribute to the idea that direct I/O can be a valuable way to improve storage performance for a range of systems and applications.

3.10 Discussion and Future Work

The results presented in Sections 3.4 and 3.5 show that when CPU resources are sufficiently constrained, there are clear bottlenecks in storage operations in Linux, besides the access times of storage devices themselves. In order to utilise the full potential 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. Additionally, the results presented in Sections 3.7 and 3.8 show that predictability of timing and patterns of access to storage hardware vary greatly between file systems, VFS function flags, and various sizes of request.

3.10.1 Alternative Operating Systems

There are several potential solutions to the problems covered, ranging from optimisations in existing software implementations to more radical system architecture changes. While one solution could be to avoid Linux entirely if efficient and highly predictable access to storage is required, instead opting to use a dedicated real-time operating system instead, such as FreeRTOS [129] or RTEMS [130], this is not feasible in all scenarios. Linux offers a number of benefits over many operating systems, such as a large software and support base, and support for many hardware architectures and devices.

3.10.2 Potential File System and VFS Changes

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, however such optimisations would potentially require large changes to the structure of the Linux kernel.

Results show that using direct I/O can give a major boost to performance, especially when a storage device is sufficiently fast to take advantage of this, but only if block sizes are above a reasonable threshold for disk access operations as speeds are severely reduced when block sizes are small.

Given its potential benefits, a reimplemented direct I/O-type system 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. This approach may be impractical however, due to the definition of large and small block sizes being very dependent on a specific system set-up, and using any generic

values would cause the same issues as the current general-purpose storage solutions.

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, however this is unlikely to ever happen, both due to the potential number of fundamental changes required across many layers of the kernel, and due to the high likelihood of resistance from kernel maintainers.

3.10.3 Block Device Alternatives

A number of alternatives or changes to the Linux block layer could improve performance and predictability for the system examined in this chapter. For example, a BSD-like approach of removing cached block devices entirely would solve any problems caused by buffering data in the block layer, but this would also likely never be accepted into the Linux kernel, especially as it is already possible to create raw devices, albeit as second-class device nodes.

Other alternatives to the block layer include bypassing the kernel entirely, and controlling storage devices from a user-space library. Examples of this include the Micron UNVMe driver [131] and Intel SPDK [132], both of which mainly target low-overhead access to data-centre storage, but similar principles could be applied to embedded systems. While this method removes many layers of overheads from the kernel, it does also add extra complexity into user-space applications, and raises issues of security and resource sharing as the privileged kernel layer is no longer in full control of the storage. These user-space libraries also require an element of kernel support, although this can be from more generic drivers for accessing hardware device buses and memory spaces. Resource usage can also be higher, especially with slower devices, due to the need to actively poll for the completion of storage requests rather than using kernel-level interrupt routines.

An additional option would be to access storage using a kernel driver that continues to present storage through the standard VFS, but as a character device instead of a block device, thus avoiding any block-layer overheads on the data path. This approach is examined further in Chapter 4, with the development of a Linux character device for efficient and predictable storage access.

3.10.4 Hardware Acceleration

A further 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 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.

A hardware-accelerated storage approach is investigated further in Chapter 4, with the aim of mapping storage directly into the system memory map, and abstracting the loading and storing of low-level storage blocks away from software, into an accelerator core on an FPGA.

3.10.5 Future Analysis Work

As with any problem analysis work, there is always potential for much deeper investigation into the operation of storage devices in an embedded Linux environment, in order to fully understand the bottlenecks involved across a greater number of platforms and use-cases.

While the results presented in this chapter highlight some examples of circumstances where storage speeds are heavily influenced by areas other than storage devices themselves, they focus on relatively simple tests compared to the full range of potential applications that rely on accessing storage in Linux. Further experimentation would be required with benchmarks based on a great number of real-world usage patterns in order to fully gauge the scope of the issues highlighted here, however the results presented, and the tools and methods provided, give a solid foundation for any future work in the area.

As well as experimental work on existing implementations, practical work to test the feasibility of the solutions suggested above is necessary in order to properly improve on the current situation, and provide further points for analysis. A number of these solutions are explored in Chapter 4, drawing on the results in this chapter to provide a model of storage, identifying areas that have issues and may be improved, and giving a base on which to test implementations.

3.11 Conclusion

The analysis presented in this chapter highlights a number of problems with the current implementation of the Linux storage model

when applied to embedded systems with fast storage, stemming from the high levels of complexity shown in Section 3.3, providing a level of motivation and justification for further investigation and an argument for improvements to be made in this area. The results from experiments in Sections 3.4 and 3.5 show that while CPU-based storage bottlenecks are most visible on constrained embedded systems, improvements may also be necessary for more-capable systems to take advantage of increasingly-fast storage devices, especially when high CPU load and energy usage is a concern.

Timing and profiling results presented in Sections 3.7 and 3.8 show a large variability in storage access times, which can be largely attributed to software when using solid state storage in a resource-constrained system. While using direct I/O and raw devices can reduce software overheads, there are still fundamental decisions within the standard Linux storage stack that cause inefficiencies and make low-level storage access patterns hard to predict.

Section 3.6 additionally presents details of a custom profiling hardware component for FPGA-based systems, along with included user- and kernel-space support libraries. This component allows for the measurement of software and hardware triggers with high accuracy and low overheads, facilitating the collection of results for this, and future, work.

The results and analysis shown in this chapter directly motivate and inform the further research on alternative methods of modelling and accessing high-speed storage for more efficient and predictable operation on embedded platforms, described in Chapter 4, as well as developing ideas for future work detailed in Chapter 5.

4

Alternative Storage Interfaces for Embedded Linux Systems

While previous chapters have focussed on the analysis of current storage access methods for real-time embedded systems, particularly with a focus on embedded Linux, this chapter proposes and examines a number of alternative interfaces for increasing both predictability and efficiency. These include bypassing the current file system, VFS and block layer of the Linux kernel entirely, instead creating a more direct path to storage, with varying levels of kernel, software and hardware involvement.

4.1 Introduction

In modern real-time embedded systems, the demand for access to large amounts of persistent data is increasing, for example, for the storage of modifiable data to prevent loss due to low power or malfunction, or when the size of data exceeds that of main system memory. This is supported by solid-state storage devices (for example, SSDs) that offer improved speed, predictability, reliability, space efficiency and energy efficiency compared to traditional mechanical storage media (such as hard disk drives). However, a key challenge for operating systems is to provide efficient and predictable access to persistent storage. The usual approach is to provide file system abstractions, with complex and inefficient supporting software within the operating system. While support for file systems in real-time operating systems such as FreeRTOS [129] and RTEMS [130] is improving, for embedded Linux, provision of predictable, efficient access to storage remains an open issue, despite improvements to Linux for real-time use [133]. This chapter firstly provides and evaluates the CharIO storage device driver, which bypasses the file system and the Linux block layer to provide increased timing predictability and improvements in performance. Following this, a user-space NVMe storage driver for embedded systems is presented, which has been ported from existing work targeting non-embedded platforms, Finally, an NVMe driver for a dedicated

MicroBlaze-based storage coprocessor is demonstrated, which allows low-level storage operations to be carried out by dedicated hardware instead of the main CPU.

In the majority of computer systems, persistent storage is accessed via a file system and the block device layer of the operating system, which maps a file name into a list of blocks on disk that are transferred to and from main memory. While this approach offers the benefit of abstraction over the storage, it is relatively complex and inefficient as it provides many additional services (for example, checking data integrity, enforcing file permissions and disk space quotas, file sharing between processes, and file caching), as well as a non-specific interface to many different storage device types [75, 80]. Elements of this inefficiency and lack of predictability are explored previously in Chapters 2 and 3, motivating the need to investigate alternatives.

For real-time embedded systems, timing predictability is a core requirement, in order to guarantee all necessary deadlines in a system can be met, as well as having sufficient performance to meet these deadlines [134]. As demonstrated in Chapter 3, the standard Linux storage stack provision does not lead to sufficiently predictable storage access. A number of new approaches are therefore proposed, which remove the complexity of the file system and block layer to provide applications with fast, predictable access to persistent storage. Ideas centre around bypassing the file system and block layer entirely, thus removing a number of obstacles to loading and storing data, such as caching, block operation scheduling, and the resolution of files, albeit at the expense of conveniences such as a kernel-level file hierarchy and system-wide data caching. A further extension is to support physical memory addressing for storage commands from user applications, bypassing additional levels of the Linux kernel, such as virtual memory and cache management, and allowing the direct transfer of stored data from addressable areas outside of main memory, which may be useful in an embedded system. A potential management interface is also proposed, using a simple user-space file system, which can be used to load data for a specific task into a storage buffer with minimal reliance on the operating system. The idea of moving a greater amount of the storage stack into hardware is also explored, replacing software routines running in or on top of the main operating system with auxiliary processors and hardware alternatives.

The remainder of the chapter is structured as follows. Section 4.2 introduces appropriate background and related work. Section 4.3 describes the CharIO storage interface, which is evaluated alongside other storage access methods in Section 4.4. Section 4.5 proposes a user-space alternative to CharIO, drawing on work from a wider open source project. This then contributes to a MicroBlaze storage driver presented in Section 4.7, which could be used as a storage coprocessor

alongside a master CPU. Finally, conclusions and ideas for future work are offered in Section 4.8.

4.2 Background and Related Work

In recent years, the evolution of persistent storage devices has outpaced improvements in CPU speeds, leading to ever-increasing pressure for efficiency in the way operating systems handle storage. Whereas a typical hard-disk drive might leave software routines responsible for less than 1% of the latency and energy usage of storage operations, this can increase to around 20% of latency and over 75% of energy usage for solid-state storage devices, and even higher when considering future non-volatile memory technologies [12]. This divergence of hardware and software performance is even more apparent in embedded systems, whose limited power and processing resources throttle software even further.

A large amount of this energy usage and latency can be caused by the file system, with modern file systems becoming ever more complex in pursuit of high-level user features that may not be appropriate for some systems [80]. These systems include those where efficiency is particularly important, such as the embedded domain. This complexity can also cause issues with timing consistency and predictability, especially when the range of configuration options is so large [135], and when pairings between file systems and storage device types can have an extreme effect on how well each performs [136]. Alongside suggestions to reduce the 'obesity' of file systems by tailoring them to be more appropriate to the systems they serve [75], more-extreme suggestions for improving storage performance include replacing operating system support with a more efficient in-memory user-space file system [137], or offloading file system functionality entirely into a custom hardware accelerator core [101].

Accessing storage in Linux, and in most other operating systems, involves several interacting layers of software, as shown in Figure 4.1. Storage interfaces designed specifically for accessing high-speed solid-state storage, such as NVMe, can offer efficiency improvements over older options such as AHCI or PATA, but overall performance and predictability is still limited by the rest of the storage stack. Storage hardware will also perform its own operations on top of the software stack, which are largely beyond the control of the operating system. The limited control over the internal operation of storage hardware is beginning to be addressed, with projects such as LightNVM (the Linux Open-Channel SSD subsystem) [138] giving more control to software applications and drivers, and allowing for more predictable access latencies.

The interactions between these layers can be hard to predict, as they are designed for fast best-case performance, rather than predictability

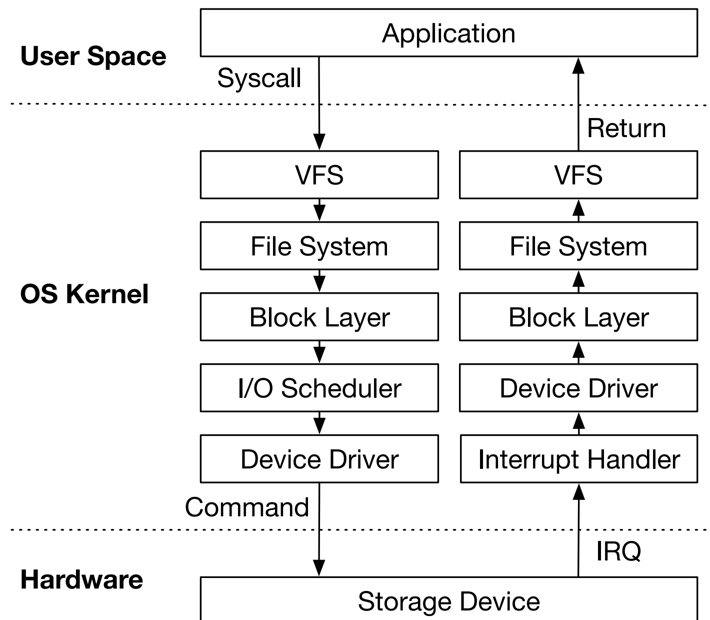


Figure 4.1: Storage access layers in Linux

or simplicity. For example, the number of times that a single-block read from an open *ext4* file actually accesses the storage device is variable: it could be once if the physical block is known, twice if extent information has to be looked up first, or not at all if the block is already cached by the operating system. Profiling results from Section 3.8 show that the situation is even more unpredictable when writing to storage devices, even when not using a file system. Additionally, control may potentially be returned to the user application at any time before the transfer is complete, if the storage request is processed asynchronously. Linux provides a number of ways to simplify the layers between an application and storage, such as 'direct I/O' and raw devices, both of which bypass the kernel page cache, and opening block devices directly with no file system. The effectiveness of these methods is ultimately limited, however, due to their reliance on the kernel's block layer and associated scheduler, as well as the fundamental principles of accessing files in Linux, as demonstrated in Chapter 3.

4.3 The CharIO Storage Interface

To investigate a simpler interface to storage from Linux that bypasses file systems and the block layer, a driver was created that presents the flat storage space of an NVMe SSD to user-space applications as a basic, optimised *character* device, rather than a standard block device and file system. Using a character device has the advantage of conforming to the standard UNIX model of device nodes being accessible through the VFS, while removing complex block layer features such

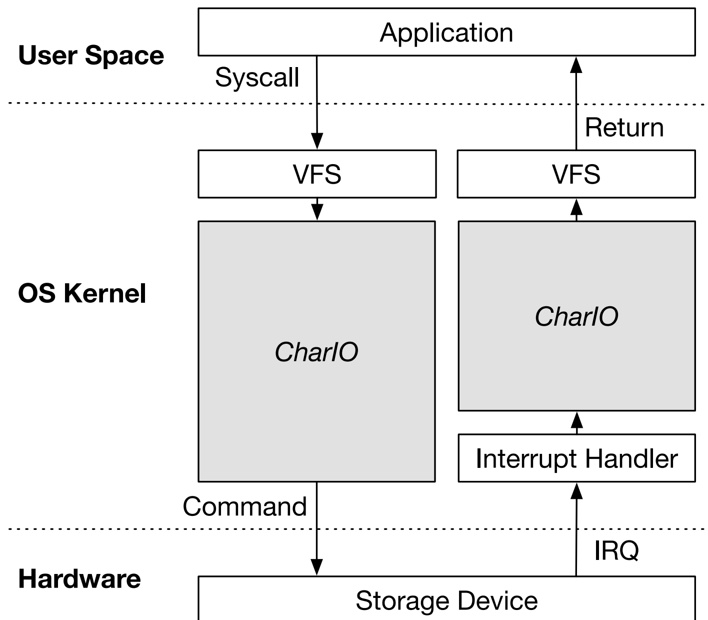


Figure 4.2: Storage access layers using CharIO driver

as the I/O scheduler, request queueing mechanisms, page cache, and asynchronous requests. This therefore has the potential to improve the efficiency and predictability of storage operations, as there is less software complexity between an application and a storage device, and the driver can be tuned to work in a more optimal manner for real-time embedded systems.

Full source code for the CharIO Linux kernel module, accompanying user-space library, and associated benchmarks and tests is available at [6, 7].

4.3.1 High-level Overview

At a high level, the CharIO kernel module acts as a wrapper around a modified version of the standard Linux NVMe device driver, creating a `/dev/chardiskX` character device node[1] instead of a `/dev/nvmeXnY` block device node[2] when an SSD is attached. This device node can then be accessed from user-space applications, supporting the standard open, close, read, write and seek system calls, and translating these into commands sent directly to the underlying storage device. This is shown in contrast to the standard Linux storage stack in Figure 4.2, with the key differences (other than lack of explicit file system) being the removal of the block layer and I/O scheduler by using a character device instead of a block device in the VFS, and the unification of the file interface and hardware device interfaces into a single module.

1 Where `X` is the number of the device.
2 Where `X` is the number of the device, and `Y` is the number of the NVMe namespace.

For efficiency during a read or write, all data is directly transferred by the storage hardware to or from buffers within the user-space application, similar to how direct I/O transfers operate in the standard block storage model. This means that data is not additionally copied into a kernel-level buffer, saving both the CPU time that would be used for the copy operation, and the memory overhead of requiring a second instance of the same data. Removal of the intermediate buffer does add the requirement that data to transfer must be aligned to the block structure of the underlying storage device, however. For example, if an SSD uses a 4 KiB block size for storage, the size of the data to transfer must be a multiple of 4 KiB, and the data must be aligned to a 4 KiB address in memory.

In order to maintain as much control over the storage operation as possible, each transfer is completed both *atomically* and *sequentially* within the kernel, with system calls blocking while data is transferred, and control only being returned to the calling application after completion. This control flow helps to keep the accountability of storage operations under the task that initiated the request, a property that is essential for real-time applications, as operations cannot then be completed asynchronously at an arbitrary later time and potentially cause interference to other tasks.

While improving predictability and accountability for individual storage operations, the lack of asynchronous transfers in CharIO is a trade-off that has the potential to be detrimental to the overall I/O performance of a system. This is due to the rest of the preemptive multitasking operating system working in a largely asynchronous manner, using kernel buffers and task-switching to utilise CPU time and memory bandwidth more effectively. CharIO's synchronous handling of I/O is necessary to provide the guarantee that data has been fully transferred before the return of a read or write system call, but leads to applications being blocked when they could potentially continue to run while data is copied in the background. It may be possible to create a compromise between these two approaches, allowing a CharIO application to optionally specify asynchronous I/O, and creating separate synchronisation points at which applications will block until transfers are complete, however this is not currently implemented.

4.3.2 Kernel Module Structure

Figure 4.3 shows an overview of the kernel module structure, with the various paths from a user-space application down to the NVMe SSD hardware. The main content of the module's source code is split across two C files. The first, *chario.c*, contains the main interfaces for the module, including its initialisation and exit routines, sysfs attributes used for triggering internal tests, and the main storage

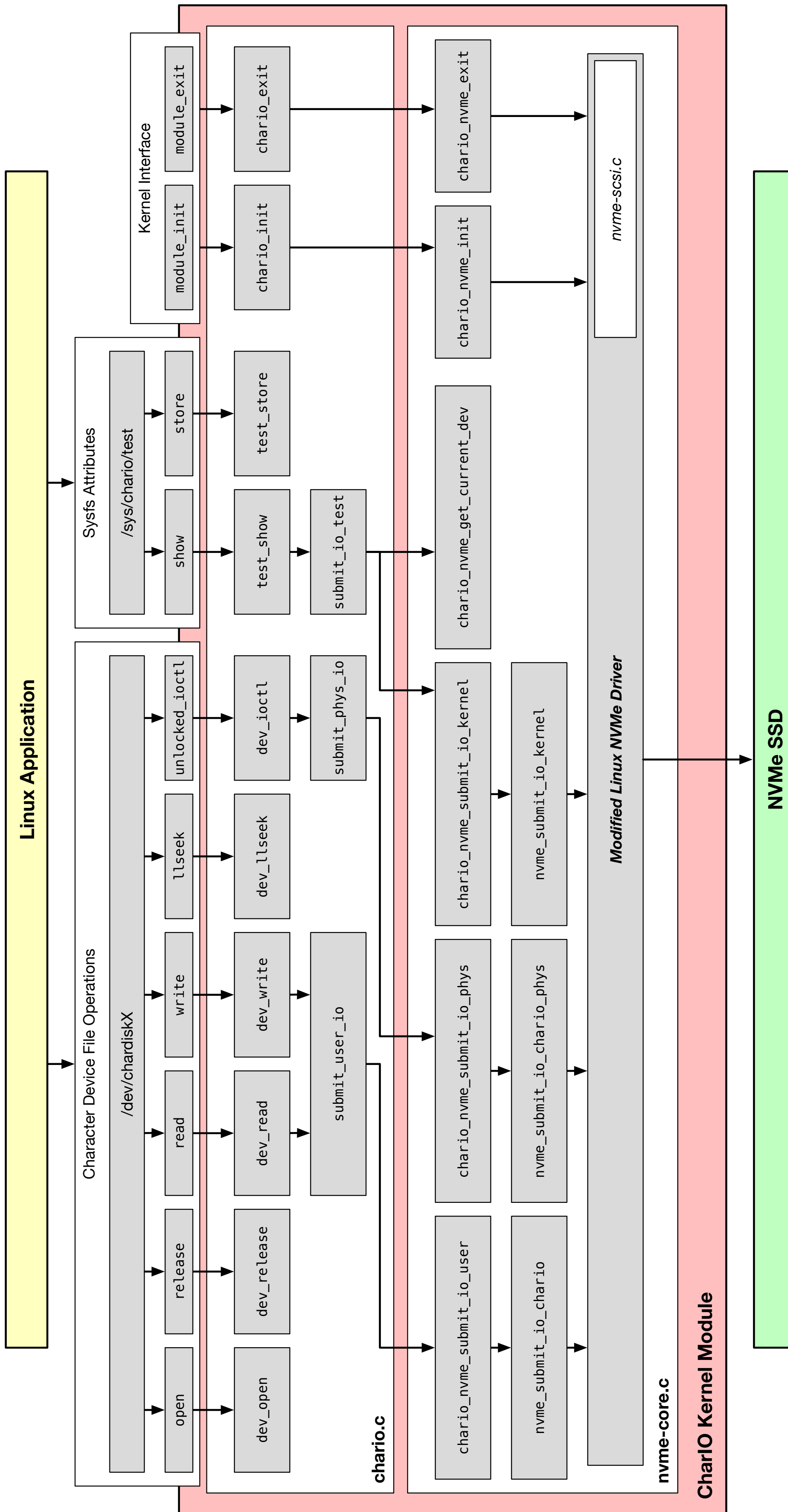


Figure 4.3: CharIO Linux kernel module structure

character device endpoints that are accessible through the VFS. The second, *nvme-core.c*, is a modified version of the core NVMe driver from Linux kernel v4.1.15, with additional functions added to support CharIO operations, as well as changes to the ways in which memory is allocated, and commands are created and submitted.

On module initialisation, the driver creates a character device in `/dev/` to act as the main interface to the storage device, as well as setting up the hardware via calls into the NVMe driver. The initialisation of the NVMe driver is almost identical to the standard kernel driver, other than not creating storage block devices or the device controller character device. In addition to the CharIO character device, an attribute item is created in */sys/chario/* to allow for an alternative method of accessing internal driver information and triggering kernel-level tests for debugging.

After initialisation, storage can be accessed through the character device as if it was a single, fixed, flat file with the full size of the underlying SSD, however I/O operations must be rounded and aligned to the physical block size of the device (for example, 4 KiB).

4.3.3 Accessing Storage

Storage is accessed through the `/dev/chardiskX` character device, using the following standard Linux VFS system calls, typically via standard library wrapper functions in an application.

`open`/`close` system calls

File open and close functionality is currently handled entirely by the VFS, with the CharIO driver registering file operations on open and release (called at the last close of a file) purely for informational purposes or future use.

`read`/`write` system calls

When a read is initiated on `/dev/chardiskX` from a user-space application, control is passed through the VFS to the CharIO driver, which then creates and sends requests on to the NVMe driver layer for processing.

The first task performed by the read function is to split the transfer into 4 MiB units, in order to remain compatible with existing NVMe and kernel DMA functions used at lower levels. These transfers are then processed one by one, beginning with the creation of a structure describing the operation, detailing the command, the number of device blocks to transfer, the starting block number, and the virtual address of data in memory. This structure is then passed to the NVMe layer, which performs the rest of the transfer.

The NVMe driver layer first sets up the calling application's data memory for a DMA transfer by the SSD controller, by pinning the necessary pages so they will remain available throughout the transfer, translating virtual memory addresses to physical addresses, and flushing or invalidating cache lines for the memory area. The transfer is then broken down further into smaller data units based on the maximum request size of the hardware device (for example, this is 32 blocks or 128 KiB on the Intel SSD 750 [16]). For each of these smaller units in turn, an NVMe device command structure is created, which is then submitted to the primary I/O queue of the storage device by copying it to a predesignated area of memory and notifying the device of its presence. The driver takes advantage of the hardware command queues present on the storage device, sending each new command immediately after the last, rather than waiting for it to complete first. This reduces the time spent waiting for the device, as further commands can be initialised while data is being transferred, while also eliminating the need for software queues in the kernel.

The completion of each command is indicated by an interrupt from the storage device, which is forwarded through the PCIe driver to the CharIO NVMe layer. Once all commands have been serviced and all data has been transferred, or if an error has occurred, control is passed back up through the driver layers to the application that originally made the system call, cleaning up any resources and releasing DMA memory pages along the way.

Writing data to the storage device operates essentially the same as reading, just with a different opcode passed in the NVMe command, and the storage device controller performing DMA in the opposite direction.

`lseek` *system call*

Seeking the device works the same as it would for any file or block device, changing the file pointer of the currently opened device instance to the specified offset, but with the caveat that the seek offset must be aligned to the underlying block size of the storage device, similar to reads and writes.

`ioctl` *system call*

As well as supporting any standard Linux character device *ioctl* calls, two additional calls are defined – `CHARIO_IOCTL_READ_PHYS` and `CHARIO_IOCTL_WRITE_PHYS` – allowing I/O transactions to be carried out on physical instead of virtual memory addresses in the system, which is potentially useful in an embedded environment, as described below.

4.3.4 Addressing Physical Memory

As well as standard file read and write operations between the CharIO character device and a buffer within an application, the driver supports transferring data directly between an NVMe SSD and any arbitrary physical memory address that is accessible to the device. This address may be outside of the main system memory managed by the Linux kernel, allowing for dedicated memory areas to be used as storage buffers without the overheads of Linux memory management for every transfer. In contrast, the standard Linux VFS read and write functions operate using the virtual address space of the calling process, and do not allow any I/O operations that bypass the page cache to access addresses that are mapped outside of paged system memory.

In order to accept physically-addressed transfers, the CharIO device node uses custom *ioctl* read and write commands that relay the address and the transfer size to the kernel driver. The same character device is opened and used as with all other CharIO operations, and the standard seek functions are used to specify the starting block for the transfer on the storage device.

Using the physical addressing mode means the kernel does not set up or manage page mappings or cache lines for the memory region being accessed, which can save significant processor time within the lower levels of the kernel driver. This mechanism can also be used to directly transfer data from physical addresses of other devices in the system, such as a network controller, sensor interface, or shared scratchpad memory, entirely avoiding copying data through main memory for these operations.

While the overall CharIO kernel module is capable of running on a variety of system architectures, physically-addressed transfers introduce the restriction that PCIe-attached storage is on a cache-coherent interconnect with the CPU, in order to maintain data consistency. This may only be available in certain embedded systems, for example using the Accelerator Coherency Port (ACP) interface on certain Arm processors, such as that on the Zynq-7000 SoC. The storage device must also have a direct view of the memory being used, which in some systems could require setting up an IOMMU, or other memory protection or translation unit, to allow this. One method of then mapping this external memory into Linux to be available to applications is using the Userspace I/O (UIO) driver, which allows physical memory regions to be defined for access through memory-mapped I/O on a device file.

Allowing applications to specify arbitrary physical addresses for data transfers clearly exposes potential security and safety risks for a system, as any areas of memory could be stored onto the SSD, and data from the SSD could be (accidentally or intentionally) forcibly written to any area of memory. While in an embedded system this

trade-off could be acceptable, as the software and hardware is likely to be more tightly controlled than in a general-purpose computer system, it could still be an issue if left unchecked. Potential solutions to this could be limiting the ranges of allowed physical addresses within the kernel driver (effectively how memory is managed in a general block device), potentially with a more secure interface to configure this, or placing hardware on the memory bus of the PCIe storage device, such as an IOMMU or a simple address validator, to limit its access to memory.

4.3.5 Potential Driver Enhancements

While a general proof-of-concept implementation of the CharIO driver is complete, there are areas that could potentially be improved in order to provide further efficiency gains, particularly regarding memory set-up and management. For example, for every read or write call to the module, the user-space pages of memory containing the buffer for the transfer, along with the area of kernel memory containing the NVMe command structure, are freshly set up for DMA. However, as the module has full control over memory allocations, it would be feasible to set up static DMA regions ahead of time and re-use them across requests, further reducing the amount of unpredictable activity involved in I/O operations. Modifications like this would, however, reduce the flexibility of individual transfer buffer locations, and cause a small amount more memory to be held active while the driver is loaded, in addition to further departures from the standard Linux kernel methods of handling memory.

4.3.6 User-space Library

In addition to accessing storage devices through CharIO directly using a `/dev/chardiskX` device node, a simple user-space, file system-like library has been developed, allowing a more-structured approach to accessing storage. To maintain the ideals of simplicity, efficiency and predictability from the CharIO kernel module, the library supports associating contiguous areas of storage with basic file identifiers, which can then be loaded, unloaded, or flushed to disk as required.

Internally, the library supports either standard read and write system calls for accessing the character device node, or specifying a physical memory address to use as a buffer location with the CharIO *ioctl* commands. This allows for buffers outside of kernel-managed memory to be used, such as sections of unmapped DDR RAM, or specialised high-speed or predicable buffers.

While using the CharIO character device as a flat file through the Linux VFS does not differ significantly from any other storage device, the user-space library offers a more convenient initial method of

providing some structure to the storage, while more sophisticated functionality could then be built on top of this base.

4.4 Evaluation of CharIO

To evaluate the CharIO driver, an experimental platform was set-up around an Avnet Zynq Mini-ITX development board [43]. This contains a Xilinx Zynq-7000 system-on-chip with 1 GiB of DDR3 SDRAM. The Zynq contains a dual-core 800 MHz Arm Cortex-A9 CPU [41], connected to an FPGA-based PCIe interface via the Zynq's Accelerator Coherency Port, with the PCIe connected to an NVMe SSD (Intel SSD 750 [16]). The Linux kernel is deployed on the Arm cores – specifically version 4.1.15-rt17, with PREEMPT_RT real-time patches [139] applied. A detailed specification of the experimental platform, including a diagram of system components and connectivity, can be found in Appendix B.2, and is the same base embedded platform used for analysis work in Chapter 3.

The custom profiling timer component described in Section 3.6, allowing high-accuracy, unobtrusive timing of events from the FPGA logic, was used to measure software and hardware execution times during CharIO's operation. The profiling timer 'tags' an event when software issues a write to a register in the peripheral, or when a specific hardware interrupt occurs, recording the time that the event happened. The only interference when collecting timing information is two 32-bit register writes to the core for each event tag. Events can then be read back from an FPGA buffer after the experiment is complete, ensuring measurements can be achieved consistently and without interruption (unlike with software timing functions). Kernel modifications allow events to be timed within kernel code, with these profiling points dynamically controlled through system calls.

4.4.1 Profiling CharIO Operations

A series of experiments were performed to examine how the low-level implementations of existing file systems and the block layer perform in the Linux kernel, and how this compares to the simpler alternative of CharIO. To measure this, simple periodic file read and write operations were performed from user-space, in a process set up with a real-time priority on an idle system. During the tests, the timing of key points in the data transfer was measured using the profiling timer component.

CharIO was tested against an ext4 file system (set-up using default parameters), as well as the block device created by the standard NVMe driver for the SSD. The ext4 and NVMe block device transfers were run using the `O_SYNC` flag set (ensuring the write operation blocks until data has been physically written to underlying hardware), and

both with and without the `O_DIRECT` flag set (enabling and disabling 'direct I/O' transfers). CharIO was tested both as a standard character device, and in its physically-addressed mode. Physically-addressed transfers were configured to use a buffer in the same physical DDR memory as Linux, but outside of the Linux system's virtual memory space, whose size was artificially limited using the device tree.

The results displayed in Figure 4.4 show the mean times across 10,000 experimental runs, taken when performing sequential read operations with transfer sizes of 40 KiB and 80 KiB. Timing measurements are plotted for when: the system call is entered ('read' or 'ioctl'), the I/O command is submitted to the SSD ('nvme_submit_iod-rd' or 'chario-cmd-submitted'), the hardware interrupt is triggered ('interrupt'), the kernel begins handling the interrupt, and the user-space application is resumed ('userspace'). The results for ext4 and block device transfers are the same as those from Figure 3.34 in Section 3.8, which are now additionally compared to CharIO.

The results show that CharIO spends less time performing computation in the kernel than ext4 or the block device, both before and after data is transferred. Overall transfer speeds remain relatively similar to direct I/O, and total latency is slightly improved against the other methods on the larger transfer. For all CharIO results, time is shifted more into an increased response period from the SSD, which may be due to the driver, such as how it submits commands, or simply due to external factors, such as memory or storage response time fluctuations. A shift of processing time from software to hardware can still be beneficial however, as it suggests a lower overall CPU utilisation. There is a further significant reduction measured in software time when using the physical addressing mode, due to the lack of kernel-level memory management required both before and after the transfer. This suggests that simplifying storage operations further through reducing the memory management overhead of transfers is potentially a very positive direction for development when considering software efficiency.

In addition to performing less work in the kernel, CharIO is far more predictable in how it interacts with the storage device compared to ext4 or the NVMe block device node. The CharIO driver produces a consistent, calculable number of device operations for every high-level command, based solely on the number of blocks read or written. In contrast, for large transfer sizes, the number of low-level commands involved in completing an ext4 or block device operation can be extremely unpredictable, even when using direct I/O. Figures 4.5 and 4.6 show the distribution of the number of PCIe interrupts (analogous to the total number of device operations) counted using the profiling timer hardware across 1,000 read and write operations of 512 MiB each, for block device, ext4, and CharIO. While reading, the number of interrupts does not vary greatly, with the highest range being

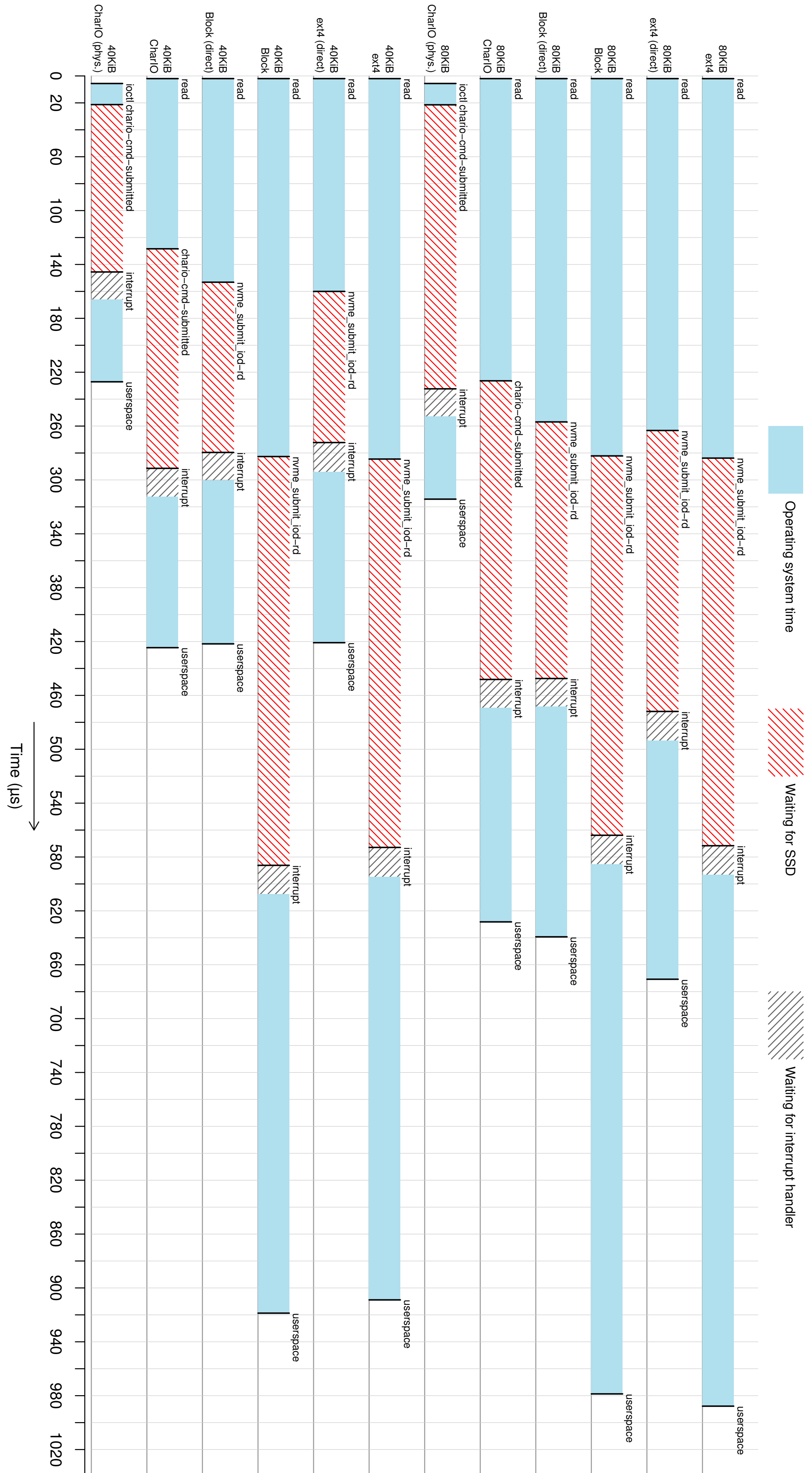


Figure 4.4: Operating system and hardware latency measured for 40KiB and 80KiB reads from ext4, block device and CharIO storage

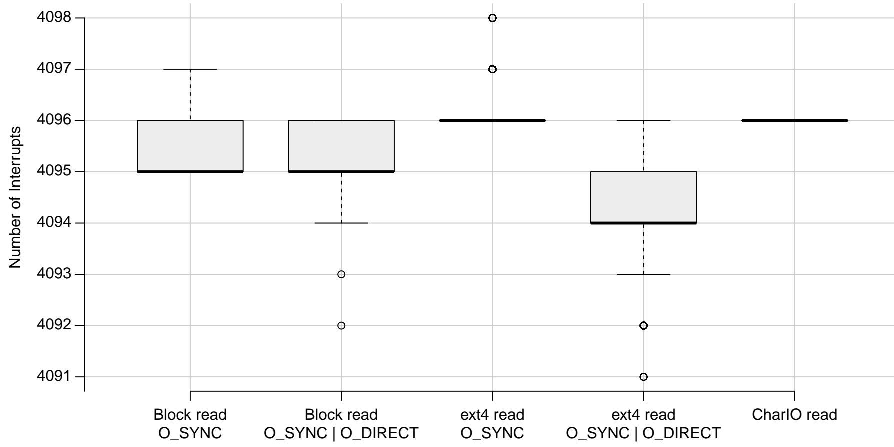


Figure 4.5: Box plot of interrupt counts across 1,000 512 MiB reads

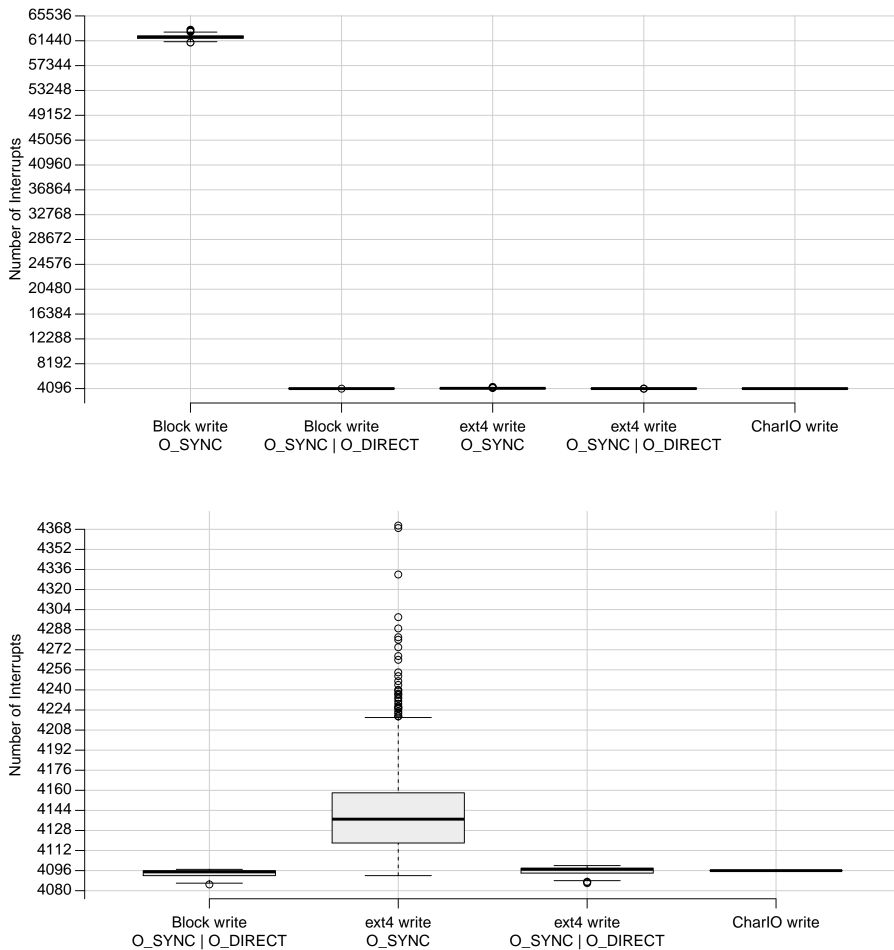


Figure 4.6: Box plots of interrupt counts across 1,000 512 MiB writes. Bottom plot shows the same data but with O_SYNC block device removed, so more detail can be seen across the other four types.

5, but CharIO is the only method that can guarantee a completely predictable number of operations (4,096 transfers of 128 KiB each). Interrupt count variability was found to be far more exaggerated when writing, due to effects of the file system, block scheduler, and other hidden kernel-level storage management. Standard I/O block device writes give the most unpredictable results, generating many smaller transfers to the disk compared to the other methods. Direct I/O shows an improvement in predictability when writing, but results still show a significant difference in the number of interrupts recorded over the runs. This behaviour means calculating meaningful average results for a write equivalent of Figure 4.4 is practically impossible, as access patterns and the number of commands sent to the SSD vary so greatly.

Figures 4.7 and 4.8 provide probability density functions of overall transfer time across 1,000 reads and writes from the CharIO device, for various transfer sizes, and using both standard I/O and physically-addressed transfers to a dedicated area of memory. These show similar patterns to those in Section 3.8, with distinct landscapes of peaks and troughs. Physically addressed results generally present slightly tighter distributions, however this is limited by the unpredictability of the SSD hardware itself, and the PCIe and memory interconnects between the CPU and storage.

4.4.2 Evaluating CharIO Performance

The basic read and write performance of the CharIO driver was evaluated by measuring transfer speed and CPU usage compared to an ext4 file system and an NVMe block device node operating through both standard and direct I/O on the same storage device.

The `dd` utility was used to perform 100GiB sequential transfers, with `/dev/zero` used as a low-overhead source of data for disk writes, and data read from the disk being discarded to `/dev/null`. During I/O operations, kernel CPU usage was periodically sampled using `dstat` to give an indication of the system load caused by the different storage interfaces. As the specific block size of a transfer can have a large effect when bypassing the page cache (as demonstrated in Chapter 3), a range of block sizes were tested.

Results from these experiments are shown in Table 4.1, including a ratio of speed/CPU usage, to give an indication of I/O operation efficiency. CPU usage is shown as a single-core percentage, with 200% indicating full utilisation of both cores in the system. From these results, CharIO shows the highest speeds for larger block sizes (for those tested over 4KiB), and uses the least kernel CPU time for all transfers. For 4KiB block-size transfers, CharIO outperforms both ext4 and block device direct I/O speeds, and is also faster than standard I/O on the block device when writing. This highlights how the complexity and inefficiencies in file systems and the Linux block

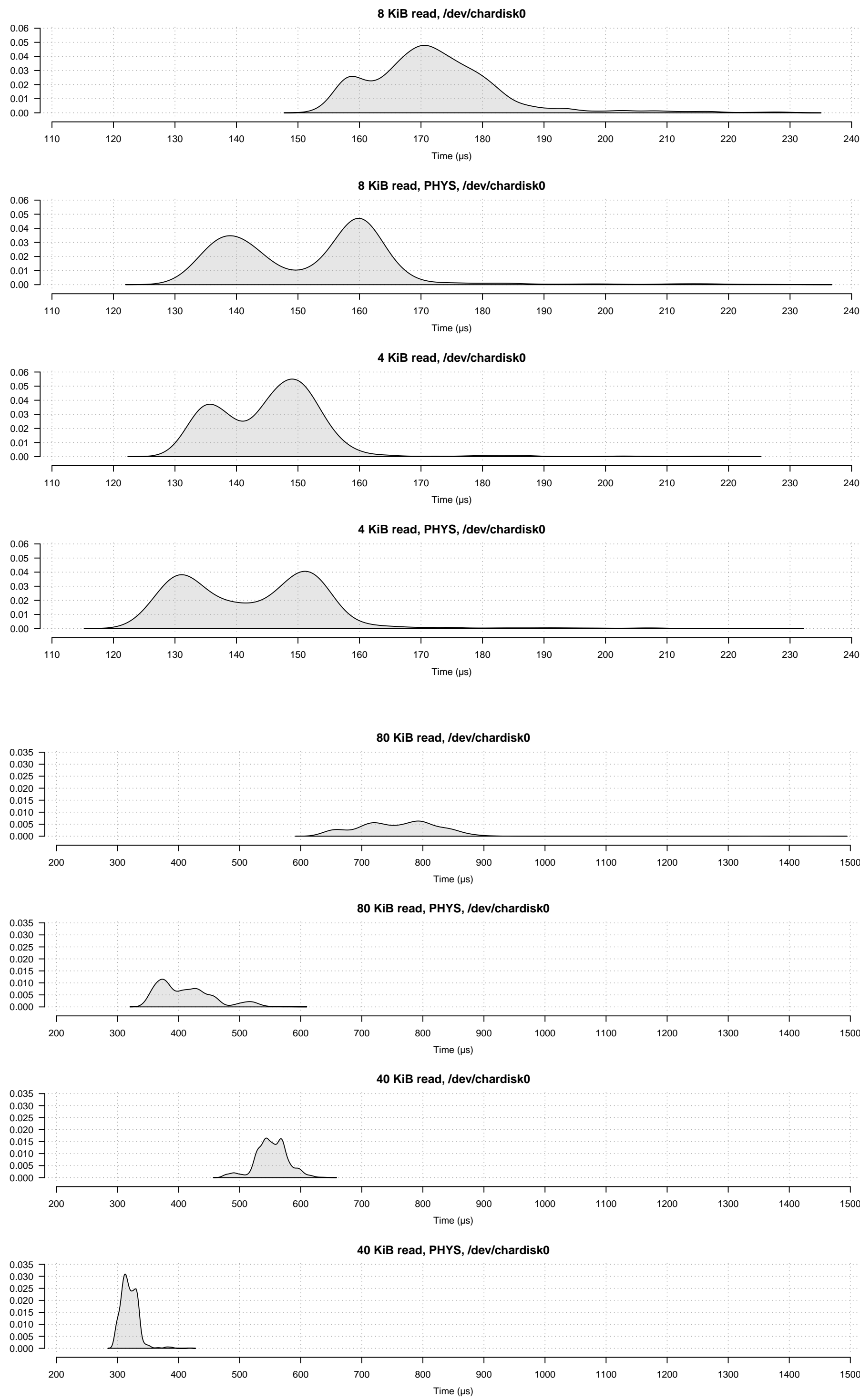


Figure 4.7: Probability density function of 8 KiB, 4 KiB, 80 KiB and 40 KiB read timings, with and without physical addressing, from CharIO device

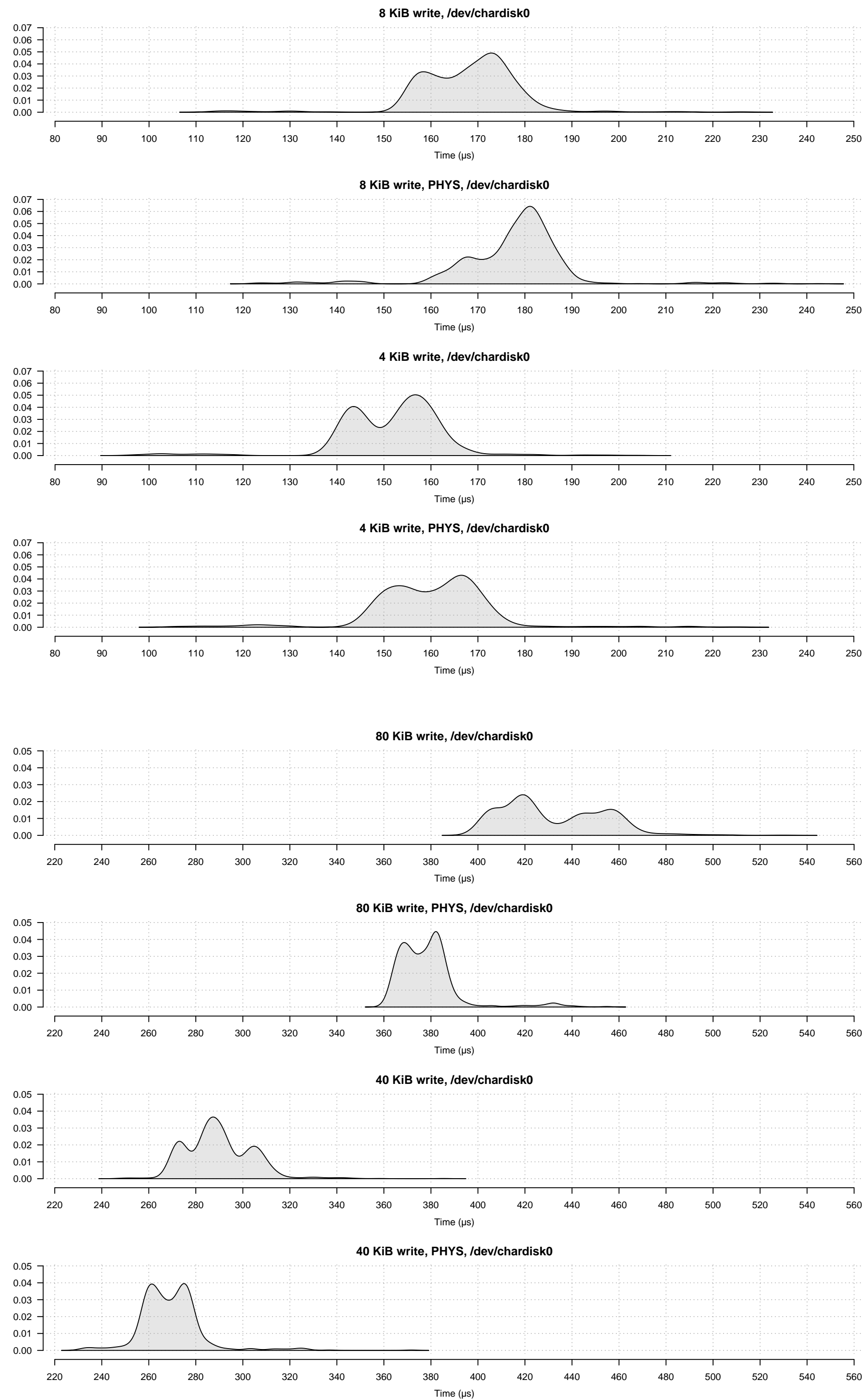


Figure 4.8: Probability density function of 8 KiB, 4 KiB, 80 KiB and 40 KiB write timings, with and without physical addressing, from CharIO device

Table 4.1: Comparison of storage access transfer speeds and CPU usages with CharIO, for different block sizes

	Speed (MiB/s)				System CPU usage (%)				Speed/CPU ratio			
Device and operation	4KiB	128KiB	1MiB	16MiB	4KiB	128KiB	1MiB	16MiB	4KiB	128KiB	1MiB	16MiB
CharIO read	51.0	**154.2**	**152.0**	**151.6**	**100.2**	**82.8**	**83.4**	**82.2**	0.51	**1.86**	**1.82**	**1.84**
Block device read	**130.6**	136.4	125.4	126.0	151.4	154.0	155.4	158.4	**0.86**	0.89	0.81	0.80
Block device read (direct I/O)	44.5	88.7	146.8	146.5	105.3	92.0	143.1	148.3	0.42	0.96	1.03	0.99
ext4 read	114.2	119.9	112.8	112.8	143.4	152.1	155.7	152.7	0.80	0.79	0.72	0.74
ext4 read (direct I/O)	43.2	146.3	130.8	146.2	107.0	83.6	143.5	145.2	0.40	1.75	0.91	1.01
CharIO write	48.7	**148.4**	**169.3**	**188.8**	**99.1**	**81.6**	**82.3**	**79.6**	0.49	**1.82**	**2.06**	**2.37**
Block device write	36.8	36.0	35.2	42.8	126.1	125.8	129.2	161.0	0.29	0.29	0.27	0.27
Block device write (direct I/O)	43.1	93.3	157.3	180.6	102.8	85.9	105.5	106.5	0.42	1.09	1.49	1.70
ext4 write	**85.0**	85.3	80.5	81.0	160.0	169.1	165.6	165.7	**0.53**	0.50	0.49	0.49
ext4 write (direct I/O)	21.5	132.8	143.5	175.2	102.5	85.6	104.9	107.0	0.21	1.55	1.37	1.64

I/O layer can have a considerable impact on storage performance in an embedded system due to high CPU usage.

4.5 User-space NVMe Driver for Embedded Linux

An alternative method of accessing a storage device in Linux is to create an entirely custom storage device driver, thus fully avoiding the VFS, exposing low-level hardware functionality to applications instead of file operations, and going one step further than CharIO's method of simplifying the kernel-level storage path. Creating such a custom driver deviates more significantly from the standard Linux storage model, but in doing so allows for a greater level of customisation and control.

4.5.1 User-space Storage Driver

While a custom storage driver could be constructed entirely in kernel code, using system calls to interface with applications, implementation could also be in user-space library code, with the kernel simply facilitating raw access to the buses and memory space of a storage device. One example of such a driver is UNVMe [131], an open-source user-space NVMe driver created by Micron Technology, built on top of Linux's Virtual Function I/O (VFIO) system, and using the Virtualization Technology for Directed I/O (VT-d) feature of certain Intel CPUs for access to a PCIe storage device via a supported system's IOMMU. An associated proof-of-concept user-space file system, User Space Nameless Filesystem (UNFS) [140], additionally provides a basic higher-level file interface above UNVMe's custom API.

The original design of UNVMe focusses on high-performance access to storage in a data centre-type environment, relying on specific virtualisation extensions to Intel's x86 architecture to access hardware, as well as an IOMMU in the system to safely remap PCIe device memory into a process's address space. While the basic idea of accessing a storage device directly from a user-space driver, and thus removing the kernel storage stack, could be desirable in the domain of embedded and real-time systems, the specific set-up of UNVMe is incompatible with many embedded architectures, where x86 CPUs with virtualisation extensions and IOMMUs are not common. To this end, *Embedded* UNVMe has been developed for this thesis by modifying the existing UNVMe codebase to remove its reliance on x86 hardware virtualisation and IOMMU support, allowing its use on a wider variety of system architectures.

Embedded UNVMe initially targets Linux running on the ARM64 CPUs of the Xilinx Zynq UltraScale+ MPSoC architecture [42], in order to evaluate and test the feasibility of running a user-space NVMe driver in a relatively powerful and modern embedded context. Full

source code for the Embedded UNVMe driver, with support for the Zynq UltraScale+ platform, is available online at [8].

4.5.2 Porting UNVMe to the Zynq UltraScale+

The Zynq UltraScale+ MPSoC architecture is similar to the Zynq-7000 platform described previously in this thesis, but with a quad-core Arm Cortex-A53 64-bit CPU replacing the dual-core Cortex-A9, a larger array of auxiliary peripherals and interfaces, and a greater amount of FPGA logic attached. Full details of the Zynq UltraScale+ platform, and the Xilinx ZCU102 development board [44] used for implementation work, can be found in Appendix B.3.

The first step in porting the driver away from x86 was removing all architecture-specific inline assembly instructions from the code, such as timing events using the Time Stamp Counter (TSC) register. These register-level timing functions were replaced with generic Linux clock functions from the C time library (`time.h`), specifically, reading the system's `CLOCK_MONOTONIC_RAW` value using the `clock_gettime` function. This has a slight overhead compared to using CPU-specific timers, however this is insignificant for its uses within the UNVMe code, and still provides a monotonically increasing time value, while being compatible with a much greater number of systems.

Secondly, due to the lack of Intel VT-d and an IOMMU, an alternative way to access raw system memory from user-space is required for the driver to operate. The NVMe protocol uses an area of memory shared by the CPU and storage hardware for communication, primarily through the CPU creating command queue data structures which are subsequently read by the storage device using DMA, and the storage device writing the status of transfers back into a completion queue structure. Alongside this, access to PCIe configuration memory is required by the driver for setting-up the storage controller, and agreeing upon the parameters and location used for the shared memory space.

While the Linux VFIO driver can still be used in 'No-IOMMU' mode to configure PCIe devices when an IOMMU is not present in the system, this removes the ability for user-space programs to negotiate the mapping of arbitrary memory areas, such as that required for the NVMe communication data structures. Therefore, VFIO continues to be used as a safe and convenient way to configure the PCIe interface of the NVMe device, but an alternative is needed for accessing memory. For this, the Linux Userspace I/O (UIO) system was employed, which allows an area of memory to be mapped though a device file (for example, `/dev/uio0`), with the kernel simply creating the virtual-to-physical address map and managing caches to allow for DMA operations. This memory area is configured using a basic loadable kernel module, which may be configured either with a fixed location as a platform device, or using an entry in the system's device tree

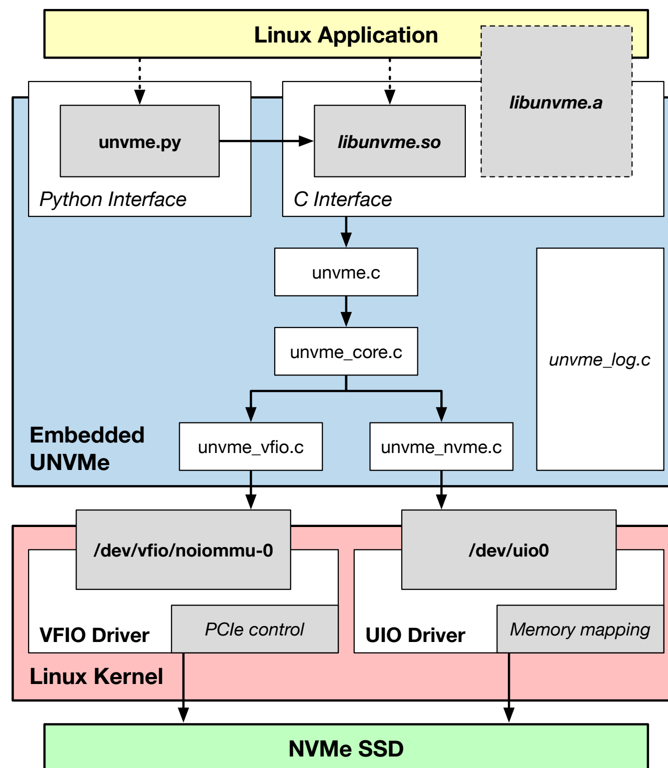


Figure 4.9: Architecture of the Embedded UNVMe driver

to both reserve the memory and assign it to the custom UIO driver simultaneously. The user-space driver can then allocate space for transfers and create command queues in this reserved memory area with virtual addressing, while being aware of the physical memory mappings to use with the storage device commands. An example of both a platform device and a device tree UIO driver for use with Embedded UNVMe are available online at [8].

4.5.3 Embedded UNVMe Driver Operation

A high-level overview of the Embedded UNVMe driver architecture, showing its application interfaces and route to communicating with a storage device via the Linux kernel, is shown in Figure 4.9.

The Embedded UNVMe library can be built either as a static library to be copied into an application at compile-time, or as a dynamically-linked shared object library that can be referenced at runtime. The main interface into the driver is then accessed directly through a number of functions defined in the library, all executing in user-space other than low-level memory-mapping and PCIe control operations. This is in contrast to a kernel-level storage driver, which must be installed as a separate loadable kernel module (or compiled into the kernel binary itself), then interfaced through system calls.

Storage is primarily accessed through high-level functions defined in `unvme.h`, allowing opening and closing devices, allocating and freeing memory, reading and writing data, and submitting generic or vendor-specific commands to the storage device. Commands can either be submitted synchronously or asynchronously, with synchronous commands blocking until the completion of the associated operation, and asynchronous commands returning immediately after submission, then relying on later library calls in the application to poll the operation status and access any returned data. In addition to these high-level functions, the driver's internal lower-level NVMe functions can be used directly from an application, allowing more complete control over features such as memory management and command queueing. As well as the C interface and sample code, a Python wrapper is provided as an example of allowing straightforward access to the storage driver from a higher-level scripting language.

On initialising an NVMe storage device from an Embedded UNVMe application, the PCIe device is deregistered from the kernel's NVMe driver (if appropriate), and set-up for use with the user-space driver by registering it as a `vfio-noiommu` device. An appropriate UIO driver must also be loaded in order for an area of shared physical memory to be directly accessible from the user-space application. Once a UIO driver has been loaded, and the PCIe device has been bound to the VFIO driver, calling the `unvme_open` library function from within an application will configure the storage device for UNVMe, and create the necessary data structures in the shared memory for commands and transfers. From this point, memory blocks can be allocated for transfers within the shared space accessed through UIO, and data can be read from or written to the storage device. More advanced functionality can be also be achieved, such as creating commands that operate similar to the physically-addressed transfers of CharIO.

4.5.4 Embedded UNVMe Driver Limitations

One major limitation of the Embedded UNVMe driver, compared to an in-kernel storage driver, or even traditional UNVMe using an IOMMU, is the lack of system-level memory protection. While any standard NVMe hardware device still has the potential to read or write data from any physical address using DMA (assuming an IOMMU is not used), a kernel-level storage driver can at least provide privileged software protection against this, sanitising addresses before issuing commands to a device, and preventing an application accessing forbidden areas.

Access to full system memory by the NVMe device (and therefore by the Embedded UNVMe application) has implications in both security and system integrity. In terms of security, it gives the potential for one application to read or modify the memory of another, or even

of the operating system itself, an ability that is usually prevented by the kernel's management of virtual memory. Regarding system integrity, an application may intentionally or unintentionally corrupt areas of memory that it would not normally have access to, via the NVMe device, potentially causing issues for any software running on the system. Within the area of real-time embedded systems, memory protection can be especially important in mixed-criticality systems, where tasks of different criticalities may have different levels of access to memory, and strict separation of memory is an important consideration.

The practical implications of this lack of memory protection are limited, however, as an appropriately privileged application could potentially read memory in a more direct manner anyway (for example, using `/dev/mem`) and an embedded system is often operated as a more controlled environment than a general purpose computer in terms of software. It may also be possible to prevent rogue memory transfers by restricting access on the memory bus used by the PCIe hardware to specific address ranges, either through configuration of that bus in the SoC hardware, or through additional hardware programmed into the FPGA fabric to monitor and control the bus activity, such as that used between components in the PHANTOM project FPGA architecture [141].

A further limitation of a fully user-space driver is a lack of access to kernel-level interrupt routines, meaning the completion status of a command is polled though reading the head of the completion queue in memory. While this is not necessarily as efficient as interrupt-driven I/O, it does remove an element of unpredictability associated with the high-priority, pre-emptive nature of interrupt routines which may be undesirable in a real-time system. It is possible to set up interrupt handling through the VFIO or UIO kernel drivers, by registering a callback function with an *ioctl* or monitoring the UIO device file respectively, but these push more work and control into the kernel and away from the library, and complicate the implementation of the driver.

4.6 Evaluation of Embedded UNVMe

The Zynq UltraScale+ Embedded UNVMe implementation was tested using the Intel SSD 750 [16] used in previous experiments, in order to compare its raw performance to a standard NVMe block device.

4.6.1 Basic Full-disk Read/Write Performance

As a basic test of storage throughput, the entire 400 GB SSD was read from and written to sequentially, both as a standard NVMe block device using the GNU `dd` utility, and through the Embedded UNVMe

Table 4.2: Sequential transfer speeds for NVMe block device (with and without direct I/O) and Embedded UNVMe on Zynq UltraScale+

Interface (block size)	Read (MiB/s)	Write (MiB/s)
Block device (128 KiB)	798.21	646.46
Block device (512 MiB)	730.21	514.08
Block device direct I/O (128 KiB)	803.31	690.80
Block device direct I/O (512 MiB)	1377.78	763.61
Embedded UNVMe (128 KiB)	**1444.34**	**1004.58**

driver. For UNVMe, each individual transfer is 128 KiB, as this is the maximum size of any single I/O command sent to the SSD used for the test. For the block device, block sizes of both 128 KiB and 512 MiB were tested, in order to provide both a direct comparison and an extreme example where the kernel has access to a large block of data respectively. The code used for benchmarking using UNVMe can be seen in Appendix C.2, which also serves as an example of how to use the library for basic I/O operations.

Results from this benchmarking are presented in Table 4.2, showing the UNVMe driver performing significantly faster than the block device. Direct I/O on the block device is faster than standard I/O, likely due to reduced memory copy overheads from not using the page cache. This is especially apparent with the larger block size transfers, which greatly improves the speed of direct I/O, but reduces the speed of standard I/O. Using a very large block size reduces the overall overhead caused by context-switching into the kernel for every storage operation considerably, as well as increasing the efficiency of setting up areas of memory for DMA transfers, but for standard I/O this probably also overwhelms the page cache, leading to the slowdown. These trade-offs do not require consideration for Embedded UNVMe, however, as its storage operations are performed entirely in user-space code, and its operating memory is mapped and reserved ahead of time through the UIO system.

4.6.2 Benchmarking Embedded UNVMe with FIO

Further benchmarking tests were performed using FIO [126], to compare sequential and random read and write performance of the Embedded UNVMe driver to an NVMe block device in more depth, while using multiple threads and taking advantage of NVMe hardware queues. The UNVMe library provides an FIO *ioengine* plug-in, which was used for these tests, along with the Linux *libaio* engine for asynchronous block device transfers.

Table 4.3: Random transfer speeds for NVMe direct I/O block device and UNVMe driver on Zynq UltraScale+

		Per-thread CPU usage (%)		
Driver/operation	Speed (MiB/s)	User	System	Total
NVMe block read	744	8.57	16.40	24.97
UNVMe read	**1415**	24.99	0	24.99
NVMe block write	**559**	6.70	12.30	19.00
UNVMe write	445	24.98	0	24.98

Initial FIO Benchmarks

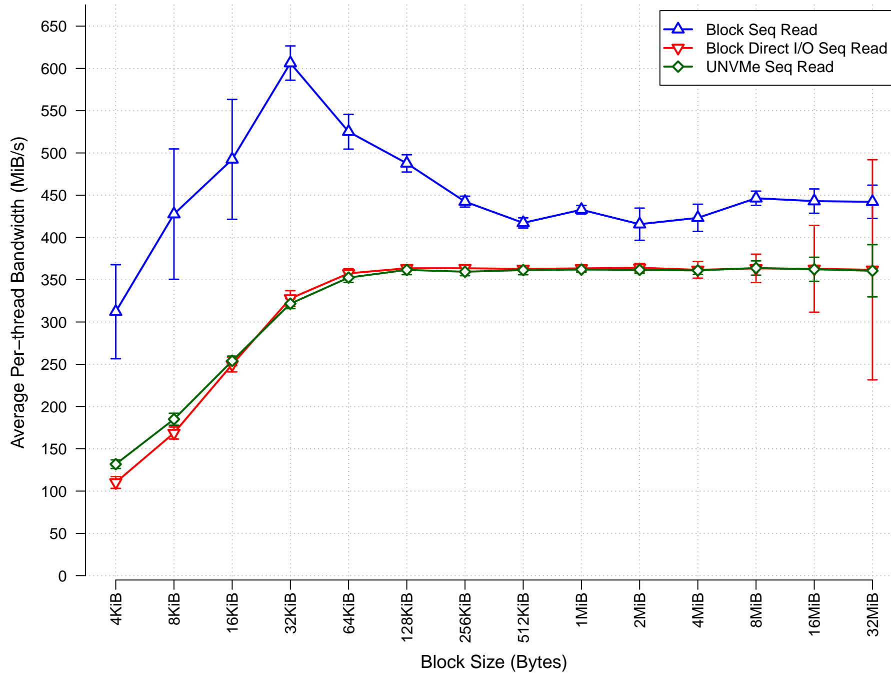
A limited initial test was performed using the the default FIO benchmark tests distributed with UNVMe source code, measuring speed and CPU utilisation while performing random transfers across 16 threads and 32 hardware queues, with a 4 KiB block size, and running for 120 seconds each. A summary of the results of these tests is shown in Table 4.3.

Despite the multithreaded and multi-queue nature of these benchmarks, the small block size and random I/O pattern reduces performance compared to the basic sequential tests. While the direct I/O block device results still show an improvement over their standard I/O sequential equivalents, block device read speed is affected far more than UNVMe. UNVMe writes suffer most significantly in these benchmarks, where speeds fall behind those of the block device, likely due to the UNVMe driver performing writes in an entirely synchronous manner, unlike the block layer scheduler.

Each test fully utilises the four CPU cores on the platform (with the sum of the 16 threads totalling 400% usage), except for the block device write, which leaves almost an entire core free. The high CPU utilisation of UNVMe is partly due to it actively polling for the completion of each I/O operation. In a real application, this time could potentially be yielded to the scheduler, or used for other processing tasks within the application. The UNVMe driver CPU usage is also entirely in user-space, whereas the block device is split between user-space and the kernel at around a 1:2 ratio. This attribute of UNVMe may be desirable in a real-time system, as pushing CPU time into user-space can aid with task resource usage accountability, and the pre-emption of user-space code is far more trivial than the kernel.

Detailed FIO Benchmarks

Following these tests, a more thorough investigation into the performance of Embedded UNVMe on the Zynq UltraScale+ platform was performed, with measurements taken for multiple block sizes of random and sequential read and write operations, similar to the



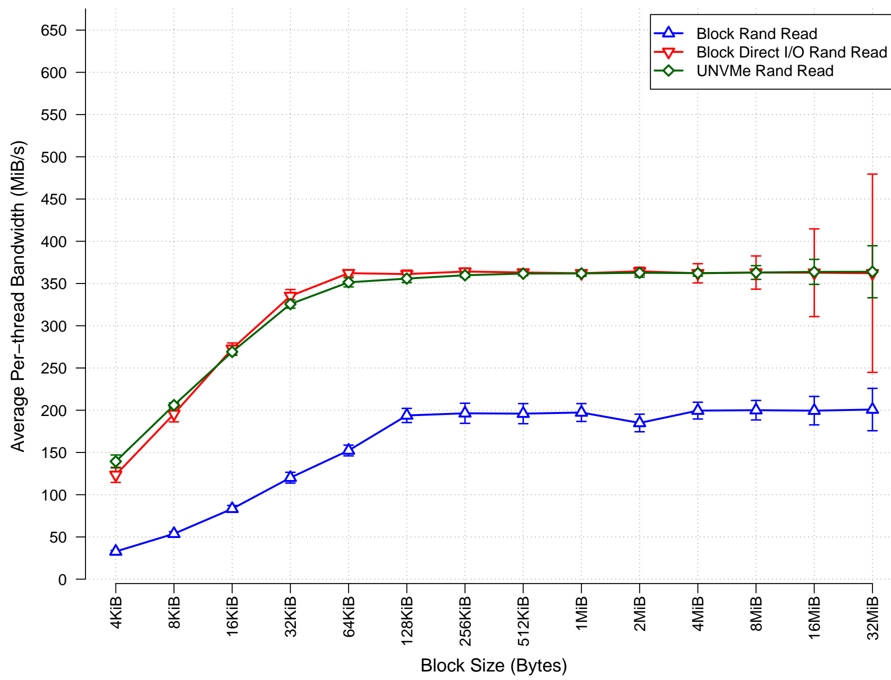
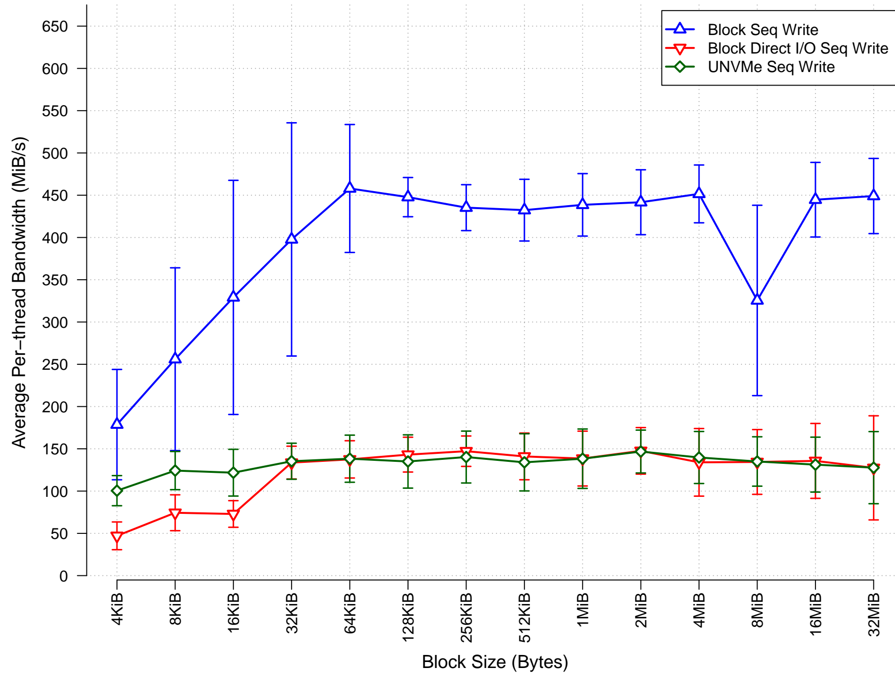


Figure 4.10: Read speeds for NVMe block device and UNVMe on Zynq UltraScale+ platform (x-axis log scale, error bars show standard deviation)



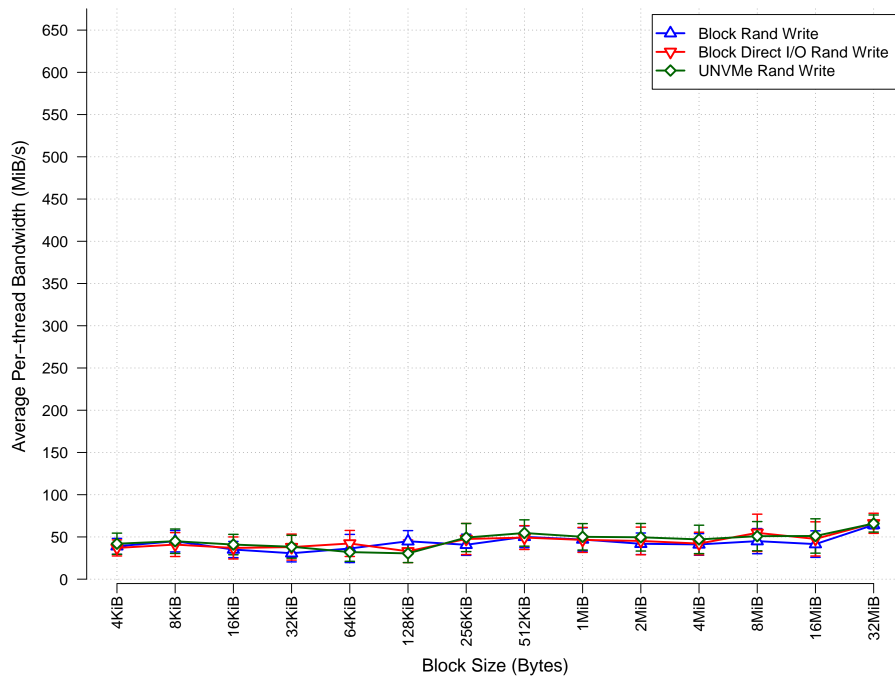


Figure 4.11: Write speeds for NVMe block device and UNVMe on Zynq UltraScale+ platform (x-axis log scale, error bars show standard deviation)

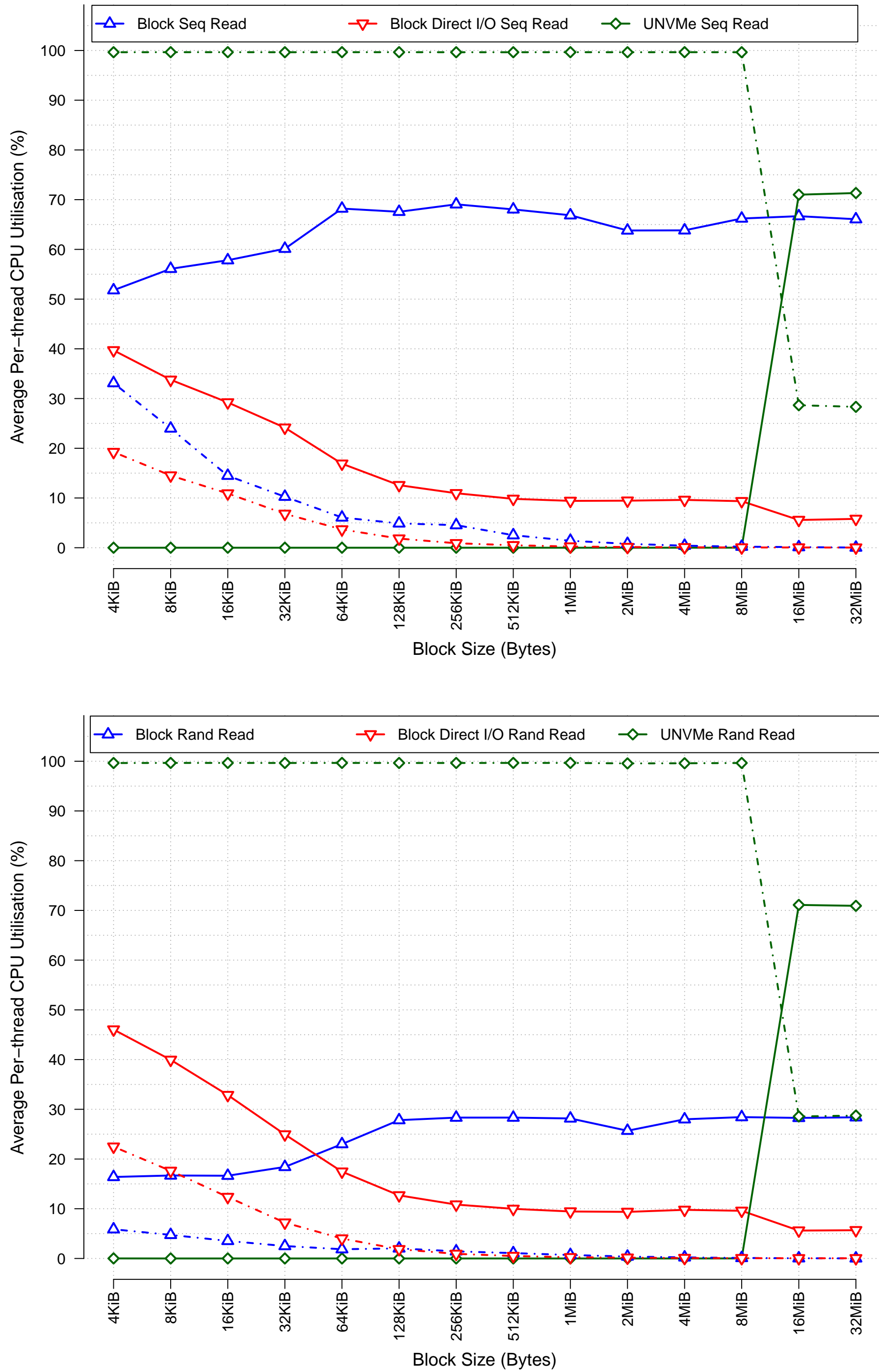


Figure 4.12: Kernel and user CPU utilisation for NVMe block device and UNVMe read operations on Zynq UltraScale+ platform; solid line shows kernel, dashed line shows user (x-axis log scale)

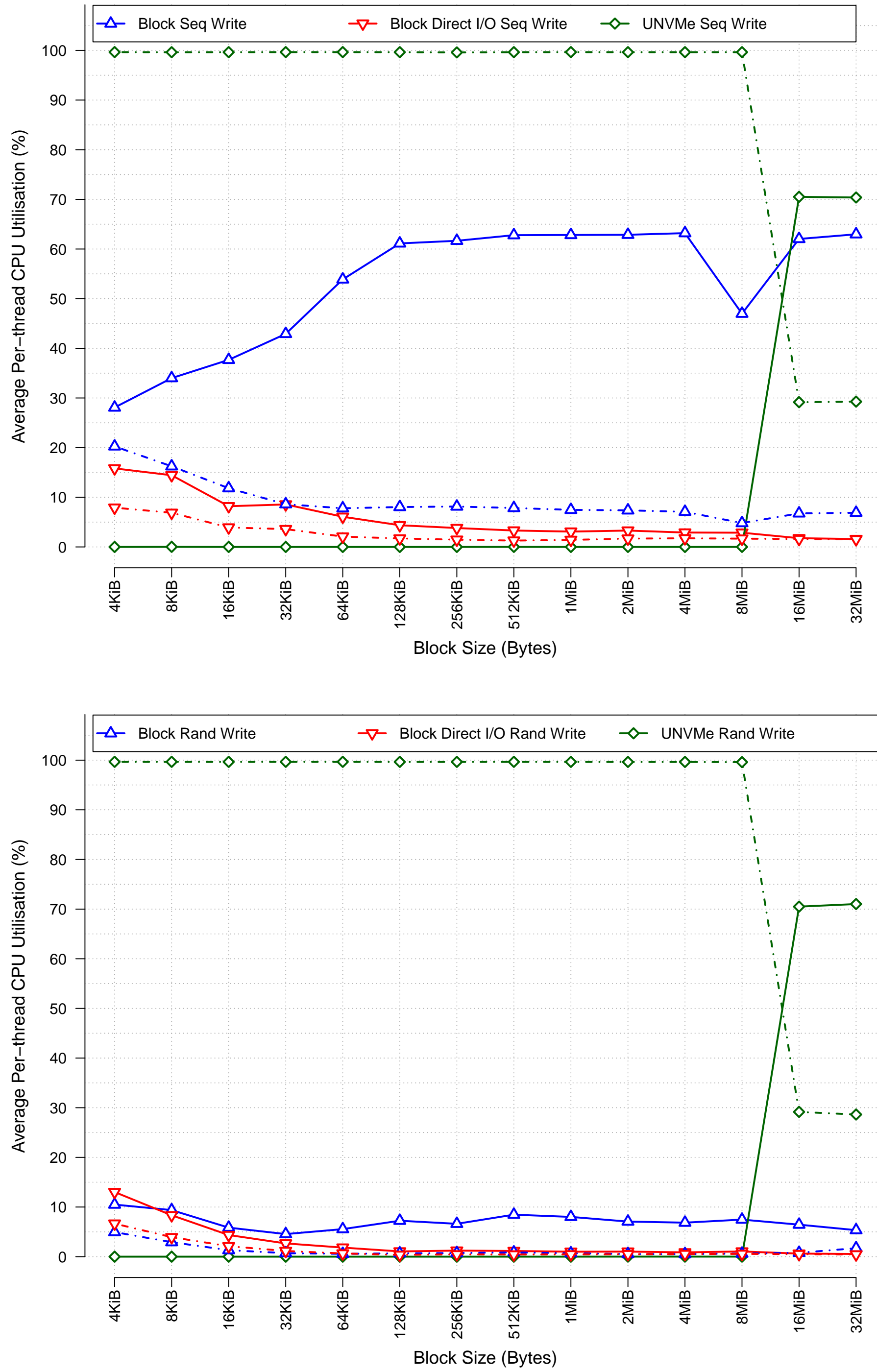


Figure 4.13: Kernel and user CPU utilisation for NVMe block device and UNVMe write operations on Zynq UltraScale+ platform; solid line shows kernel, dashed line shows user (x-axis log scale)

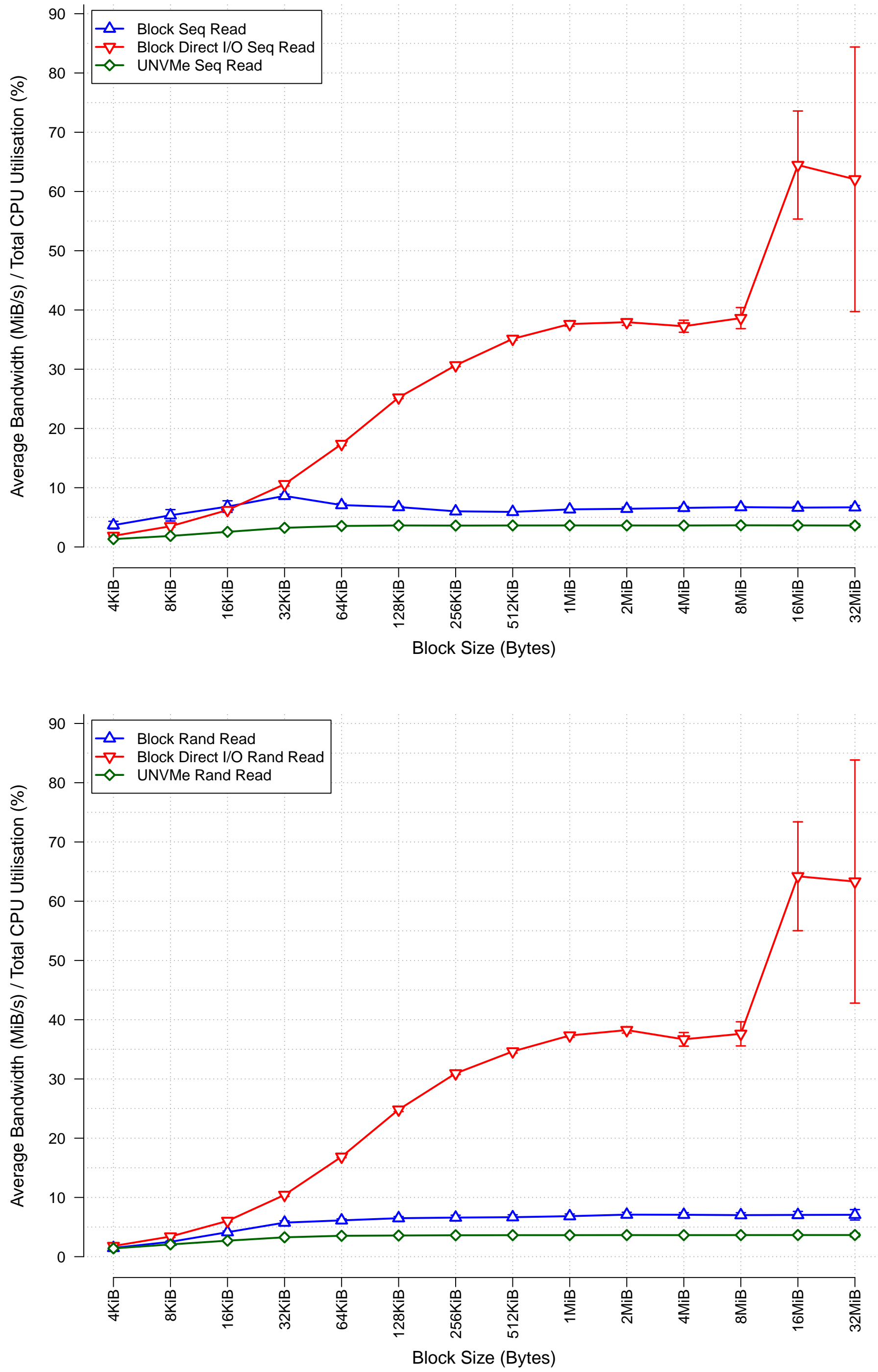


Figure 4.14: Read speed per percent of CPU utilisation for NVMe block device and UNVMe on Zynq UltraScale+ platform (x-axis log scale, error bars show standard deviation)

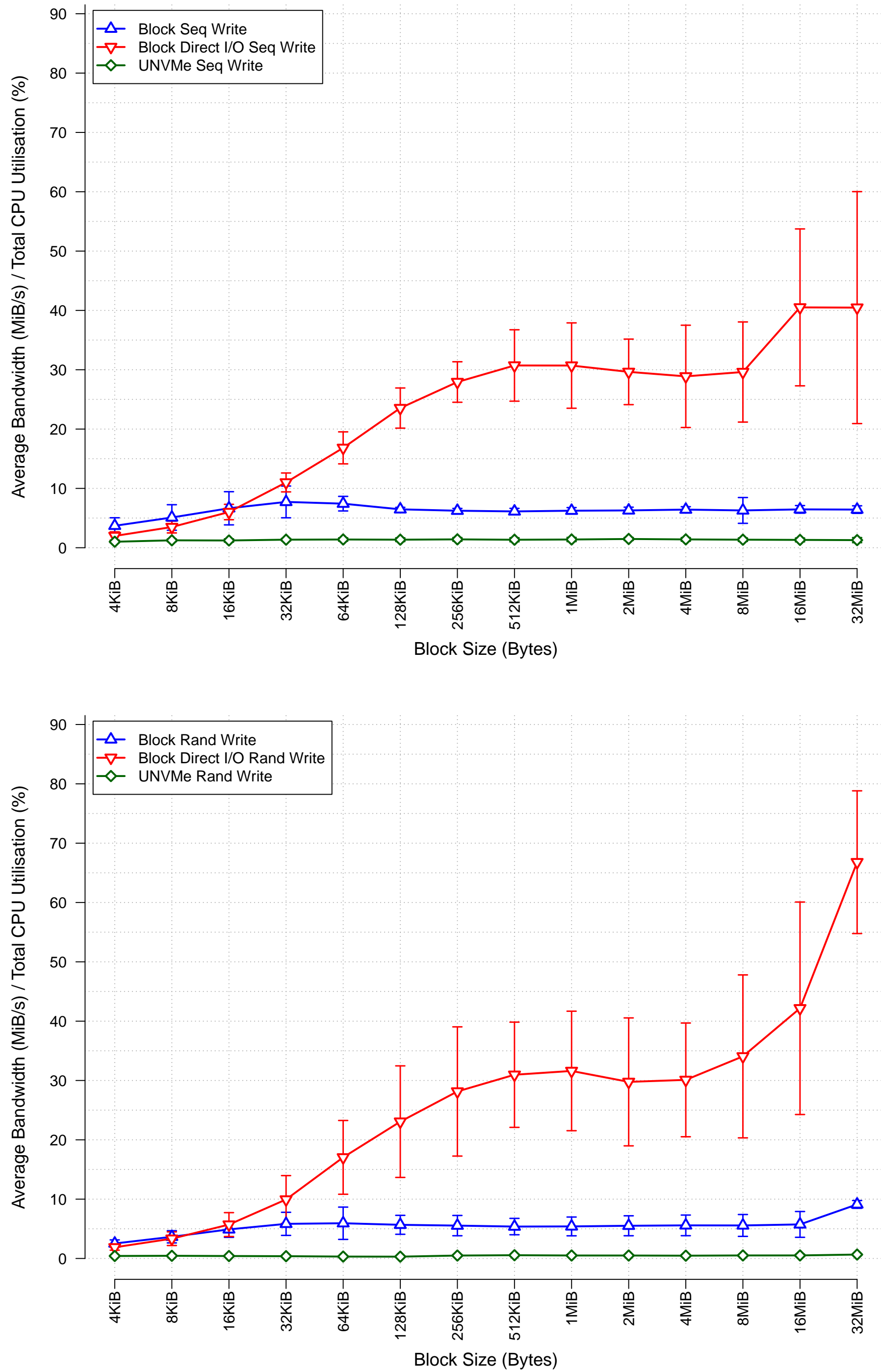


Figure 4.15: Write speed per percent of CPU utilisation for NVMe block device and UNVMe on Zynq UltraScale+ platform (x-axis log scale, error bars show standard deviation)

experiments presented previously in Sections 3.4 and 3.5. Additionally, block device transfers were run with and without direct I/O, in order to provide more insight and data for comparison. In order to create a thorough high-performance stress-test of the I/O system, but with a level of parallelism appropriate for the platform, tests were run with an I/O queue depth of 4 and using 4 threads of activity (across the 4 CPU cores). Due to the set amount of shared memory reserved for the Embedded UNVMe driver, and the pre-allocation of I/O memory buffers, the maximum size of all transfer blocks and associated device queues cannot exceed 1 GiB, limiting any power-of-two transfers to 32 MiB when using 4x4 queues. Minimum block size is limited to 4 KiB, as this is the hardware sector size used by the SSD device.

Figure 4.10 shows the average per-thread read bandwidth of the transfers for each pattern, block size, and device, with overall bandwidth being four times this. UNVMe and direct I/O block device transfers perform very similarly, with both sequential and random access patterns, increasing in speed until around 128 KiB (the maximum data size that can be transferred with a single NVMe command on the SSD being tested), and holding relatively steady after this point. There are two key differences observed between these two interfaces, however: firstly, UNVMe performs slightly better at smaller block sizes, likely due to the reduced overheads of not requiring a context switch into the kernel for every operation; and secondly, the variability of block device transfers, shown by the standard deviation, is far higher, especially for large block sizes, likely caused by the overheads of allocating transfer memory on the fly.

For sequential reads, the standard I/O block transfers perform much better for all block sizes, albeit with large standard deviations at smaller block sizes. This can probably be explained by the Linux *libaio* interface handling the sequential, asynchronous, cached transfers from multiple threads in a much more intelligent manner than the more rigid direct I/O and UNVMe transfers, allowing creative scheduling of requests, and potentially the re-use of data if it is present in the page cache when a second thread requests it. Performance peaks at 32 KiB, then reduces dramatically until levelling out above 512 KiB, suggesting that above this point the page cache is becoming less effective for the specific benchmark operations being performed, perhaps because there is less possibility for data to be reused, and that more pages must be purged and reallocated before each transfer. Random reads are very different, however, with the standard I/O block device performing consistently significantly worse than the other access methods. This suggests that any advantages in using the asynchronous I/O scheduler and page cache can be greatly reduced when dealing with more random access patterns.

Figure 4.11 shows the same per-thread bandwidth results, but for write tests. Sequential write results show very similar patterns to

sequential read, but with an overall slower speed for direct I/O and UNVMe, and much larger standard deviations, particularly for standard I/O block device transfers. The very large standard deviations for standard I/O block transfers are accompanied by the possibility of large outliers, although this still performs significantly better than direct I/O or UNVMe, likely due to the specific benchmarks taking good advantage of the page cache and *libaio* scheduler for multiple threads. Both UNVMe and direct I/O perform more consistently across all block sizes, but the drop for small block sizes is much more significant for the block device. Random write results are almost identical (within noise) for all three storage access methods, and perform relatively consistently and slowly across all block sizes, suggesting that the storage device itself may be more of a bottleneck here than any software involved in accessing it.

The CPU performance results shown in Figures 4.12 and 4.13 largely demonstrate what would be expected in these benchmark scenarios, based on results presented previously in this chapter and in Section 4.4. For example, standard I/O transfers generally use more CPU time than direct I/O, CPU utilisation generally reduces as block size increases (other than for certain standard I/O tests), and the majority of block device CPU time is spent in the kernel. It would be expected that UNVMe uses an entire core per thread of user CPU time, while using no kernel CPU time, due to performing all of its processing in user-space, and to constantly polling for the completion of commands in any potential idle time during these benchmarks, which is indeed the case for the majority of results, however in all tests, there is a large shift in user to kernel CPU time for UNVMe at 16 MiB and 32 MiB block sizes. As the UNVMe driver does not explicitly perform any computation in the kernel, this shift must be due to a side-effect of the storage operations, possibly related to the scheduling of multiple threads of simultaneous I/O, or processes internal to the UIO or VFIO kernel drivers. A more in-depth profiling of UNVMe and kernel code could provide further insight into the specific cause of this behaviour if required.

Figures 4.14 and 4.15 show an indication of the efficiency of the storage operations, represented as the average speed obtained per percent of total CPU utilisation, for each I/O thread. UNVMe results are largely irrelevant, as the driver will use 100% CPU in these experiments regardless of its actual efficiency, thanks to actively polling for command completion. It would be possible to measure the proportion of wait and busy processing time separately through modifying the UVNMe driver, but this is outside the scope of FIO. The standard I/O block device performs relatively consistently across all block sizes in all experiments, with a slight efficiency drop below around 32 KiB, and a slight increase at 32 MiB for random writes. In contrast, the efficiency of all direct I/O block device results scale largely with block

size, with small reductions in CPU usage at 16 MiB and 32 MiB being amplified in the efficiency measure. This increase in efficiency with block size is probably mostly due to the number of context switches and separate memory allocations reducing as block sizes get larger, which is enforced by the rigid operation of direct I/O in the kernel.

4.7 NVMe Coprocessor for Embedded Systems

Due to the tight coupling of CPUs and FPGA logic in the Zynq UltraScale+ MPSoC, it is possible to offload software functionality from the hard CPU cores onto dedicated hardware or soft coprocessor cores, while still retaining the same view of system memory and peripherals. These peripherals include the PCIe controller of the SoC, therefore giving the potential of direct access to an NVMe SSD from the FPGA. Through taking advantage of this, a Xilinx MicroBlaze soft processor core [142] has been implemented in FPGA logic alongside Linux on the Arm cores, in order to run NVMe commands independently of the main CPU, thus acting as a dedicated storage controller coprocessor. A full overview of the system architecture of this design, including a block diagram describing the connectivity of the MicroBlaze with the Zynq processing system, can be found in Appendix B.3.

In order to control the storage device from the FPGA hardware, and independently to the main CPU, Linux running on the Arm cores must relinquish direct control of the NVMe device and PCIe controller. The most straightforward way to achieve this is to remove the PCIe definition from the system's device tree, meaning the Linux kernel has no knowledge of its presence at all when it boots, leaving it free to be configured by another part of the system. Another possible approach could be to use the hot-plugging potential of PCIe to dynamically transition control of the storage hardware between Linux and the FPGA, however this would likely require modifications to Linux kernel drivers in order for it to operate in a stable manner.

Alongside this, the MicroBlaze software must have the ability to control both PCIe and NVMe interfaces at a low level. This is achieved through bare-metal code running on the MicroBlaze, based on the Embedded UNVMe driver, accompanied by a basic driver to set up the Zynq UltraScale+ MPSoC PCIe peripheral. A thorough series of tests were also written to check the operation of the storage interface, with results compared against known correct values from UNVMe on Linux, as well as benchmarks to provide basic performance metrics. Full source code for the MicroBlaze NVMe driver, including a custom driver for controlling the Zynq UltraScale+ PCIe controller peripheral, is available online at [9].

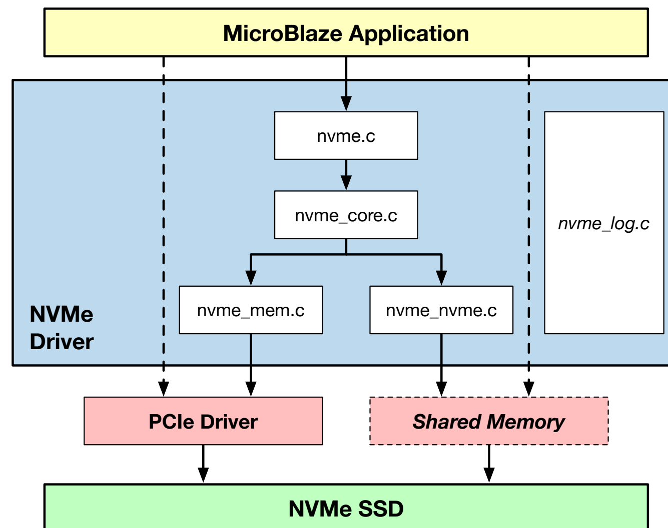


Figure 4.16: Architecture of the MicroBlaze NVMe driver

4.7.1 MicroBlaze NVMe Driver Implementation

While the bare-metal MicroBlaze NVMe driver is based on the Embedded UNVMe Linux driver (which in turn is based on the original UNVMe driver for Intel x86 systems), there are several major differences between the two, largely stemming from the lack of operating system, but also due to some architecture differences. In general, removing the operating system simplifies the layers required for an application to access storage, something that is quite desirable for a real-time embedded system, but this also puts additional requirements into the driver itself, and causes some changes to its structure. An overview of the software architecture of the MicroBlaze NVMe driver is shown in Figure 4.16, which highlights both its similarities and differences to Embedded UNVMe.

The major modifications compared to Embedded UNVMe are detailed below, the culmination of which is a stand-alone executable (or library) that can fully configure and interact with an NVMe SSD over PCIe, relying only on the Xilinx-provided C standard library, and with a predictable, static memory footprint.

PCIe Device Configuration

The Linux Embedded UNVMe library relies on existing PCIe support from the kernel, which removes the need for it to manually set up the PCIe bridge or storage device hardware at a low level. It therefore only interacts with the PCIe hardware through the VFIO driver, which provides a relatively high-level interface into the kernel's view of PCIe devices. On the MicroBlaze, however, no existing driver framework is available for accessing the PCIe peripheral of the Zynq UltraScale+ MPSoC, or for configuring an NVMe device attached to it, so a basic

custom driver was written for this purpose. Firstly, this driver must set up the SoC's built-in PCIe peripheral hardware as a root port bridge, in order to enable a downstream NVMe storage device to access the bus. Appropriate memory apertures must also be configured, for allowing software to communicate with the NVMe device using a static area of shared memory attached to an AXI bus. Once the PCIe bridge is set up, the NVMe device is configured using the PCIe Enhanced Configuration Access Mechanism (ECAM), which also allows further changes to the configuration of the device by the NVMe driver as required at runtime, using simple memory-mapped registers.

Memory Access and Allocation

As there is no operating system running to manage memory access and allocation on the MicroBlaze, this must be entirely handled by the driver itself (beyond what is available in the C standard library). Memory access is generally simpler on the MicroBlaze than the Arm CPU, as it avoids complicated cache mechanisms and MMU translations (although a basic MMU is optionally available for use with a MicroBlaze if needed for memory protection or translation). All memory access and operations that were previously managed by the Linux kernel for the Embedded UNVMe driver are reimplemented in the driver itself, including allocating areas of shared memory for command structures, queues and transfer buffers, and managing DMA access between areas of the system.

Beyond the basic memory system changes allowing MicroBlaze code to access the SSD, further changes were made to the driver in order to remove any areas using dynamic memory allocation. While this limits the flexibility of the driver slightly, it also gives strict bounds, configurable at compile time, on the amount of memory that can be used, removing unpredictability in both overall memory usage and execution time variability caused by runtime memory allocations. This lack of runtime flexibility is also unlikely to be an issue in an embedded context, where strict allocation of memory between subsystems at design-time is often desirable rather than limiting.

A further advantage to using static areas of memory with a low-level microcontroller implementation is the ability to use areas outside of main system memory for buffering data and interacting with the storage device, such as scratchpad memories or physically separate DDR interfaces, further removing potential interference with the rest of the system when performing storage operations.

4.7.2 Evaluating NVMe on MicroBlaze

In order to evaluate the NVMe MicroBlaze driver, an appropriate hardware design was created for the Zynq UltraScale+ MPSoC sys-

Table 4.4: Sequential transfer speeds for Arm Embedded UNVMe driver and MicroBlaze NVMe driver on Zynq UltraScale+

Storage interface	Read (MiB/s)	Write (MiB/s)
Embedded UNVMe (Arm)	**1444**	1004
MicroBlaze NVMe	1442	**1007**

tem [42], once again running on a ZCU102 development board [44], and tests and supporting code reimplemented to fit the bare-metal, single executable code model. Further details of the platform set-up can be found in Appendix B.3.

The same basic full-disk sequential read and write benchmarks were run as with Embedded UNVMe, but using the MicroBlaze NVMe driver. These benchmarks were chosen to give a good indication of driver performance, relative to running the closest equivalent tests in Linux.

Performance results are shown in Table 4.4, with the MicroBlaze driver performing almost identically to Embedded UNVMe running on the Arm cores. This result shows essentially equivalent performance between the two drivers, despite the MicroBlaze being a fairly limited soft core processor running at 100 MHz, while Embedded UNVMe is running on a comparatively capable quad-core CPU running at 1.2 GHz.

While some overheads will be caused by Linux in the UNVMe benchmarks, these are clearly negligible, or at least comparable to overheads from the slower processing speed of the MicroBlaze, and both storage access methods appear to fully utilise the available storage device and PCIe bus. The major difference in terms of overall system performance is that the MicroBlaze driver is having far less of a performance impact on the Linux system, using no Linux-aware CPU resources, and only potentially affecting available memory bandwidth on the DDR interface, whereas UNVMe uses up to a full CPU core during transfers.

Given the similarities between the performance of a full-disk write on Embedded UNVMe and the MicroBlaze NVMe driver, it can be assumed that similar performance would be experienced across the full set of more detailed FIO benchmarks run in Linux. Additionally, the idea of a 'block size' is less relevant to the lower-level MicroBlaze driver as it is in Linux, at least beyond the 128 KiB limit of a single NVMe command, as all memory is static and preallocated, and there is only a single thread of execution.

4.7.3 Coprocessor Trade-offs

The use of a coprocessor for accessing storage in an embedded system (and more generally, the use of any coprocessor) has advantages and disadvantages compared to other methods of achieving the same task. In many aspects, a programmable coprocessor sits between a fully-software solution and a dedicated hardware solution in terms of both positives and negatives.

There are two main advantages of using a coprocessor compared to a standard software solution to this problem: firstly, it reduces pressure on the main CPU caused by storage operations that can now be offloaded; and secondly, it allows the creation of a higher-level interface for storage access, such as that described in Section 5.2 potentially simplifying the main CPU software. Together, these points can allow for performance improvements across the system, although potentially not with the level of efficiency of a dedicated hardware component or modifications to the storage device itself.

A coprocessor implemented as described in this chapter is far easier to program and maintain than a dedicated piece of hardware for achieving the same task, as writing and modifying code running on the MicroBlaze is far more flexible than creating Register-Transfer Level (RTL) hardware components, even on an FPGA. For example, while the FPGA logic of the Zynq SoC can be reprogrammed dynamically while Linux is running, the hardware design must be created and synthesized on a relatively powerful external machine, which is a complicated and time-consuming process, whereas MicroBlaze software can be recompiled and programmed directly on the Zynq itself. However, cross-compiling code for a separate microcontroller is still often more effort than creating software that runs as an application, library or kernel driver on the main CPU, which would be the standard method for interfacing with a storage device.

In terms of overall system architecture, a coprocessor and a dedicated hardware component have similar requirements, with both needing auxiliary hardware that is tightly-coupled with the main CPU. For example, both options could feasibly be implemented using an FPGA-based SoC with shared memory available between programmable logic and a main CPU. Both options add increased complexity into a system from hardware and architectural perspectives, however the complexity of software and operating system drivers running on the main CPU may be reduced as a result of offloading tasks.

4.7.4 Further Storage Coprocessor Possibilities

While the MicroBlaze storage driver offers a good trade-off between the flexibility of software and embedded performance through the use of the FPGA fabric, a further direction for the NVMe driver could be

to offload functionality into a dedicated hardware core. The lack of dynamic memory allocation in the MicroBlaze code makes it a good candidate for translation to hardware through a high-level synthesis language, such as Vivado HLS [55], although this step is by no means trivial. A mixed hardware/software system is also a potential compromise solution, moving some of the more static, timing-critical parts of software into a hardware IP core, while retaining the flexibility of software for other areas.

The creation of a storage coprocessor also raises questions of how it might integrate into the rest of a system, both in terms of software control from Linux applications, and overall architecture design. Ideas related to this are explored as future work in Section 5.2.

4.8 Conclusions and Future Work

Efficient access to persistent storage is an important issue for embedded and real-time systems, in which processing resources are limited, and guarantees about timing predictability are as important as data throughput. The storage architectures of modern operating systems involve many complex interactions, which can perform well in many general-purpose scenarios, but more straightforward methods to access to storage can be preferable in many real-time and embedded contexts, as well as other areas of computing where minimising CPU usage is important. This chapter presents three such approaches, ranging from kernel-level modifications, to a user-space library, and a more hardware-oriented storage architecture. A summary of these three approaches is presented in Table 4.5, giving a broad comparison of features, and suggesting advantages and disadvantages of each.

CharIO, the interface proposed for bypassing a number of the complexities associated with operating system storage management, shows promise in improving the performance and predictability of accessing storage in embedded Linux, while still presenting storage through the traditional VFS. While these improvements come at the cost of some conveniences and features offered by traditional file systems, these features may not be required for many real-time and embedded tasks, where a simpler method of accessing storage can be more appropriate.

The results measured from porting the UNVMe driver to an embedded platform show additional promise in low-overhead, predictable access to high-speed storage in such devices, through removing many of the complexities introduced by the Linux kernel's storage stack. Shifting the processing burden of storage access to a separate coprocessor shows the potential for such an architecture to be useful in an embedded context, while benchmark results demonstrate that a simplified and highly bounded storage driver can perform well at a limited clock speed, even with fast storage hardware.

Table 4.5: Comparison of proposed storage interfaces

	CharIO	Embedded UNVMe	NVMe Coprocessor
Accessible through standard Linux VFS devices	Yes, through `/dev/chardiskX` character device	No, requires user-space library	No, requires custom interface to coprocessor
Supports physically-addressed transfers	Yes, using custom *ioctl* calls	Yes, using generic command interface	Yes, using generic command interface
Supports asynchronous I/O operations	No, synchronous command interface only	Yes, with polling for completion	Yes, with polling for completion
Majority of processing time	In kernel code	In user-space library	In coprocessor firmware
Memory protection	In kernel module, or using MMU/hardware	In library code, or using MMU/hardware	In coprocessor firmware, or using MMU/hardware
Hardware support required	None for standard operation; cache-coherent PCIe interconnect for physical addressing	Linux VFIO support, and shared memory mapped through UIO	Separate coprocessor (e.g. MicroBlaze) alongside main CPU

4.8.1 Future Work

As well as further analysis work, and expanding the functionality and compatibility of the CharIO, Embedded UNVMe and MicroBlaze NVMe drivers to different platforms and devices, one major area for future work is to better integrate the interfaces presented in this chapter into complete applications. While a flat presentation of a storage device is adequate or preferential for some applications, such as database engines, which may prefer the possibility of constructing their own file hierarchies (or equivalent) on top of a simple storage space, widespread adoption of alternative storage access methods is unlikely without a sufficiently accessible interface.

Section 5.2 presents a design that builds on the storage coprocessor concept in this chapter to demonstrate how it could fit into a more complete overhaul of high-speed storage access in an embedded system, while filling the gap between current block-based storage devices and potential future byte-addressable non-volatile memory technologies. A further area for future work is the implementation of this proposed hardware storage architecture design, drawing on the implementation work detailed in this chapter to assist with this.

5

Conclusions and Future Work

This chapter provides a summary of the work contained in this thesis, outlining specific contributions and addressing the thesis hypothesis, while evaluating the thesis as a whole. This is followed by a discussion of future work leading from the ideas in this thesis, and an overall conclusion.

5.1 Summary and Contributions

This thesis explores the issues related to accessing high-speed storage in a real-time embedded system, providing measurement and analysis of current storage methods in this context, and proposing alternatives in order to provide improvements in the area.

5.1.1 Summary of Contributions

The following main contributions are offered in each chapter of the thesis.

Chapter 1: Introduction

As well as introducing and motivating the research in this thesis, Chapter 1 presents historical trends in multiple related areas of computer systems, including CPU and memory speeds, storage interface bandwidths, and programmable hardware capabilities, in order to contextualise current discrepancies between high-speed storage and processing improvements.

Chapter 2: Background and Related Work

Chapter 2 provides a comprehensive overview of literature related to storage in real-time embedded systems, highlighting issues related to the area, outlining work that has been carried out to address these issues, and identifying limitations of existing approaches to improvements. The chapter also provides a general background in areas

covered by the thesis, including the evolution of persistent storage, operating systems, and hardware acceleration. A high-level model of storage access is defined, categorising potential operations provided by a typical storage stack into more general domains, along with a series of metrics for consistent measurement of storage systems. Ideas and analysis from background work and related literature inform and motivate the remainder of the thesis.

Chapter 3: Problem Analysis

Chapter 3 provides a large amount of analysis on storage access in Linux, specifically in how it relates to the predictability and efficiency required for real-time and embedded systems. Initial profiling work leads to a thorough benchmarking of standard and direct I/O access to a number of storage devices in an embedded platform, examining both the speed and CPU usage to give an indication of efficiency. This is followed by the presentation of a custom hardware component for low-overhead timing of software and hardware triggers, which is used to measure the variability of periodic storage access, as well as accurate low-level profiling of storage access to expand on these results. The results from this chapter serve to both inform on the differences between current storage access methods in Linux, and to further motivate the need for alternatives in order to improve results based on the desired metrics.

Chapter 4: Alternative Storage Interfaces

Chapter 4 presents a number of alternative storage interfaces for Linux, largely focussing on predictability and efficiency over unnecessary convenience or complex features. CharIO, a kernel driver presenting SSD storage as as character device, is presented and evaluated, showing promise for simplifying the kernel storage stack. This is followed by removing kernel interference to a greater level, through a port of the UNVMe driver to the Zynq UltraScale+ embedded platform, and a complete NVMe driver implementation running on a MicroBlaze soft processor core separate to the main CPU.

Chapter 5: Future Work

As well as concluding the thesis, Chapter 5 discusses multiple directions for future work, including a design for a more ambitious restructuring of the storage stack. This proposed hardware storage architecture presents the idea of creating a simple memory-mapped interface for software to access storage, using custom hardware components to manage this mapping.

5.1.2 Thesis Summary

In summary, the thesis addresses the research hypothesis proposed in Section 1.4 in several ways, confirming that a reduction in the complexity of the software storage stack can indeed result in more predictable and efficient access to storage. Chapters 3 and 4 show through detailed analysis and measurement of both existing and novel storage interfaces that this is the case.

5.2 Hardware Storage Architecture Design

One avenue of future work following from the ideas presented in Chapter 4 is a fundamental change to how storage fits into a system at a logical hardware level, and how it is accessed by applications and the operating system. This change could help to remove the bottlenecks and unpredictability inherent in current persistent storage models by replacing the file system model of accessing a block device with a memory-mapped approach. While vaguely similar in concept to how data can be accessed as memory-mapped files in Linux currently, more fundamental architecture changes could push this abstraction to the CPU's physical view of memory, rather than simply mapping a kernel buffer into an application's process space. An intermediate hardware level would be required to provide translation between memory-mapped files and storage itself, allowing for transparent access to a process's data when paired with sufficient operating system support. This section outlines a design for such a storage interface, and discusses its potential merits and shortfalls.

Techniques such as this are likely to become increasingly of interest, as storage technologies are becoming addressable at a finer level and accessible with lower latencies, and as more system architectures are able to address large memory spaces with high flexibility. For example, the proposed system is similar to the 'shared address space' model presented in [143], however with SSD block storage instead of byte-addressable NVRAM. More integrated solutions for addressing non-volatile memory on the system's main memory bus are emerging, such as Non-Volatile Dual In-line Memory Modules (NVDIMMs) [144, 145], which allows for a more direct replacement of main memory with non-volatile storage. Most current NVRAM implementations still achieve a byte-addressable granularity through a combination of NAND flash storage and volatile byte-addressable memory, however [144].

5.2.1 Mapping Storage into Memory

Currently, most system architectures make a clear distinction between main volatile system memory and non-volatile secondary storage devices, modelling the latter as separate devices that must be accessed

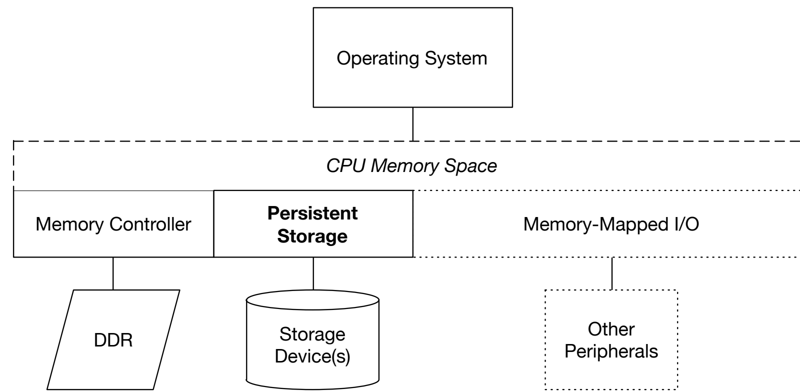


Figure 5.1: High-level view of persistent storage as a first-class member of a system's memory map

at a block level by copying data to and from main memory as it is used. An alternative to this could be to directly map files from a storage device as areas in the standard virtual memory map of a process, removing the need to use explicit read and write system calls to access data.

While the current concept of memory-mapped files in Linux (using the `mmap` system call) relies on the kernel page cache and standard block layer below this, the proposed system would bypass the kernel block layer and storage device drivers entirely, creating a simpler file-to-memory mapping at a lower level in the memory hierarchy, as shown in Figure 5.1.

When considering mapping storage directly into the memory space of a system, the size of the addressable region must be taken into account. For a 64-bit architecture, the potential memory space is so large that a storage device could feasibly be mapped entirely into a memory region, and addressed as such. While many embedded systems operate with a 32-bit address space, limiting the maximum size of a memory region to 4 GiB, or a virtual implementation of a 64-bit address space that is physically limited to a smaller number of address lines (such as 40-bit), a general trend amongst systems is for address spaces to be growing (for example, compare the Xilinx Zynq-7000 [41] with the newer Zynq UltraScale+ [42] SoCs). However, as an address space has a potential to be far smaller than the requirements for persistent storage memory, this must be considered if implementing this type of mapping, potentially using a paging scheme to bring storage into scope only as it is used, and to continually remap the addressable region as necessary. Any additional memory mapped peripherals in a system must also be taken into account, further limiting the potential contiguous free space in a small address map.

To create an effective mapping from a plain CPU memory region into addressable storage, a Storage Memory Management Unit (SMMU) is proposed. Such a peripheral could be added as an additional layer in

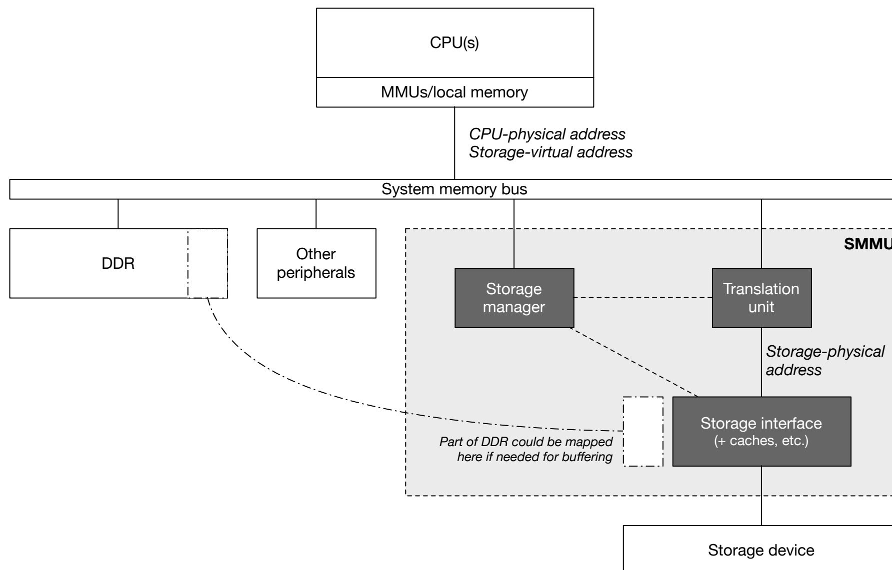


Figure 5.2: Proposed hardware storage architecture design

the memory hierarchy, translating CPU-physical but storage-virtual addresses into storage-physical addresses, which refer to specific blocks of data on a storage device.

5.2.2 Hardware Support

The proposed architecture changes require a higher level of hardware support than is currently necessary for storage in a typical system, in order to translate memory requests to storage devices accesses, and potentially to accelerate further storage operations.

On top of the translation functions of the SMMU, file system and storage management operations can optionally be placed in either hardware or software, creating a mapping from some concept of 'files' to blocks or areas on the storage device. While a hardware implementation could offer performance benefits and allow for more optimisations to be performed at a lower level, this type of functionality is challenging to create and less flexible than a software alternative. The use of embedded soft processor cores can offer a reasonable compromise in this area, as demonstrated in Section 4.7.

The memory available to hardware components for buffering data as storage requests are serviced is another consideration for such a system. As block storage devices are inherently addressed by block number rather than absolute address, there must always be a buffer available to hold a block if it is to be accessed in a byte-addressable manner. Many development platforms have additional memory that could be used to this effect, such as secondary DDR RAM, SRAM scratchpad memories, or more-limited block RAMs within FPGA logic.

To achieve the above, the architecture shown in Figure 5.2 is proposed, with the SMMU comprising a storage manager for overall control, a translation unit to provide address translation functionality, and a storage interface, including buffers and caches as required by any block-level storage device below. As the SMMU is at a lower level in the memory hierarchy than the main system MMU, addresses coming from the CPU are in its physical domain, but are still virtual in terms of storage. These are then translated to storage-physical addresses by the translation unit, based on the current system context as decided by the storage manager, before being passed to the storage interface for requests to be serviced.

5.2.3 Operating System Support

In order for files to be addressable through a memory interface, a certain level of operating system support is required. The extent of changes needed at the operating system level involves, at a minimum, modifications being made to storage and memory management in order for devices to be accessed effectively. In order for storage to be accessible securely in a multi-process environment, the operating system must inform the storage manager of the current system state, so translation tables for a specific process and files can be loaded if necessary.

5.2.4 Hardware Device Acceleration

With appropriate low-level hardware support, the memory-mapping of storage devices may go further than being limited to the operating system's perspective, allowing other system devices to access storage via the memory-mapped interface through DMA. This could potentially allow for devices such as network controllers to send packets directly into persistent storage, even adding intermediate hardware-accelerators for processing data, while being controlled by an operating system treating both volatile and non-volatile memory as a single resource.

5.2.5 Hardware Architecture Implementation

The major challenge of this area of future work is an implementation of the proposed storage architecture design, allowing for it to be thoroughly evaluated and further refined through experimentation. Such an implementation could feasibly be performed on the same platform as the storage coprocessor experiments in Chapter 4, and also reuse the MicroBlaze NVMe driver as a method to access storage hardware.

5.3 Other Future Work

There are a number of areas of future work identified by the thesis, some raised from limitations of the work presented, and some as a progression of ideas or addressing questions raised by the research.

In general, the work presented in this thesis, particularly the analysis work in Chapter 3, motivates a large amount of future investigation into the simplification of storage systems. The results from detailed benchmarking and profiling, along with the examination of background literature and general storage trends, show that established methods of accessing storage devices through operating systems are by no means optimal, especially when considering systems that require high levels of predictability, accountability, and efficiency.

5.3.1 Further Problem Analysis Work

While the problem analysis work presented in Chapter 3 is comprehensive in some respects, more work could be carried out in many areas to extend and validate its conclusions. For example, the work focuses on a single embedded platform and (for some experiments) a single server platform, which gives only a basic indication of how the results might apply in each of these domains. While the platforms were chosen to be representative of a typical system, and are detailed comprehensively in Appendix B, testing on a larger variety of platforms, and with a larger number of different storage devices, would give a much more comprehensive view from which to evaluate and draw conclusions.

Additionally, the results presented in Chapter 3 only apply to the Linux operating system, and more specifically to the exact versions in use during the experiments. While there are many common ideas shared between most mainstream operating system, and the progression of major features between subsequent releases of the Linux kernel is often quite slow and measured, full conclusions about the state of storage in Linux, or in operating systems in general, cannot be constructed without a much larger and longer-term analysis effort. However, the results presented are still completely valid, with this caveat, and are definitely strong enough to provide motivation for further research and (hopefully) eventual practical changes to be made.

Another limitation of the analysis work presented is the lack of 'real-world' benchmarking, however this definition can cover such a large number of use cases and patterns that contrived but representative benchmarks, and the presentation of higher-level concepts, can be much more useful as a foundation for research. It is therefore left as future work for specific real-world tests to be found and validated,

building upon the analysis work in this thesis, and likely experiencing similar conclusions.

5.3.2 Storage Interface Improvements

For the alternative storage interfaces presented in Chapter 4, further analysis and evaluation work similar to that mentioned for Chapter 3 would give greater weight to their proposal, as well as directing areas where potential improvements could be made. As well as this, implementations of the drivers themselves could be greatly expanded and optimised in multiple areas, bringing them from proof-of-concept research work to something that could be used in a wider context. For example, the current CharIO driver is heavily tied to NVMe storage, but a more general concept could be applied to other storage hardware. Further work on expanding the basic user-space library would also allow for the potential of more widespread adoption.

5.3.3 Future Platforms and Technologies

Another cause of future work, and one that is largely out of the control of this thesis, is the ever-progressing landscape of technologies related to storage, embedded system, and computers in general. While experimental work presented uses a number of current technologies, such as NVMe SSDs, modern Arm SoCs, and large FPGAs, these are likely to change fairly rapidly over the coming years. For example, the Zynq UltraScale+ platform ultimately used in Chapter 4 was first announced in February 2014 [146], around four months after the start of this EngD, with actual practical hardware and development boards not available until nearly three years after this date. Similarly, there have been nearly 400,000 individual commits to the mainline Linux kernel in the five years from September 2013 (`v3.12-rc3`) to September 2018 (`v4.19-rc4`), demonstrating the speed at which operating system development progresses, and potentially having significant effects on the results of experiments. In such a rapidly progressing area, technology development is continually outpacing research, to the extent that future work can always be performed using more up-to-date hardware and software.

One area that has the greatest potential to disrupt storage architectures in coming years is the availability of cheap, fast, byte-addressable non-volatile RAM, which could ultimately replace block-based storage devices entirely. The current state of the art in this area is pushing towards NVDIMM devices connected to the main memory bus of general-purpose computers [144], allowing both for traditional uses of RAM to become non-volatile, and for finer-grained access to data storage. Commercial offerings typically currently mask more traditional flash storage behind large volatile memory caches, but advancements

in non-volatile memory technologies mean these could be replaced by fully NVRAM devices soon [145]. While these storage technologies may cause some of the analysis in this thesis to become obsolete, the general ideas motivating the need to restructure the storage architectures of operating systems will be stronger than ever.

5.4 Conclusion

This thesis has demonstrated a number of issues in general-purpose storage architectures when applied to real-time embedded systems, particularity in the efficiency and predictability of accessing high-speed storage devices in an embedded Linux system. Detailed problem analysis work motivates research in this area, highlighting a number of problems with current implementations, while making suggestions for how best to deal with these. Based on this analysis, a number of alternative storage access methods are proposed for real-time embedded systems. These include both user-space and hardware-based storage drivers, and CharIO, a Linux kernel module that exposes storage as a character device, avoiding the complexities of the block layer and traditional file systems, and allowing more powerful access to memory through physical addressing.

Appendices

A

Historical Trends

This appendix contains additional data to compliment that presented in Section 1.3.

A.1 General CPU Trends

Data for the following plots was sourced from [24].

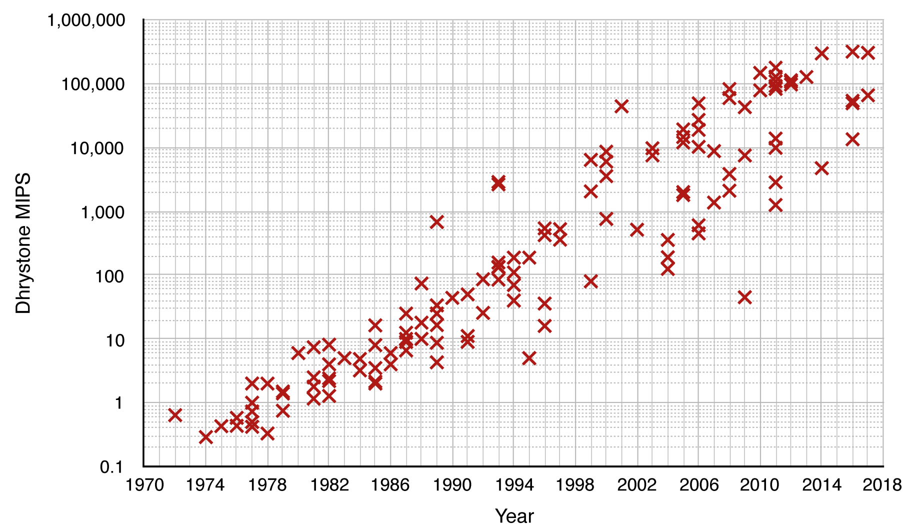


Figure A.1: CPU Dhrystone MIPS over time

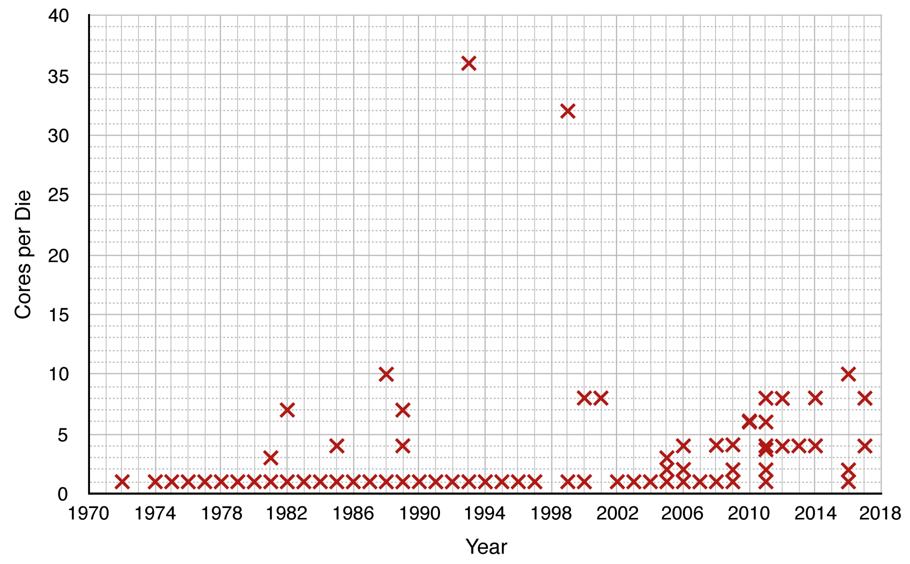


Figure A.2: CPU core counts over time

A.2 Storage Interface Trends

Data for the following plot was sourced from [29–34].

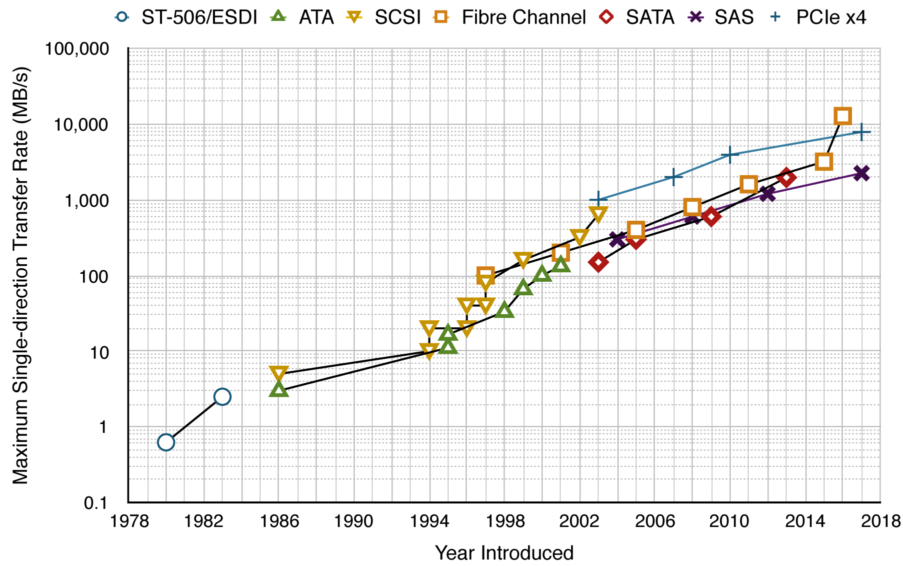


Figure A.3: Storage interface speed progression over time

A.3 FPGA Trends

Data for the following plots was sourced from [147–154].

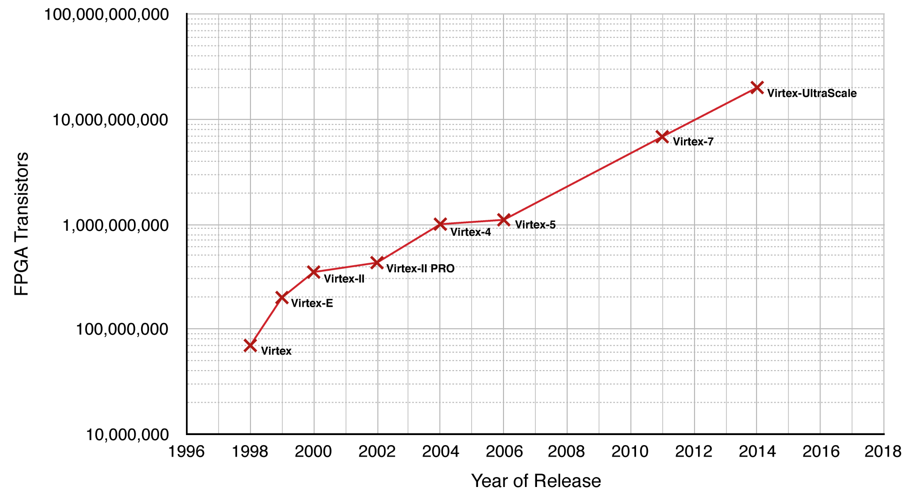


Figure A.4: FPGA transistor count progression over time

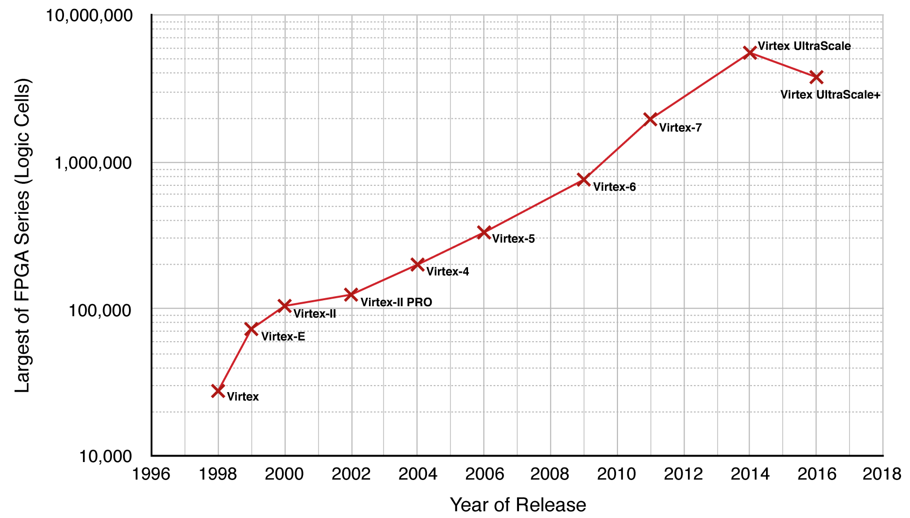


Figure A.5: FPGA programmable logic size progression over time

A.4 Hennessy-Patterson CPU Trends

Data for the following plot was sourced from [23].

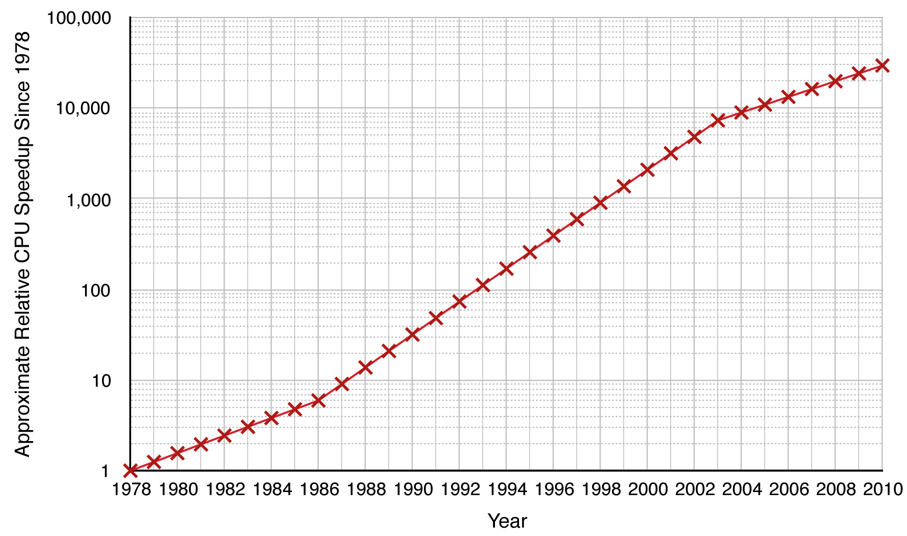


Figure A.6: Relative CPU performance increase since 1978

B

Experimental Platforms

B.1 x86-64 Server Platform

For a number of experiments in Chapters 3 and 4, an x86-64 server platform was used for initial investigations, and to gain an impression of performance on a standard Linux computer, as opposed to an embedded platform.

B.1.1 Technical specifications

CPU Intel Core 2 Quad Q9450 [155]

- 4 cores at 2.66 GHz
- 32 KiB L1 instruction and data caches (per core)
- 12 MiB L2 cache (2×6 MiB shared)

Memory 4 GiB DDR3 SDRAM

PCI Express Gen. 2 (5 GT/s), x16

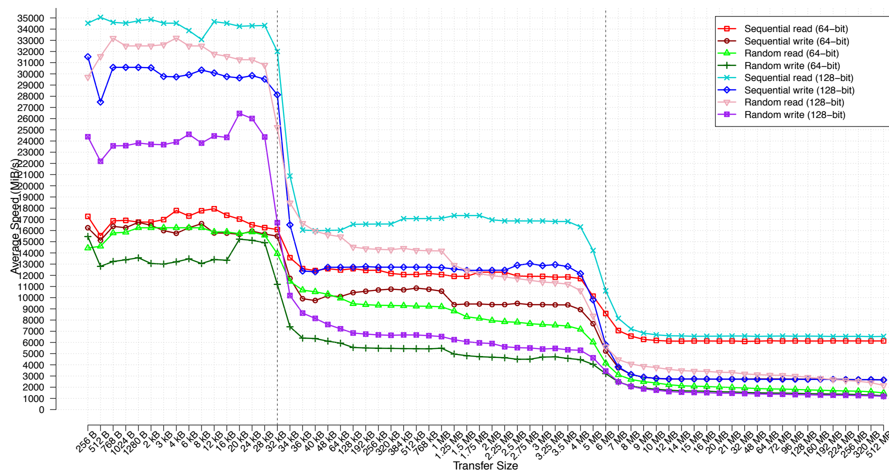


Figure B.1: Memory performance of x86-64 server

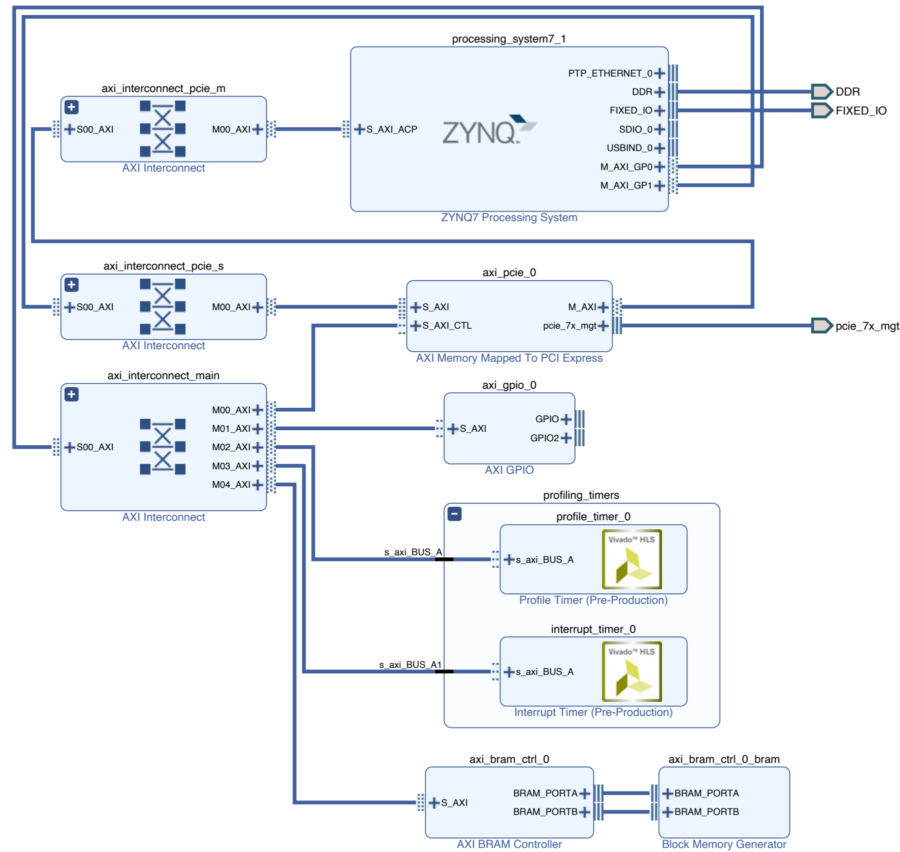


Figure B.2: System architecture of the Zynq-7000 SoC test set-up

B.1.2 Server Platform Memory Performance

Figure B.1 shows the memory performance of the server platform for different transfer sizes. The effect of the L1 and L2 caches can clearly be seen at 32 KiB and 6 MiB.

B.2 Zynq-7000 SoC Platform

For the majority of technical and experimental work carried out in Chapters 3 and 4, an Avnet Mini-ITX development board [43] was used, with a Xilinx Zynq-7000 SoC [41]. Figure B.2 shows the overall system connectivity from the Xilinx Vivado Design Suite, with the Zynq processing system, PCIe controller, profiling timers, and BRAMs.

B.2.1 Technical specifications

SoC Xilinx XC7Z100-2FFG900
CPU Arm Cortex-A9

- 2 cores at up to 800 MHz
- 32 KiB L1 instruction and data caches (per core)

  - 512 KiB L2 cache (shared)

PS Memory 1 GiB DDR3 SDRAM
PL Memory 1 GiB DDR3 SDRAM
PCI Express Gen. 2 (5 GT/s), x4 (through PL)

B.2.2 Zynq-7000 Platform Memory Performance

The CPU side (processing system) of the Zynq-7000, along with the DDR3 SDRAM present on the Mini-ITX board, has the following memory characteristics.

L1 cache (per-core)

- 4-way set-associative
- 32 byte cache lines
- 64-bit interfaces
- Cache replacement policy: pseudo round-robin or pseudo-random
- 32 KiB Data
  - Write-back/write-allocate only
  - Physically indexed and physically tagged
  - Non-blocking (four outstanding reads and four outstanding writes)
- 32 KiB Instruction
  - Virtually indexed and physically tagged

L2 cache (shared)

- Based on Arm PL310
- 512 KiB
- 8-way set-associative
- 32 byte cache lines
- Physically addressed and physically tagged
- Supports write-through and write-back
- Supports read allocate, write allocate, or read and write allocate.

Figure B.3 shows the memory performance of the Zynq-7000 platform for different transfer sizes. The effect of the L1 and L2 caches can clearly be seen at 32 KiB and 512 KiB.

For reads, the L1 and L2 caches appear to have a significant effect on read speeds. Random accesses are far slower than sequential in L1, but random is faster than sequential in L2 and DDR. For writes, the L1 cache does not appear to have much effect on sequential writes, but L2 does. Sequential writes are significantly faster than reads after transfer sizes exceed the L1 cache, and continue to outperform random writes when L2 size is also exceeded.

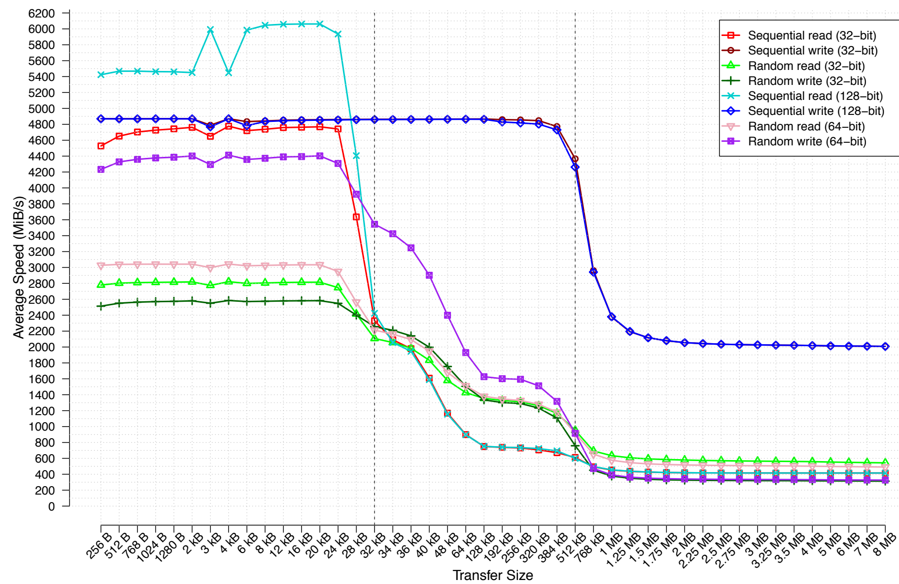


Figure B.3: Memory performance of Zynq Mini-ITX development board

B.3 Zynq UltraScale+ MPSoC Platform

For the remainder of technical and experimental work carried out in Chapter 4, including work on porting UNVMe to embedded systems, NVMe access from a MicroBlaze soft processor core, and a platform upon which to use as a base for a potential hardware storage architecture design, a Xilinx ZCU102 evaluation kit [44] was used, with a Xilinx Zynq UltraScale+ MPSoC [42]. Figure B.4 shows the overall system connectivity from the Xilinx Vivado Design Suite, with the Zynq processing system, MicroBlaze soft-core processor, and secondary DDR memory interface.

B.3.1 Technical specifications

SoC Xilinx XCZU9EG-2FFVB1156
APU Arm Cortex-A53

- 4 cores at up to 1.2 GHz
- 32 KiB L1 instruction and data caches (per core)
- 1 MiB L2 cache (shared)

RTPU Arm Cortex-R5

- 2 cores at up to 600 MHz
- 32 KiB L1 instruction and data caches (per core)

PS Memory 4 GiB DDR4 SDRAM
PL Memory 512 MiB DDR4 SDRAM
PCI Express Gen. 2 (5 GT/s), x4 (from PS)

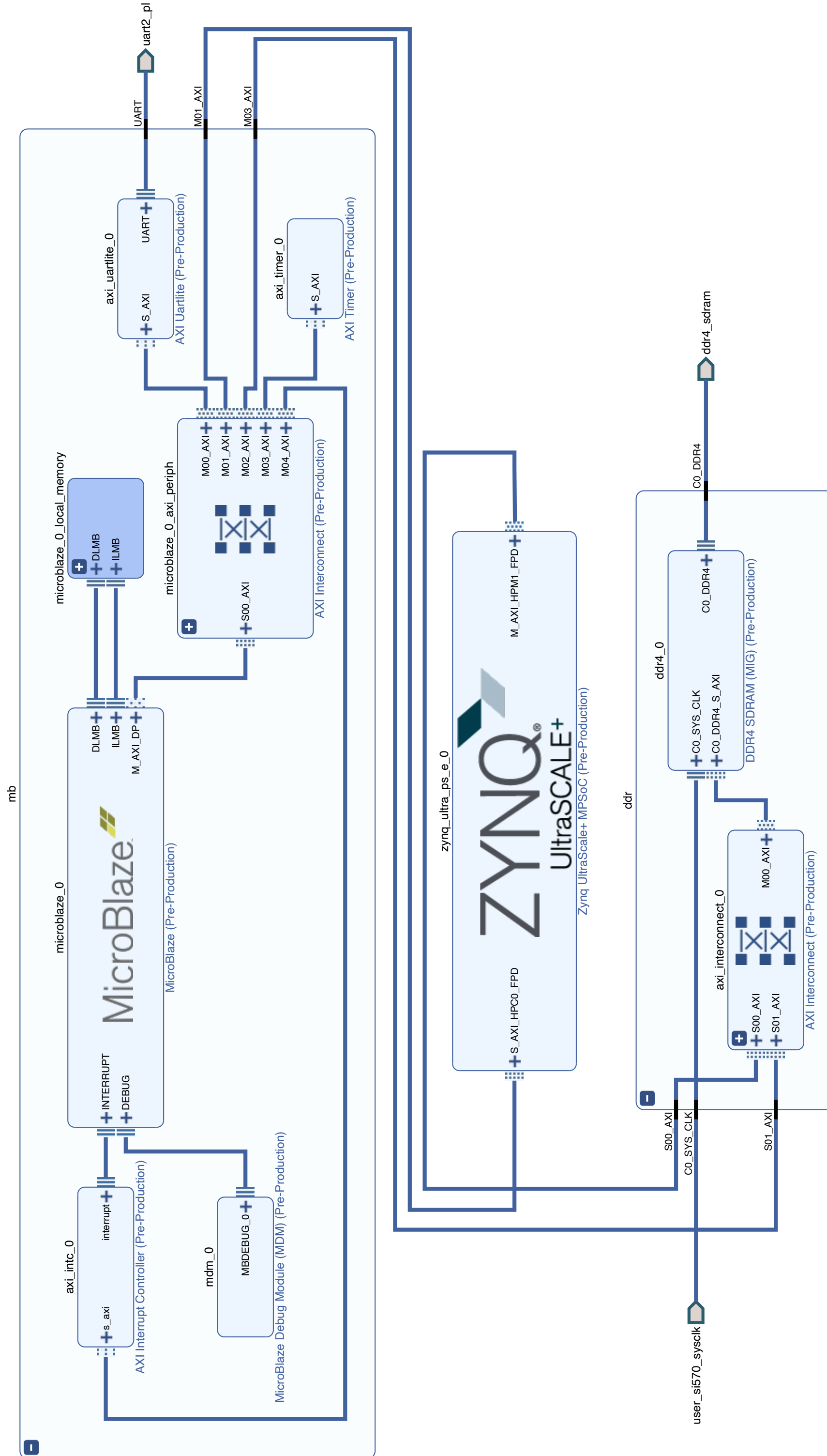


Figure B.4: System architecture of the Zynq UltraScale+ MPSoC test set-up

B.4 Additional Hardware

B.4.1 Storage Devices and Controllers

The following storage devices and controllers were used for the experiments detailed in Chapters 3 and 4.

- StarTech 4-Port RAID Controller (PCIe Gen. 2 x2 to SATA III) [125]
- Western Digital Blue HDD 500GB (SATA III) [123]
- Intel SSD 535 Series 240GB (SATA III) [124]
- Intel SSD 750 Series 400GB (NVMe over PCIe Gen. 3 x4) [16]

C

Source Code Listings

C.1 File Copy for Profiling

copy.c

```
#include <fcntl.h>

int main(int argc, char *argv[]) {
    int inf;
    int outf;
    int len;
    char buf[4096];

    if (argc < 3)
        return 1;

    inf = open(argv[1], O_RDONLY, 0);
    outf = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0600);
    do {
        len = read(inf, buf, 4096);
        write(outf, buf, len);
    } while (len);

    close(outf);
    close(inf);

    return 0;
}
```

C.2 UNVMe Full-disk Read/Write Benchmark

unvme-benchmark.c

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "unvme.h"

void read_benchmark() {
    printf("\r\nRead benchmark running\r\n");

    void *buf;
    const unvme_ns_t *ns;
    struct timespec start, end;
    double time, bps;
    u64 blocks = 0x5d27215;
    u64 blocks_per_io;
    u64 size, size_per_io;

    ns = unvme_openq("01:00.0", 1, 1024);

    blocks_per_io = ns->maxbpio * ns->maxiopq;
    size = ns->blocksize * blocks;
    size_per_io = ns->blocksize * blocks_per_io;

    buf = unvme_alloc(ns, size_per_io);

    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    for (int i = 0; i < blocks; i += blocks_per_io) {
        if (i + blocks_per_io <= blocks)
            unvme_read(ns, 0, buf, i, blocks_per_io);
        else
            unvme_read(ns, 0, buf, i, blocks - i);
    }
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);

    time = (double)((end.tv_sec * 1000000000LL + end.tv_nsec) -
           (start.tv_sec * 1000000000LL + start.tv_nsec)) /
           1000000000.0;
    bps = size / time / (1024.0 * 1024.0);

    printf("Reading %llu bytes took %.3lf seconds - "
           "%.3lf MiB/s\r\n", size, time, bps);

    unvme_free(ns, buf);
    unvme_close(ns);

    printf("\r\nRead benchmark done.\r\n");
}
```

```
void write_benchmark() {
    printf("\r\nWrite benchmark running\r\n");

    void *buf;
    const unvme_ns_t *ns;
    struct timespec start, end;
    double time, bps;
    u64 blocks = 0x5d27215;
    u64 blocks_per_io;
    u64 size, size_per_io;

    ns = unvme_openq("01:00.0", 1, 1024);

    blocks_per_io = ns->maxbpio * ns->maxiopq;
    size = ns->blocksize * blocks;
    size_per_io = ns->blocksize * blocks_per_io;

    buf = unvme_alloc(ns, size_per_io);

    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
    for (int i = 0; i < blocks; i += blocks_per_io) {
        if (i + blocks_per_io <= blocks)
            unvme_write(ns, 0, buf, i, blocks_per_io);
        else
            unvme_write(ns, 0, buf, i, blocks - i);
    }
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);

    time = (double)((end.tv_sec * 1000000000LL + end.tv_nsec) -
           (start.tv_sec * 1000000000LL + start.tv_nsec)) /
           1000000000.0;
    bps = size / time / (1024.0 * 1024.0);

    printf("Writing %llu bytes took %.3lf seconds - "
           "%.3lf MiB/s\r\n", size, time, bps);

    unvme_free(ns, buf);
    unvme_close(ns);

    printf("\r\nWrite benchmark done.\r\n");
}

int main() {
    read_benchmark();
    write_benchmark();
    return 0;
}
```

C.3 Filebench Benchmark Workloads

Information about the Filebench Workload Model Language (WML) can be found at [156]. The original predefined workload sources (including licence information), on which the following are based, can be found online with the Filebench source code at [10].

C.3.1 Web server (*webserver.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=600000
set $meandirwidth=20
set $filesize=cvar(type=cvar-gamma-aligned,parameters=mean:16384;gamma
    :1.5;align:4096)
set $nthreads=50
set $iosize=1m
set $meanappendsize=16k

define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=100,readonly
define fileset name=logfiles,path=$dir,size=$filesize,entries=1,dirwidth=
    $meandirwidth,prealloc

define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop openfile name=openfile1,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile1,fd=1,iosize=$iosize
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile2,fd=1,iosize=$iosize
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize
    flowop closefile name=closefile4,fd=1
    flowop openfile name=openfile5,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile5,fd=1,iosize=$iosize
    flowop closefile name=closefile5,fd=1
    flowop openfile name=openfile6,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile6,fd=1,iosize=$iosize
    flowop closefile name=closefile6,fd=1
    flowop openfile name=openfile7,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile7,fd=1,iosize=$iosize
    flowop closefile name=closefile7,fd=1
    flowop openfile name=openfile8,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile8,fd=1,iosize=$iosize
    flowop closefile name=closefile8,fd=1
    flowop openfile name=openfile9,filesetname=bigfileset,fd=1
```

```
    flowop readwholefile name=readfile9,fd=1,iosize=$iosize
    flowop closefile name=closefile9,fd=1
    flowop openfile name=openfile10,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile10,fd=1,iosize=$iosize
    flowop closefile name=closefile10,fd=1
    flowop appendfilerand name=appendlog,filesetname=logfiles,iosize=
        $meanappendsize,fd=2
  }
}

echo  "Web-server Version 3.1 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.2 Web server direct I/O (*webserver-direct.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=600000
set $meandirwidth=20
set $filesize=cvar(type=cvar-gamma,parameters=mean:16384;gamma:1.5)
set $nthreads=50
set $iosize=1m
set $meanappendsize=16k

define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=100,readonly
define fileset name=logfiles,path=$dir,size=$filesize,entries=1,dirwidth=
    $meandirwidth,prealloc

define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop openfile name=openfile1,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile1,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile2,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize,directio
```

```
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile4,fd=1
    flowop openfile name=openfile5,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile5,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile5,fd=1
    flowop openfile name=openfile6,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile6,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile6,fd=1
    flowop openfile name=openfile7,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile7,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile7,fd=1
    flowop openfile name=openfile8,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile8,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile8,fd=1
    flowop openfile name=openfile9,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile9,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile9,fd=1
    flowop openfile name=openfile10,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile10,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile10,fd=1
    flowop appendfilerand name=appendlog,filesetname=logfiles,iosize=
        $meanappendsize,fd=2,directio
  }
}

echo  "Web-server Direct I/O Version 3.1 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.3 File server (*fileserver.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=80000
set $meandirwidth=20
set $filesize=cvar(type=cvar-gamma,parameters=mean:131072;gamma:1.5)
set $nthreads=50
set $iosize=1m
set $meanappendsize=16k
```

```
define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=80

define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop createfile name=createfile1,filesetname=bigfileset,fd=1
    flowop writewholefile name=wrtfile1,srcfd=1,fd=1,iosize=$iosize
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile1,filesetname=bigfileset,fd=1
    flowop appendfilerand name=appendfilerand1,iosize=$meanappendsize,fd
        =1
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile1,fd=1,iosize=$iosize
    flowop closefile name=closefile3,fd=1
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop statfile name=statfile1,filesetname=bigfileset
  }
}

echo  "File-server Version 3.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.4 File server direct I/O (*fileserver-direct.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=80000
set $meandirwidth=20
set $filesize=cvar(type=cvar-gamma-aligned,parameters=mean:131072;gamma
    :1.5;align:4096)
set $nthreads=50
set $iosize=1m
set $meanappendsize=16k

define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=80

define process name=filereader,instances=1
```

```
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop createfile name=createfile1,filesetname=bigfileset,fd=1,
        directio
    flowop writewholefile name=wrtfile1,srcfd=1,fd=1,iosize=$iosize,
        directio
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile1,filesetname=bigfileset,fd=1,directio
    flowop appendfilerand name=appendfilerand1,iosize=$meanappendsize,fd
        =1,directio
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile1,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile3,fd=1
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop statfile name=statfile1,filesetname=bigfileset
  }
}

echo  "File-server Direct I/O Version 3.0 personality successfully loaded
    "

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.5 Mail server (*varmail.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=600000
set $meandirwidth=1000000
set $filesize=cvar(type=cvar-gamma,parameters=mean:16384;gamma:1.5)
set $nthreads=16
set $iosize=1m
set $meanappendsize=16k

define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=80

define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
```

```
  {
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop createfile name=createfile2,filesetname=bigfileset,fd=1
    flowop appendfilerand name=appendfilerand2,iosize=$meanappendsize,fd
        =1
    flowop fsync name=fsyncfile2,fd=1
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize
    flowop appendfilerand name=appendfilerand3,iosize=$meanappendsize,fd
        =1
    flowop fsync name=fsyncfile3,fd=1
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize
    flowop closefile name=closefile4,fd=1
  }
}

echo  "Varmail Version 3.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.6 Mail server direct I/O (*varmail-direct.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=600000
set $meandirwidth=1000000
set $filesize=cvar(type=cvar-gamma-aligned,parameters=mean:16384;gamma
    :1.5;align:4096)
set $nthreads=16
set $iosize=1m
set $meanappendsize=16k

define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=80

define process name=filereader,instances=1
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
```

```
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop createfile name=createfile2,filesetname=bigfileset,fd=1,
        directio
    flowop appendfilerand name=appendfilerand2,iosize=$meanappendsize,fd
        =1,directio
    flowop fsync name=fsyncfile2,fd=1
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize,directio
    flowop appendfilerand name=appendfilerand3,iosize=$meanappendsize,fd
        =1,directio
    flowop fsync name=fsyncfile3,fd=1
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile4,fd=1
  }
}

echo  "Varmail Direct I/O Version 3.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.7 Video server (*videoserver.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $eventrate=96
set $filesize=10g
set $nthreads=48
set $numactivevids=8
set $numpassivevids=28
set $reuseit=false
set $readiosize=256k
set $writeiosize=1m

set $passvidsname=passivevids
set $actvidsname=activevids

set $repintval=10

eventgen rate=$eventrate
```

```
define fileset name=$actvidsname,path=$dir,size=$filesize,entries=
    $numactivevids,dirwidth=4,prealloc,paralloc,reuse=$reuseit
define fileset name=$passvidsname,path=$dir,size=$filesize,entries=
    $numpassivevids,dirwidth=20,prealloc=50,paralloc,reuse=$reuseit

define process name=vidwriter,instances=1
{
  thread name=vidwriter,memsize=10m,instances=1
  {
    flowop deletefile name=vidremover,filesetname=$passvidsname
    flowop createfile name=wrtopen,filesetname=$passvidsname,fd=1
    flowop writewholefile name=newvid,iosize=$writeiosize,fd=1,srcfd=1
    flowop closefile name=wrtclose, fd=1
    flowop delay name=replaceinterval, value=$repintval
  }
}

define process name=vidreaders,instances=1
{
  thread name=vidreaders,memsize=10m,instances=$nthreads
  {
    flowop read name=vidreader,filesetname=$actvidsname,iosize=
        $readiosize
    flowop bwlimit name=serverlimit, target=vidreader
  }
}

echo  "Video Server Version 3.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.8 Video server direct I/O (*videoserver-direct.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $eventrate=96
set $filesize=10g
set $nthreads=48
set $numactivevids=8
set $numpassivevids=28
set $reuseit=false
```

```
set $readiosize=256k
set $writeiosize=1m

set $passvidsname=passivevids
set $actvidsname=activevids

set $repintval=10

eventgen rate=$eventrate

define fileset name=$actvidsname,path=$dir,size=$filesize,entries=
    $numactivevids,dirwidth=4,prealloc,paralloc,reuse=$reuseit
define fileset name=$passvidsname,path=$dir,size=$filesize,entries=
    $numpassivevids,dirwidth=20,prealloc=50,paralloc,reuse=$reuseit

define process name=vidwriter,instances=1
{
  thread name=vidwriter,memsize=10m,instances=1
  {
    flowop deletefile name=vidremover,filesetname=$passvidsname
    flowop createfile name=wrtopen,filesetname=$passvidsname,fd=1,
        directio
    flowop writewholefile name=newvid,iosize=$writeiosize,fd=1,srcfd=1,
        directio
    flowop closefile name=wrtclose, fd=1
    flowop delay name=replaceinterval, value=$repintval
  }
}

define process name=vidreaders,instances=1
{
  thread name=vidreaders,memsize=10m,instances=$nthreads
  {
    flowop read name=vidreader,filesetname=$actvidsname,iosize=
        $readiosize,directio
    flowop bwlimit name=serverlimit, target=vidreader
  }
}

echo  "Video Server Direct I/O Version 3.0 personality successfully
    loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.9 Network file server (*netsfs.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $eventrate=10
set $meandirwidth=20
set $nthreads=1
set $nfiles=400000
set $sync=false

eventgen rate=$eventrate

set $wrtiosize = randvar(type=tabular, min=1k, round=1k, randtable =
{{ 0,   1k,    7k},
 {50,   9k,   15k},
 {14,  17k,   23k},
 {14,  33k,   39k},
 {12,  65k,   71k},
 {10, 129k,  135k}
})

set $rdiosize = randvar(type=tabular, min=8k, round=1k, randtable =
{{85,   8k,   8k},
 { 8,  17k,  23k},
 { 4,  33k,  39k},
 { 2,  65k,  71k},
 { 1, 129k, 135k}
})

set $filesize = randvar(type=tabular, min=1k, round=1k, randtable =
{{33,   1k,    1k},
 {21,   1k,    3k},
 {13,   3k,    5k},
 {10,   5k,   11k},
 {08,  11k,   21k},
 {05,  21k,   43k},
 {04,  43k,   85k},
 {03,  85k,  171k},
 {02, 171k,  341k},
 {01, 341k, 1707k}
})

set $fileidx = randvar(type=gamma, min=0, gamma=100)

define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=60

define flowop name=rmw
{
  flowop openfile name=openfile1,filesetname=bigfileset,indexed=$fileidx,
      fd=1
  flowop readwholefile name=readfile1,iosize=$rdiosize,fd=1
  flowop createfile name=newfile2,filesetname=bigfileset,indexed=$fileidx
      ,fd=2
```

```
  flowop writewholefile name=writefile2,fd=2,iosize=$wrtiosize,srcfd=1
  flowop closefile name=closefile1,fd=1
  flowop closefile name=closefile2,fd=2
  flowop deletefile name=deletefile1,fd=1
}

define flowop name=launch
{
  flowop openfile name=openfile3,filesetname=bigfileset,indexed=$fileidx,
      fd=3
  flowop readwholefile name=readfile3,iosize=$rdiosize,fd=3
  flowop openfile name=openfile4,filesetname=bigfileset,indexed=$fileidx,
      fd=4
  flowop readwholefile name=readfile4,iosize=$rdiosize,fd=4
  flowop closefile name=closefile3,fd=3
  flowop openfile name=openfile5,filesetname=bigfileset,indexed=$fileidx,
      fd=5
  flowop readwholefile name=readfile5,iosize=$rdiosize,fd=5
  flowop closefile name=closefile4,fd=4
  flowop closefile name=closefile5,fd=5
}

define flowop name=appnd
{
  flowop openfile name=openfile6,filesetname=bigfileset,indexed=$fileidx,
      fd=6
  flowop appendfilerand name=appendfilerand6,iosize=$wrtiosize,fd=6
  flowop closefile name=closefile6,fd=6
}

define process name=netclient,instances=1
{
  thread name=fileuser,memsize=10m,instances=$nthreads
  {
    flowop launch name=launch1, iters=1
    flowop rmw name=rmw1, iters=6
    flowop appnd name=appnd1, iters=3
    flowop statfile name=statfile1,filesetname=bigfileset,indexed=
        $fileidx
    flowop eventlimit name=ratecontrol
  }
}

echo  "NetworkFileServer Version 1.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"
```

```
echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.10 Network file server direct I/O (*netsfs-direct.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $eventrate=10
set $meandirwidth=20
set $nthreads=1
set $nfiles=400000
set $sync=false

eventgen rate=$eventrate

set $wrtiosize = randvar(type=tabular, min=4k, round=4k, randtable =
{{ 0,   1k,    7k},
 {50,   9k,   15k},
 {14,  17k,   23k},
 {14,  33k,   39k},
 {12,  65k,   71k},
 {10, 129k,  135k}
})

set $rdiosize = randvar(type=tabular, min=8k, round=4k, randtable =
{{85,   8k,   8k},
 { 8,  17k,  23k},
 { 4,  33k,  39k},
 { 2,  65k,  71k},
 { 1, 129k, 135k}
})

set $filesize = randvar(type=tabular, min=4k, round=4k, randtable =
{{33,   1k,    1k},
 {21,   1k,    3k},
 {13,   3k,    5k},
 {10,   5k,   11k},
 {08,  11k,   21k},
 {05,  21k,   43k},
 {04,  43k,   85k},
 {03,  85k,  171k},
 {02, 171k,  341k},
 {01, 341k, 1707k}
})

set $fileidx = randvar(type=gamma, min=0, gamma=100)

define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=60

define flowop name=rmw
{
  flowop openfile name=openfile1,filesetname=bigfileset,indexed=$fileidx,
      fd=1
  flowop readwholefile name=readfile1,iosize=$rdiosize,fd=1
```

```
  flowop createfile name=newfile2,filesetname=bigfileset,indexed=$fileidx
      ,fd=2
  flowop writewholefile name=writefile2,fd=2,iosize=$wrtiosize,srcfd=1
  flowop closefile name=closefile1,fd=1
  flowop closefile name=closefile2,fd=2
  flowop deletefile name=deletefile1,fd=1
}

define flowop name=launch
{
  flowop openfile name=openfile3,filesetname=bigfileset,indexed=$fileidx,
      fd=3,directio
  flowop readwholefile name=readfile3,iosize=$rdiosize,fd=3,directio
  flowop openfile name=openfile4,filesetname=bigfileset,indexed=$fileidx,
      fd=4,directio
  flowop readwholefile name=readfile4,iosize=$rdiosize,fd=4,directio
  flowop closefile name=closefile3,fd=3
  flowop openfile name=openfile5,filesetname=bigfileset,indexed=$fileidx,
      fd=5,directio
  flowop readwholefile name=readfile5,iosize=$rdiosize,fd=5,directio
  flowop closefile name=closefile4,fd=4
  flowop closefile name=closefile5,fd=5
}

define flowop name=appnd
{
  flowop openfile name=openfile6,filesetname=bigfileset,indexed=$fileidx,
      fd=6,directio
  flowop appendfilerand name=appendfilerand6,iosize=$wrtiosize,fd=6,
      directio
  flowop closefile name=closefile6,fd=6
}

define process name=netclient,instances=1
{
  thread name=fileuser,memsize=10m,instances=$nthreads
  {
    flowop launch name=launch1, iters=1
    flowop rmw name=rmw1, iters=6,directio
    flowop appnd name=appnd1, iters=3,directio
    flowop statfile name=statfile1,filesetname=bigfileset,indexed=
        $fileidx
    flowop eventlimit name=ratecontrol
  }
}

echo  "NetworkFileServer Version 1.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
```

```
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.11 Web proxy (*webproxy.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=600000
set $meandirwidth=1000000
set $meanfilesize=16k
set $nthreads=80
set $meaniosize=16k
set $iosize=1m

define fileset name=bigfileset,path=$dir,size=$meanfilesize,entries=
    $nfiles,dirwidth=$meandirwidth,prealloc=80

define process name=proxycache,instances=1
{
  thread name=proxycache,memsize=10m,instances=$nthreads
  {
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop createfile name=createfile1,filesetname=bigfileset,fd=1
    flowop appendfilerand name=appendfilerand1,iosize=$meaniosize,fd=1
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile2,fd=1,iosize=$iosize
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize
    flowop closefile name=closefile4,fd=1
    flowop openfile name=openfile5,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile5,fd=1,iosize=$iosize
    flowop closefile name=closefile5,fd=1
    flowop openfile name=openfile6,filesetname=bigfileset,fd=1
    flowop readwholefile name=readfile6,fd=1,iosize=$iosize
    flowop closefile name=closefile6,fd=1
    flowop opslimit name=limit
  }
}

echo  "Web proxy-server Version 3.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
```

```
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.12 Web proxy direct I/O (*webproxy-direct.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $nfiles=600000
set $meandirwidth=1000000
set $meanfilesize=16k
set $nthreads=80
set $meaniosize=16k
set $iosize=1m

define fileset name=bigfileset,path=$dir,size=$meanfilesize,entries=
    $nfiles,dirwidth=$meandirwidth,prealloc=80

define process name=proxycache,instances=1
{
  thread name=proxycache,memsize=10m,instances=$nthreads
  {
    flowop deletefile name=deletefile1,filesetname=bigfileset
    flowop createfile name=createfile1,filesetname=bigfileset,fd=1,
        directio
    flowop appendfilerand name=appendfilerand1,iosize=$meaniosize,fd=1,
        directio
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile2,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile4,fd=1
    flowop openfile name=openfile5,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile5,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile5,fd=1
    flowop openfile name=openfile6,filesetname=bigfileset,fd=1,directio
    flowop readwholefile name=readfile6,fd=1,iosize=$iosize,directio
    flowop closefile name=closefile6,fd=1
    flowop opslimit name=limit
  }
}

echo  "Web proxy-server Direct I/O Version 3.0 personality successfully
    loaded"
```

```
echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.13 OLTP database (*oltp.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $eventrate=0
set $iosize=2k
set $nshadows=200
set $ndbwriters=10
set $usermode=200000
set $filesize=10m
set $memperthread=1m
set $workingset=0
set $logfilesize=10m
set $nfiles=1000
set $nlogfiles=1
set $directio=0

eventgen rate = $eventrate

# Define a datafile and logfile
define fileset name=datafiles,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=1024,prealloc=100,reuse
define fileset name=logfile,path=$dir,size=$logfilesize,entries=
    $nlogfiles,dirwidth=1024,prealloc=100,reuse

define process name=lgwr,instances=1
{
  thread name=lgwr,memsize=$memperthread
  {
    flowop aiowrite name=lg-write,filesetname=logfile,
        iosize=256k,random,directio=$directio,dsync
    flowop aiowait name=lg-aiowait
    flowop semblock name=lg-block,value=3200,highwater=1000
  }
}

# Define database writer processes
define process name=dbwr,instances=$ndbwriters
{
  thread name=dbwr,memsize=$memperthread
```

```
  {
    flowop aiowrite name=dbwrite-a,filesetname=datafiles,
        iosize=$iosize,workingset=$workingset,random,iters=100,opennext,
            directio=$directio,dsync
    flowop hog name=dbwr-hog,value=10000
    flowop semblock name=dbwr-block,value=1000,highwater=2000
    flowop aiowait name=dbwr-aiowait
  }
}

define process name=shadow,instances=$nshadows
{
  thread name=shadow,memsize=$memperthread
  {
    flowop read name=shadowread,filesetname=datafiles,
      iosize=$iosize,workingset=$workingset,random,opennext,directio=
          $directio
    flowop hog name=shadowhog,value=$usermode
    flowop sempost name=shadow-post-lg,value=1,target=lg-block,blocking
    flowop sempost name=shadow-post-dbwr,value=1,target=dbwr-block,
        blocking
    flowop eventlimit name=random-rate
  }
}

echo "OLTP Version 3.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

C.3.14 OLTP database direct I/O (*oltp-direct.f*)

```
set $runtime=3600

set $dir=/mnt/nvme
set $eventrate=0
set $iosize=4k
set $nshadows=200
set $ndbwriters=10
set $usermode=200000
set $filesize=10m
set $memperthread=1m
set $workingset=0
set $logfilesize=10m
```

```
set $nfiles=1000
set $nlogfiles=1
set $directio=1

eventgen rate = $eventrate

# Define a datafile and logfile
define fileset name=datafiles,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=1024,prealloc=100,reuse
define fileset name=logfile,path=$dir,size=$logfilesize,entries=
    $nlogfiles,dirwidth=1024,prealloc=100,reuse

define process name=lgwr,instances=1
{
  thread name=lgwr,memsize=$memperthread
  {
    flowop aiowrite name=lg-write,filesetname=logfile,
        iosize=256k,random,directio=$directio,dsync
    flowop aiowait name=lg-aiowait
    flowop semblock name=lg-block,value=3200,highwater=1000
  }
}

# Define database writer processes
define process name=dbwr,instances=$ndbwriters
{
  thread name=dbwr,memsize=$memperthread
  {
    flowop aiowrite name=dbwrite-a,filesetname=datafiles,
        iosize=$iosize,workingset=$workingset,random,iters=100,opennext,
            directio=$directio,dsync
    flowop hog name=dbwr-hog,value=10000
    flowop semblock name=dbwr-block,value=1000,highwater=2000
    flowop aiowait name=dbwr-aiowait
  }
}

define process name=shadow,instances=$nshadows
{
  thread name=shadow,memsize=$memperthread
  {
    flowop read name=shadowread,filesetname=datafiles,
      iosize=$iosize,workingset=$workingset,random,opennext,directio=
          $directio
    flowop hog name=shadowhog,value=$usermode
    flowop sempost name=shadow-post-lg,value=1,target=lg-block,blocking
    flowop sempost name=shadow-post-dbwr,value=1,target=dbwr-block,
        blocking
    flowop eventlimit name=random-rate
  }
}

echo "OLTP Direct I/O Version 3.0 personality successfully loaded"

echo "Working directory: $dir"
echo "Removing old files..."
system "rm -rf $dir/*"
```

```
system "sync"
echo "Trimming SSD..."
system "fstrim -v $dir"
echo "Creating filesets..."
create files
echo "Dropping system caches..."
system "sync"
system "echo 3 > /proc/sys/vm/drop_caches"

echo "Running benchmark for $runtime seconds..."
run $runtime
```

D

Experimental Data

D.1 Sample Output of *opreport*

```
CPU: Core 2, speed 2664 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted)
    with a unit mask of 0x00 (Unhalted core cycles) count 1000000
CPU_CLK_UNHALT...|
  samples|      %|
------------------
    19014 100.000 copy
        CPU_CLK_UNHALT...|
          samples|      %|
        ------------------
            18845 99.1112 kallsyms
              118  0.6206 libc-2.13.so
               51  0.2682 copy
```

D.2 Sample Per-symbol Output of *opreport*

```
CPU: Core 2, speed 2664 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted)
    with a unit mask of 0x00 (Unhalted core cycles) count 1000000
samples  %        image name       symbol name
1991     10.4729  kallsyms         copy_user_generic_string
449       2.3618  kallsyms         get_page_from_freelist
298       1.5675  kallsyms         ext4_mark_iloc_dirty
284       1.4939  kallsyms         do_raw_spin_lock
228       1.1993  kallsyms         ext4_da_get_block_prep
227       1.1940  kallsyms         kmem_cache_alloc
214       1.1257  kallsyms         start_this_handle
203       1.0678  kallsyms         ext4_ext_find_extent
173       0.9100  kallsyms         ext4_ext_map_blocks
163       0.8574  kallsyms         ext4_da_write_end
161       0.8469  kallsyms         __radix_tree_lookup
155       0.8153  kallsyms         generic_perform_write
153       0.8048  kallsyms         generic_file_read_iter
148       0.7785  kallsyms         ____cache_alloc
148       0.7785  kallsyms         ext4_file_write_iter
146       0.7680  kallsyms         __kmalloc
143       0.7522  kallsyms         __alloc_pages_nodemask
143       0.7522  kallsyms         arch_local_irq_save
137       0.7206  kallsyms         add_to_page_cache_lru
136       0.7154  kallsyms         put_page_testzero
135       0.7101  kallsyms         __block_write_begin
133       0.6996  kallsyms         jbd2_journal_stop
132       0.6943  kallsyms         __add_to_page_cache_locked
124       0.6523  kallsyms         ext4_da_write_begin
123       0.6470  kallsyms         test_and_set_bit
116       0.6102  kallsyms         do_get_write_access
115       0.6049  kallsyms         __find_get_block
114       0.5997  kallsyms         alloc_pages_current
109       0.5734  kallsyms         __cache_free.isra.42
108       0.5681  kallsyms         test_and_set_bit
107       0.5628  kallsyms         mark_page_accessed
105       0.5523  kallsyms         fsnotify
103       0.5418  kallsyms         ext4_es_insert_extent
102       0.5365  kallsyms         __wake_up_bit
102       0.5365  kallsyms         pagecache_get_page
102       0.5365  kallsyms         vfs_write
101       0.5313  kallsyms         __rmqueue
101       0.5313  kallsyms         copy_page_to_iter
100       0.5260  kallsyms         __ext4_handle_dirty_metadata
98        0.5155  kallsyms         ext4_get_inode_flags
97        0.5102  kallsyms         jbd_unlock_bh_journal_head
96        0.5050  kallsyms         zone_dirty_ok
91        0.4787  kallsyms         ext4_es_lookup_extent
90        0.4734  kallsyms         __inc_zone_state
89        0.4682  kallsyms         _raw_spin_lock_irqsave
** OUTPUT TRUNCATED TO ONE PAGE **
```

D.3 Server Platform Filebench Outputs

The following show raw outputs from Filebench benchmarks on the server platform. Information about the performance metrics collected by Filebench during benchmarking can be found at [157].

D.3.1 *webserver.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.043: Web-server Version 3.1 personality successfully loaded
31.717: Populating and pre-allocating filesets
31.717: logfiles populated: 1 files, avg. dir. width = 20, avg. dir.
    depth = 0.0, 0 leafdirs, 0.002MB total size
31.717: Removing logfiles tree (if exists)
31.719: Pre-allocating directories in logfiles tree
31.719: Pre-allocating files in logfiles tree
32.454: bigfileset populated: 600000 files, avg. dir. width = 20, avg.
    dir. depth = 4.4, 0 leafdirs, 9392.595MB total size
32.454: Removing bigfileset tree (if exists)
32.456: Pre-allocating directories in bigfileset tree
34.135: Pre-allocating files in bigfileset tree
125.829: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
125.829: Population and pre-allocation of filesets completed
125.850: Dropping system caches...
129.261: Running benchmark for 3600 seconds...
129.261: Attempting to create fileset more than once, ignoring
129.326: Starting 1 filereader instances
130.763: Running...
3732.233: Run took 3600 seconds...
3732.507: Per-Operation Breakdown
appendlog            6489742ops     1802ops/s  14.1mb/s    0.543ms/op
    [0.008ms - 100.792ms]
closefile10          6489693ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 40.197ms]
readfile10           6489696ops     1802ops/s  28.2mb/s    1.928ms/op
    [0.009ms - 192.337ms]
openfile10           6489696ops     1802ops/s   0.0mb/s    0.426ms/op
    [0.013ms - 643.764ms]
closefile9           6489696ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 33.124ms]
readfile9            6489699ops     1802ops/s  28.2mb/s    1.925ms/op
    [0.011ms - 193.451ms]
openfile9            6489699ops     1802ops/s   0.0mb/s    0.427ms/op
    [0.014ms - 482.260ms]
closefile8           6489699ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 39.594ms]
readfile8            6489706ops     1802ops/s  28.2mb/s    1.924ms/op
    [0.011ms - 192.419ms]
openfile8            6489706ops     1802ops/s   0.0mb/s    0.427ms/op
    [0.013ms - 639.902ms]
closefile7           6489707ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 29.160ms]
```

```
readfile7           6489713ops     1802ops/s  28.2mb/s    1.926ms/op
    [0.010ms - 191.726ms]
openfile7           6489714ops     1802ops/s   0.0mb/s    0.428ms/op
    [0.014ms - 640.048ms]
closefile6          6489714ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 30.061ms]
readfile6           6489720ops     1802ops/s  28.2mb/s    1.926ms/op
    [0.009ms - 192.347ms]
openfile6           6489720ops     1802ops/s   0.0mb/s    0.426ms/op
    [0.014ms - 642.626ms]
closefile5          6489720ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 39.608ms]
readfile5           6489722ops     1802ops/s  28.2mb/s    1.927ms/op
    [0.011ms - 192.221ms]
openfile5           6489722ops     1802ops/s   0.0mb/s    0.427ms/op
    [0.014ms - 641.259ms]
closefile4          6489722ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 37.758ms]
readfile4           6489728ops     1802ops/s  28.2mb/s    1.927ms/op
    [0.014ms - 193.954ms]
openfile4           6489728ops     1802ops/s   0.0mb/s    0.430ms/op
    [0.013ms - 639.098ms]
closefile3          6489728ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 44.112ms]
readfile3           6489730ops     1802ops/s  28.2mb/s    1.933ms/op
    [0.016ms - 129.260ms]
openfile3           6489730ops     1802ops/s   0.0mb/s    0.431ms/op
    [0.014ms - 481.341ms]
closefile2          6489730ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 64.431ms]
readfile2           6489736ops     1802ops/s  28.2mb/s    1.945ms/op
    [0.013ms - 128.485ms]
openfile2           6489736ops     1802ops/s   0.0mb/s    0.474ms/op
    [0.014ms - 638.071ms]
closefile1          6489736ops     1802ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 32.726ms]
readfile1           6489741ops     1802ops/s  28.2mb/s    2.869ms/op
    [0.011ms - 133.200ms]
openfile1           6489741ops     1802ops/s   0.0mb/s    0.457ms/op
    [0.015ms - 641.684ms]
3732.507: IO Summary: 201181270 ops 55860.493 ops/s 18020/1802 rd/wr
    296.2mb/s 0.814ms/op
3732.507: Shutting down processes
```

D.3.2 *webserver-direct.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.043: Web-server Direct I/O Version 3.1 personality successfully loaded
27.731: Populating and pre-allocating filesets
27.747: logfiles populated: 1 files, avg. dir. width = 20, avg. dir.
    depth = 0.0, 0 leafdirs, 0.004MB total size
27.748: Removing logfiles tree (if exists)
27.749: Pre-allocating directories in logfiles tree
27.750: Pre-allocating files in logfiles tree
```

```
28.492: bigfileset populated: 600000 files, avg. dir. width = 20, avg.
    dir. depth = 4.4, 0 leafdirs, 10577.414MB total size
28.492: Removing bigfileset tree (if exists)
28.495: Pre-allocating directories in bigfileset tree
30.370: Pre-allocating files in bigfileset tree
118.235: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
118.235: Population and pre-allocation of filesets completed
118.517: Dropping system caches...
121.897: Running benchmark for 3600 seconds...
121.897: Attempting to create fileset more than once, ignoring
121.994: Starting 1 filereader instances
123.399: Running...
3723.812: Run took 3600 seconds...
3723.835: Per-Operation Breakdown
appendlog          10370291ops     2880ops/s  28.1mb/s   10.584ms/op
    [0.001ms - 120.167ms]
closefile10        10370247ops     2880ops/s   0.0mb/s    0.007ms/op
    [0.001ms - 22.888ms]
readfile10         10370250ops     2880ops/s  50.8mb/s    0.581ms/op
    [0.061ms - 103.927ms]
openfile10         10370250ops     2880ops/s   0.0mb/s    0.046ms/op
    [0.006ms - 36.834ms]
closefile9         10370250ops     2880ops/s   0.0mb/s    0.007ms/op
    [0.001ms - 20.992ms]
readfile9          10370250ops     2880ops/s  50.8mb/s    0.581ms/op
    [0.063ms - 104.357ms]
openfile9          10370250ops     2880ops/s   0.0mb/s    0.046ms/op
    [0.006ms - 52.891ms]
closefile8         10370250ops     2880ops/s   0.0mb/s    0.007ms/op
    [0.001ms - 27.536ms]
readfile8          10370252ops     2880ops/s  50.8mb/s    0.584ms/op
    [0.065ms - 98.445ms]
openfile8          10370252ops     2880ops/s   0.0mb/s    0.046ms/op
    [0.006ms - 35.737ms]
closefile7         10370252ops     2880ops/s   0.0mb/s    0.007ms/op
    [0.001ms - 29.979ms]
readfile7          10370255ops     2880ops/s  50.8mb/s    0.588ms/op
    [0.070ms - 100.028ms]
openfile7          10370255ops     2880ops/s   0.0mb/s    0.045ms/op
    [0.005ms - 209.950ms]
closefile6         10370255ops     2880ops/s   0.0mb/s    0.006ms/op
    [0.001ms - 24.265ms]
readfile6          10370257ops     2880ops/s  50.8mb/s    0.593ms/op
    [0.068ms - 91.493ms]
openfile6          10370257ops     2880ops/s   0.0mb/s    0.045ms/op
    [0.005ms - 70.446ms]
closefile5         10370257ops     2880ops/s   0.0mb/s    0.006ms/op
    [0.001ms - 33.317ms]
readfile5          10370260ops     2880ops/s  50.8mb/s    0.600ms/op
    [0.064ms - 100.044ms]
openfile5          10370260ops     2880ops/s   0.0mb/s    0.045ms/op
    [0.005ms - 35.700ms]
closefile4         10370260ops     2880ops/s   0.0mb/s    0.006ms/op
    [0.001ms - 20.098ms]
readfile4          10370263ops     2880ops/s  50.8mb/s    0.608ms/op
    [0.063ms - 99.061ms]
```

```
openfile4           10370263ops     2880ops/s   0.0mb/s    0.044ms/op
    [0.005ms - 31.318ms]
closefile3          10370263ops     2880ops/s   0.0mb/s    0.006ms/op
    [0.001ms - 36.098ms]
readfile3           10370267ops     2880ops/s  50.8mb/s    0.616ms/op
    [0.066ms - 107.105ms]
openfile3           10370267ops     2880ops/s   0.0mb/s    0.043ms/op
    [0.005ms - 37.426ms]
closefile2          10370267ops     2880ops/s   0.0mb/s    0.006ms/op
    [0.001ms - 23.628ms]
readfile2           10370268ops     2880ops/s  50.8mb/s    0.622ms/op
    [0.059ms - 107.488ms]
openfile2           10370268ops     2880ops/s   0.0mb/s    0.042ms/op
    [0.006ms - 32.403ms]
closefile1          10370268ops     2880ops/s   0.0mb/s    0.006ms/op
    [0.001ms - 28.647ms]
readfile1           10370271ops     2880ops/s  50.8mb/s    0.614ms/op
    [0.058ms - 104.452ms]
openfile1           10370272ops     2880ops/s   0.0mb/s    0.043ms/op
    [0.006ms - 70.409ms]
3723.835: IO Summary: 321478047 ops 89289.493 ops/s 28803/2880 rd/wr
    535.9mb/s 0.551ms/op
3723.836: Shutting down processes
```

D.3.3 *fileserver.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.074: File-server Version 3.0 personality successfully loaded
4.642: Populating and pre-allocating filesets
4.743: bigfileset populated: 80000 files, avg. dir. width = 20, avg. dir.
     depth = 3.8, 0 leafdirs, 10019.570MB total size
4.743: Removing bigfileset tree (if exists)
4.745: Pre-allocating directories in bigfileset tree
4.974: Pre-allocating files in bigfileset tree
33.264: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
33.264: Population and pre-allocation of filesets completed
33.291: Dropping system caches...
34.750: Running benchmark for 3600 seconds...
34.750: Attempting to create fileset more than once, ignoring
34.808: Starting 1 filereader instances
36.283: Running...
3639.730: Run took 3600 seconds...
3640.041: Per-Operation Breakdown
statfile1           4574542ops     1269ops/s   0.0mb/s    0.271ms/op
    [0.003ms - 228.039ms]
deletefile1         4574544ops     1269ops/s   0.0mb/s    2.180ms/op
    [0.047ms - 304.056ms]
closefile3          4574544ops     1269ops/s   0.0mb/s    0.016ms/op
    [0.001ms - 181.023ms]
readfile1           4574550ops     1269ops/s 168.9mb/s    7.473ms/op
    [0.004ms - 377.563ms]
openfile2           4574551ops     1269ops/s   0.0mb/s    1.243ms/op
    [0.006ms - 321.771ms]
```

```
closefile2          4574551ops     1269ops/s   0.0mb/s    0.017ms/op
    [0.001ms - 147.937ms]
appendfilerand1     4574556ops     1269ops/s   9.9mb/s   11.223ms/op
    [0.001ms - 342.287ms]
openfile1           4574558ops     1269ops/s   0.0mb/s    1.111ms/op
    [0.007ms - 287.942ms]
closefile1          4574558ops     1269ops/s   0.0mb/s    0.037ms/op
    [0.001ms - 208.366ms]
wrtfile1            4574562ops     1269ops/s 159.0mb/s    7.995ms/op
    [0.016ms - 532.528ms]
createfile1         4574591ops     1269ops/s   0.0mb/s    5.563ms/op
    [0.023ms - 302.553ms]
3640.041: IO Summary: 50320107 ops 13964.296 ops/s 1269/2539 rd/wr 337.9
    mb/s 3.375ms/op
3640.041: Shutting down processes
```

D.3.4 *fileserver-direct.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.016: File-server Direct I/O Version 3.0 personality successfully loaded
0.873: Populating and pre-allocating filesets
0.970: bigfileset populated: 80000 files, avg. dir. width = 20, avg. dir.
     depth = 3.8, 0 leafdirs, 10175.633MB total size
0.970: Removing bigfileset tree (if exists)
0.972: Pre-allocating directories in bigfileset tree
1.148: Pre-allocating files in bigfileset tree
30.130: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
30.130: Population and pre-allocation of filesets completed
30.142: Dropping system caches...
31.877: Running benchmark for 3600 seconds...
31.877: Attempting to create fileset more than once, ignoring
31.941: Starting 1 filereader instances
33.393: Running...
3633.768: Run took 3600 seconds...
3633.771: Per-Operation Breakdown
statfile1           12934693ops     3593ops/s   0.0mb/s    0.017ms/op
    [0.003ms - 75.618ms]
deletefile1         12934693ops     3593ops/s   0.0mb/s    0.529ms/op
    [0.036ms - 215.739ms]
closefile3          12934693ops     3593ops/s   0.0mb/s    0.007ms/op
    [0.001ms - 44.845ms]
readfile1           12934698ops     3593ops/s 492.1mb/s    2.807ms/op
    [0.075ms - 181.678ms]
openfile2           12934700ops     3593ops/s   0.0mb/s    0.037ms/op
    [0.005ms - 148.913ms]
closefile2          12934700ops     3593ops/s   0.0mb/s    0.016ms/op
    [0.001ms - 65.702ms]
appendfilerand1     12934709ops     3593ops/s  35.1mb/s    3.381ms/op
    [0.001ms - 204.475ms]
openfile1           12934711ops     3593ops/s   0.0mb/s    0.039ms/op
    [0.005ms - 154.798ms]
closefile1          12934711ops     3593ops/s   0.0mb/s    0.016ms/op
    [0.001ms - 71.245ms]
```

```
wrtfile1            12934733ops     3593ops/s 457.2mb/s    6.303ms/op
    [0.044ms - 238.459ms]
createfile1         12934741ops     3593ops/s   0.0mb/s    0.478ms/op
    [0.021ms - 175.765ms]
3633.771: IO Summary: 142281782 ops 39518.611 ops/s 3593/7185 rd/wr 984.4
    mb/s 1.239ms/op
3633.771: Shutting down processes
```

D.3.5 *varmail.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.026: Varmail Version 3.0 personality successfully loaded
30.946: Populating and pre-allocating filesets
31.911: bigfileset populated: 600000 files, avg. dir. width = 1000000,
    avg. dir. depth = 1.0, 0 leafdirs, 9386.366MB total size
31.911: Removing bigfileset tree (if exists)
31.914: Pre-allocating directories in bigfileset tree
31.914: Pre-allocating files in bigfileset tree
102.633: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
102.633: Population and pre-allocation of filesets completed
102.848: Dropping system caches...
106.644: Running benchmark for 3600 seconds...
106.644: Attempting to create fileset more than once, ignoring
106.667: Starting 1 filereader instances
108.083: Running...
3708.575: Run took 3600 seconds...
3708.575: Per-Operation Breakdown
closefile4          6222236ops     1728ops/s   0.0mb/s    0.008ms/op
    [0.001ms - 16.703ms]
readfile4           6222237ops     1728ops/s  26.8mb/s    0.916ms/op
    [0.006ms - 41.143ms]
openfile4           6222237ops     1728ops/s   0.0mb/s    1.388ms/op
    [0.012ms - 183.896ms]
closefile3          6222239ops     1728ops/s   0.0mb/s    0.008ms/op
    [0.001ms - 12.455ms]
fsyncfile3          6222242ops     1728ops/s   0.0mb/s    0.765ms/op
    [0.003ms - 28.652ms]
appendfilerand3     6222243ops     1728ops/s  13.5mb/s    0.322ms/op
    [0.012ms - 29.161ms]
readfile3           6222244ops     1728ops/s  26.8mb/s    0.923ms/op
    [0.006ms - 33.103ms]
openfile3           6222246ops     1728ops/s   0.0mb/s    1.401ms/op
    [0.012ms - 146.864ms]
closefile2          6222248ops     1728ops/s   0.0mb/s    0.008ms/op
    [0.001ms - 16.088ms]
fsyncfile2          6222248ops     1728ops/s   0.0mb/s    0.797ms/op
    [0.007ms - 32.091ms]
appendfilerand2     6222249ops     1728ops/s  13.5mb/s    0.248ms/op
    [0.001ms - 24.459ms]
createfile2         6222249ops     1728ops/s   0.0mb/s    0.855ms/op
    [0.028ms - 184.398ms]
deletefile1         6222251ops     1728ops/s   0.0mb/s    1.567ms/op
    [0.046ms - 37.894ms]
```

```
3708.575: IO Summary: 80889169 ops 22466.147 ops/s 3456/3456 rd/wr  80.5
    mb/s 0.708ms/op
3708.575: Shutting down processes
```

D.3.6 *varmail-direct.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.068: Varmail Direct I/O Version 3.0 personality successfully loaded
1.830: Populating and pre-allocating filesets
2.745: bigfileset populated: 600000 files, avg. dir. width = 1000000, avg
    . dir. depth = 1.0, 0 leafdirs, 10571.594MB total size
2.745: Removing bigfileset tree (if exists)
2.747: Pre-allocating directories in bigfileset tree
2.748: Pre-allocating files in bigfileset tree
69.753: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
69.753: Population and pre-allocation of filesets completed
69.769: Dropping system caches...
73.456: Running benchmark for 3600 seconds...
73.456: Attempting to create fileset more than once, ignoring
73.496: Starting 1 filereader instances
74.928: Running...
3675.355: Run took 3600 seconds...
3675.356: Per-Operation Breakdown
closefile4          6708969ops     1863ops/s   0.0mb/s    0.006ms/op
    [0.001ms - 10.197ms]
readfile4           6708970ops     1863ops/s  35.9mb/s    0.926ms/op
    [0.002ms - 41.578ms]
openfile4           6708970ops     1863ops/s   0.0mb/s    0.054ms/op
    [0.005ms - 178.252ms]
closefile3          6708970ops     1863ops/s   0.0mb/s    0.008ms/op
    [0.001ms - 5.140ms]
fsyncfile3          6708970ops     1863ops/s   0.0mb/s    0.532ms/op
    [0.002ms - 30.254ms]
appendfilerand3     6708972ops     1863ops/s  18.2mb/s    0.764ms/op
    [0.001ms - 74.166ms]
readfile3           6708973ops     1863ops/s  35.9mb/s    0.933ms/op
    [0.002ms - 35.119ms]
openfile3           6708973ops     1863ops/s   0.0mb/s    0.053ms/op
    [0.005ms - 179.635ms]
closefile2          6708973ops     1863ops/s   0.0mb/s    0.007ms/op
    [0.001ms - 20.403ms]
fsyncfile2          6708974ops     1863ops/s   0.0mb/s    0.545ms/op
    [0.002ms - 27.980ms]
appendfilerand2     6708975ops     1863ops/s  18.2mb/s    0.742ms/op
    [0.047ms - 46.853ms]
createfile2         6708977ops     1863ops/s   0.0mb/s    1.696ms/op
    [0.027ms - 178.953ms]
deletefile1         6708982ops     1863ops/s   0.0mb/s    2.276ms/op
    [0.044ms - 56.369ms]
3675.356: IO Summary: 87216648 ops 24223.974 ops/s 3727/3727 rd/wr 108.2
    mb/s 0.657ms/op
3675.356: Shutting down processes
```

D.3.7 *videoserver.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.036: Video Server Version 3.0 personality successfully loaded
3.455: Populating and pre-allocating filesets
3.455: passivevids populated: 28 files, avg. dir. width = 20, avg. dir.
    depth = 1.1, 0 leafdirs, 286720.000MB total size
3.455: Removing passivevids tree (if exists)
3.457: Pre-allocating directories in passivevids tree
3.458: Pre-allocating files in passivevids tree
3.463: activevids populated: 8 files, avg. dir. width = 4, avg. dir.
    depth = 1.5, 0 leafdirs, 81920.000MB total size
3.463: Removing activevids tree (if exists)
3.473: Pre-allocating directories in activevids tree
3.474: Pre-allocating files in activevids tree
707.311: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
722.534: Population and pre-allocation of filesets completed
722.547: Dropping system caches...
724.822: Running benchmark for 3600 seconds...
724.823: Attempting to create fileset more than once, ignoring
724.882: Starting 1 vidreaders instances
724.948: Starting 1 vidwriter instances
726.287: Running...
4326.638: Run took 3600 seconds...
4326.915: Per-Operation Breakdown
serverlimit          1381120ops      384ops/s   0.0mb/s  124.961ms/op
    [0.001ms - 5033.931ms]
vidreader            1381264ops      384ops/s  95.9mb/s    0.281ms/op
    [0.020ms - 235.589ms]
replaceinterval      86ops        0ops/s   0.0mb/s 10000.083ms/op
    [10000.070ms - 10000.101ms]
wrtclose             87ops        0ops/s   0.0mb/s    0.009ms/op [0.008ms
     - 0.015ms]
newvid               87ops        0ops/s 247.4mb/s 31398.621ms/op
    [29949.600ms - 34653.190ms]
wrtopen              87ops        0ops/s   0.0mb/s    0.112ms/op [0.054ms
     - 0.227ms]
vidremover           87ops        0ops/s   0.0mb/s   22.202ms/op [7.357ms
     - 457.466ms]
4326.915: IO Summary: 1381612 ops 383.740 ops/s 384/0 rd/wr 343.3mb/s
    2.259ms/op
4326.915: Shutting down processes
```

D.3.8 *videoserver-direct.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.046: Video Server Direct I/O Version 3.0 personality successfully
    loaded
33.208: Populating and pre-allocating filesets
33.209: passivevids populated: 28 files, avg. dir. width = 20, avg. dir.
    depth = 1.1, 0 leafdirs, 286720.000MB total size
```

```
33.209: Removing passivevids tree (if exists)
33.211: Pre-allocating directories in passivevids tree
33.211: Pre-allocating files in passivevids tree
33.218: activevids populated: 8 files, avg. dir. width = 4, avg. dir.
    depth = 1.5, 0 leafdirs, 81920.000MB total size
33.218: Removing activevids tree (if exists)
33.262: Pre-allocating directories in activevids tree
33.263: Pre-allocating files in activevids tree
738.542: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
755.479: Population and pre-allocation of filesets completed
755.499: Dropping system caches...
757.482: Running benchmark for 3600 seconds...
757.482: Attempting to create fileset more than once, ignoring
757.543: Starting 1 vidreaders instances
757.583: Starting 1 vidwriter instances
758.980: Running...
4359.227: Run took 3600 seconds...
4359.227: Per-Operation Breakdown
serverlimit          1381480ops      384ops/s   0.0mb/s  124.759ms/op
    [0.001ms - 4993.666ms]
vidreader            1381624ops      384ops/s  95.9mb/s    0.979ms/op
    [0.023ms - 36.742ms]
replaceinterval      148ops        0ops/s   0.0mb/s 10000.076ms/op
    [10000.064ms - 10000.111ms]
wrtclose             149ops        0ops/s   0.0mb/s    0.006ms/op [0.004
    ms - 0.022ms]
newvid               149ops        0ops/s 423.8mb/s 14201.828ms/op
    [12393.277ms - 28669.766ms]
wrtopen              149ops        0ops/s   0.0mb/s    0.049ms/op [0.037
    ms - 0.139ms]
vidremover           149ops        0ops/s   0.0mb/s    5.365ms/op [0.909
    ms - 338.713ms]
4359.227: IO Summary: 1382220 ops 383.924 ops/s 384/0 rd/wr 519.7mb/s
    2.510ms/op
4359.227: Shutting down processes
```

D.3.9 *netsfs.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.039: NetworkFileServer Version 1.0 personality successfully loaded
15.086: Populating and pre-allocating filesets
15.501: bigfileset populated: 400000 files, avg. dir. width = 20, avg.
    dir. depth = 4.3, 0 leafdirs, 10391.271MB total size
15.501: Removing bigfileset tree (if exists)
15.504: Pre-allocating directories in bigfileset tree
16.712: Pre-allocating files in bigfileset tree
57.736: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
57.736: Population and pre-allocation of filesets completed
57.744: Dropping system caches...
60.643: Running benchmark for 3600 seconds...
60.643: Attempting to create fileset more than once, ignoring
60.680: Starting 1 netclient instances
62.034: Running...
```

```
3662.275: Run took 3600 seconds...
3662.275: Per-Operation Breakdown
ratecontrol          35990ops       10ops/s   0.0mb/s   99.107ms/op
    [0.000ms - 994.281ms]
statfile1            35991ops       10ops/s   0.0mb/s    0.003ms/op
    [0.002ms - 0.067ms]
appnd1.closefile6    107973ops       30ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.057ms]
appnd1.appendfilerand6 107973ops       30ops/s   0.5mb/s    0.053ms/op
    [0.006ms - 0.915ms]
appnd1.openfile6     107973ops       30ops/s   0.0mb/s    0.006ms/op
    [0.004ms - 0.299ms]
appnd1               0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
rmw1.deletefile1     215946ops       60ops/s   0.0mb/s    0.037ms/op
    [0.025ms - 0.248ms]
rmw1.closefile2      215946ops       60ops/s   0.0mb/s    0.001ms/op
    [0.001ms - 0.034ms]
rmw1.closefile1      215946ops       60ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.059ms]
rmw1.writefile2      215946ops       60ops/s   0.1mb/s    0.016ms/op
    [0.011ms - 0.085ms]
rmw1.newfile2        215946ops       60ops/s   0.0mb/s    0.027ms/op
    [0.019ms - 0.389ms]
rmw1.readfile1       215946ops       60ops/s   0.6mb/s    0.006ms/op
    [0.002ms - 0.149ms]
rmw1.openfile1       215946ops       60ops/s   0.0mb/s    0.007ms/op
    [0.004ms - 0.638ms]
rmw1                 0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
launch1.closefile5   35991ops       10ops/s   0.0mb/s    0.001ms/op
    [0.000ms - 0.060ms]
launch1.closefile4   35991ops       10ops/s   0.0mb/s    0.001ms/op
    [0.001ms - 0.047ms]
launch1.readfile5    35991ops       10ops/s   0.5mb/s    0.016ms/op
    [0.002ms - 0.122ms]
launch1.openfile5    35991ops       10ops/s   0.0mb/s    0.006ms/op
    [0.004ms - 0.335ms]
launch1.closefile3   35991ops       10ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.035ms]
launch1.readfile4    35991ops       10ops/s   0.5mb/s    0.016ms/op
    [0.002ms - 0.120ms]
launch1.openfile4    35991ops       10ops/s   0.0mb/s    0.006ms/op
    [0.004ms - 0.379ms]
launch1.readfile3    35991ops       10ops/s   0.5mb/s    0.021ms/op
    [0.002ms - 0.140ms]
launch1.openfile3    35991ops       10ops/s   0.0mb/s    0.006ms/op
    [0.003ms - 1.469ms]
launch1              0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
3662.275: IO Summary: 2195451 ops 609.808 ops/s 90/90 rd/wr   2.7mb/s
    0.014ms/op
3662.275: Shutting down processes
```

D.3.10 *netsfs-direct.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.054: NetworkFileServer Version 1.0 personality successfully loaded
6.624: Populating and pre-allocating filesets
6.975: bigfileset populated: 400000 files, avg. dir. width = 20, avg. dir
    . depth = 4.3, 0 leafdirs, 10359.824MB total size
6.975: Removing bigfileset tree (if exists)
6.978: Pre-allocating directories in bigfileset tree
8.205: Pre-allocating files in bigfileset tree
49.393: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
49.393: Population and pre-allocation of filesets completed
49.488: Dropping system caches...
51.868: Running benchmark for 3600 seconds...
51.868: Attempting to create fileset more than once, ignoring
51.868: Starting 1 netclient instances
53.366: Running...
3653.626: Run took 3600 seconds...
3653.627: Per-Operation Breakdown
ratecontrol          36000ops       10ops/s   0.0mb/s   94.380ms/op
    [0.000ms - 967.792ms]
statfile1            36001ops       10ops/s   0.0mb/s    0.004ms/op
    [0.002ms - 0.066ms]
appnd1.closefile6    108003ops       30ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.040ms]
appnd1.appendfilerand6 108003ops       30ops/s   0.6mb/s    0.073ms/op
    [0.046ms - 0.260ms]
appnd1.openfile6     108003ops       30ops/s   0.0mb/s    0.007ms/op
    [0.004ms - 0.404ms]
appnd1               0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
rmw1.deletefile1     216006ops       60ops/s   0.0mb/s    0.039ms/op
    [0.026ms - 0.205ms]
rmw1.closefile2      216006ops       60ops/s   0.0mb/s    0.001ms/op
    [0.001ms - 0.062ms]
rmw1.closefile1      216006ops       60ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.060ms]
rmw1.writefile2      216006ops       60ops/s   0.2mb/s    0.015ms/op
    [0.011ms - 0.115ms]
rmw1.newfile2        216006ops       60ops/s   0.0mb/s    0.026ms/op
    [0.020ms - 0.923ms]
rmw1.readfile1       216006ops       60ops/s   0.8mb/s    0.086ms/op
    [0.005ms - 3.954ms]
rmw1.openfile1       216006ops       60ops/s   0.0mb/s    0.006ms/op
    [0.005ms - 0.334ms]
rmw1                 0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
launch1.closefile5   36001ops       10ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.044ms]
launch1.closefile4   36001ops       10ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.030ms]
launch1.readfile5    36001ops       10ops/s   0.6mb/s    0.759ms/op
    [0.046ms - 5.662ms]
launch1.openfile5    36001ops       10ops/s   0.0mb/s    0.007ms/op
    [0.005ms - 0.342ms]
launch1.closefile3   36001ops       10ops/s   0.0mb/s    0.002ms/op
    [0.001ms - 0.049ms]
```

```
launch1.readfile4    36001ops       10ops/s   0.6mb/s    0.815ms/op
    [0.046ms - 6.460ms]
launch1.openfile4    36001ops       10ops/s   0.0mb/s    0.007ms/op
    [0.005ms - 0.344ms]
launch1.readfile3    36001ops       10ops/s   0.6mb/s    2.642ms/op
    [0.053ms - 11.963ms]
launch1.openfile3    36001ops       10ops/s   0.0mb/s    0.006ms/op
    [0.004ms - 1.372ms]
launch1              0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
3653.627: IO Summary: 2196061 ops 609.974 ops/s 90/90 rd/wr   3.4mb/s
    0.091ms/op
3653.627: Shutting down processes
```

D.3.11 *webproxy.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.029: Web proxy-server Version 3.0 personality successfully loaded
26.326: Populating and pre-allocating filesets
26.908: bigfileset populated: 600000 files, avg. dir. width = 1000000,
    avg. dir. depth = 1.0, 0 leafdirs, 9375.000MB total size
26.908: Removing bigfileset tree (if exists)
26.910: Pre-allocating directories in bigfileset tree
26.910: Pre-allocating files in bigfileset tree
92.598: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
92.598: Population and pre-allocation of filesets completed
92.626: Dropping system caches...
95.792: Running benchmark for 3600 seconds...
95.792: Attempting to create fileset more than once, ignoring
95.829: Starting 1 proxycache instances
97.257: Running...
3698.417: Run took 3600 seconds...
3698.532: Per-Operation Breakdown
limit                0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
closefile6           8929335ops     2480ops/s   0.0mb/s    0.013ms/op
    [0.001ms - 99.737ms]
readfile6            8929336ops     2480ops/s  20.4mb/s    0.652ms/op
    [0.002ms - 175.368ms]
openfile6            8929348ops     2480ops/s   0.0mb/s    4.601ms/op
    [0.004ms - 544.030ms]
closefile5           8929348ops     2480ops/s   0.0mb/s    0.013ms/op
    [0.001ms - 119.813ms]
readfile5            8929349ops     2480ops/s  20.4mb/s    0.662ms/op
    [0.002ms - 182.361ms]
openfile5            8929360ops     2480ops/s   0.0mb/s    4.596ms/op
    [0.004ms - 566.784ms]
closefile4           8929360ops     2480ops/s   0.0mb/s    0.013ms/op
    [0.001ms - 101.470ms]
readfile4            8929360ops     2480ops/s  20.4mb/s    0.682ms/op
    [0.002ms - 158.664ms]
openfile4            8929375ops     2480ops/s   0.0mb/s    4.581ms/op
    [0.004ms - 588.822ms]
```

```
closefile3          8929375ops     2480ops/s   0.0mb/s    0.014ms/op
    [0.001ms - 136.947ms]
readfile3           8929376ops     2480ops/s  20.4mb/s    0.730ms/op
    [0.002ms - 177.120ms]
openfile3           8929392ops     2480ops/s   0.0mb/s    4.657ms/op
    [0.004ms - 564.877ms]
closefile2          8929392ops     2480ops/s   0.0mb/s    0.013ms/op
    [0.001ms - 117.591ms]
readfile2           8929392ops     2480ops/s  20.4mb/s    1.106ms/op
    [0.002ms - 188.108ms]
openfile2           8929396ops     2480ops/s   0.0mb/s    1.806ms/op
    [0.005ms - 540.326ms]
closefile1          8929396ops     2480ops/s   0.0mb/s    0.009ms/op
    [0.001ms - 105.800ms]
appendfilerand1     8929396ops     2480ops/s  19.4mb/s    0.137ms/op
    [0.001ms - 162.062ms]
createfile1         8929401ops     2480ops/s   0.0mb/s    2.321ms/op
    [0.026ms - 545.293ms]
deletefile1         8929410ops     2480ops/s   0.0mb/s    5.178ms/op
    [0.038ms - 266.720ms]
3698.532: IO Summary: 169658097 ops 47111.642 ops/s 12398/2480 rd/wr
    121.4mb/s 1.673ms/op
3698.532: Shutting down processes
```

D.3.12 *webproxy-direct.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.035: Web proxy-server Direct I/O Version 3.0 personality successfully
    loaded
6.027: Populating and pre-allocating filesets
6.631: bigfileset populated: 600000 files, avg. dir. width = 1000000, avg
    . dir. depth = 1.0, 0 leafdirs, 9375.000MB total size
6.631: Removing bigfileset tree (if exists)
6.633: Pre-allocating directories in bigfileset tree
6.634: Pre-allocating files in bigfileset tree
71.151: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
71.151: Population and pre-allocation of filesets completed
71.236: Dropping system caches...
74.561: Running benchmark for 3600 seconds...
74.561: Attempting to create fileset more than once, ignoring
74.605: Starting 1 proxycache instances
76.067: Running...
3676.562: Run took 3600 seconds...
3676.589: Per-Operation Breakdown
limit               0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
closefile6          12470878ops     3464ops/s   0.0mb/s    0.013ms/op
    [0.001ms - 119.612ms]
readfile6           12470880ops     3464ops/s  34.6mb/s    0.875ms/op
    [0.002ms - 200.958ms]
openfile6           12470881ops     3464ops/s   0.0mb/s    0.089ms/op
    [0.004ms - 345.564ms]
closefile5          12470881ops     3464ops/s   0.0mb/s    0.013ms/op
    [0.001ms - 141.140ms]
```

```
readfile5           12470884ops     3464ops/s  34.6mb/s    0.876ms/op
    [0.002ms - 221.768ms]
openfile5           12470884ops     3464ops/s   0.0mb/s    0.089ms/op
    [0.004ms - 319.335ms]
closefile4          12470885ops     3464ops/s   0.0mb/s    0.013ms/op
    [0.001ms - 141.617ms]
readfile4           12470887ops     3464ops/s  34.6mb/s    0.882ms/op
    [0.002ms - 213.123ms]
openfile4           12470889ops     3464ops/s   0.0mb/s    0.089ms/op
    [0.004ms - 293.359ms]
closefile3          12470889ops     3464ops/s   0.0mb/s    0.012ms/op
    [0.001ms - 132.189ms]
readfile3           12470890ops     3464ops/s  34.6mb/s    0.895ms/op
    [0.001ms - 240.804ms]
openfile3           12470892ops     3464ops/s   0.0mb/s    0.088ms/op
    [0.004ms - 365.192ms]
closefile2          12470894ops     3464ops/s   0.0mb/s    0.012ms/op
    [0.001ms - 141.000ms]
readfile2           12470897ops     3464ops/s  34.6mb/s    0.990ms/op
    [0.002ms - 211.230ms]
openfile2           12470898ops     3464ops/s   0.0mb/s    0.083ms/op
    [0.005ms - 308.532ms]
closefile1          12470898ops     3464ops/s   0.0mb/s    0.011ms/op
    [0.001ms - 156.520ms]
appendfilerand1     12470902ops     3464ops/s  33.8mb/s    1.971ms/op
    [0.001ms - 413.090ms]
createfile1         12470917ops     3464ops/s   0.0mb/s    4.189ms/op
    [0.026ms - 531.148ms]
deletefile1         12470784ops     3464ops/s   0.0mb/s   11.181ms/op
    [0.035ms - 408.332ms]
3676.589: IO Summary: 236946810 ops 65809.575 ops/s 17318/3464 rd/wr
    206.9mb/s 1.177ms/op
3676.589: Shutting down processes
```

*D.*3.13 *oltp.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.174: OLTP Version 3.0 personality successfully loaded
2.197: Populating and pre-allocating filesets
2.235: logfile populated: 1 files, avg. dir. width = 1024, avg. dir.
    depth = 0.0, 0 leafdirs, 10.000MB total size
2.235: Removing logfile tree (if exists)
2.237: Pre-allocating directories in logfile tree
2.237: Pre-allocating files in logfile tree
2.260: datafiles populated: 1000 files, avg. dir. width = 1024, avg. dir.
     depth = 1.0, 0 leafdirs, 10000.000MB total size
2.260: Removing datafiles tree (if exists)
2.262: Pre-allocating directories in datafiles tree
2.262: Pre-allocating files in datafiles tree
27.390: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
27.390: Population and pre-allocation of filesets completed
27.403: Dropping system caches...
28.812: Running benchmark for 3600 seconds...
28.812: Attempting to create fileset more than once, ignoring
```

```
28.851: Starting 200 shadow instances
28.966: Starting 10 dbwr instances
28.972: Starting 1 lgwr instances
30.539: Running...
3644.615: Run took 3600 seconds...
3653.763: Per-Operation Breakdown
random-rate          0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
shadow-post-dbwr     32772947ops     9068ops/s   0.0mb/s   11.050ms/op
    [0.006ms - 734.869ms]
shadow-post-lg       32772956ops     9068ops/s   0.0mb/s    0.039ms/op
    [0.001ms - 815.069ms]
shadowhog            32772994ops     9068ops/s   0.0mb/s    1.554ms/op
    [0.150ms - 907.004ms]
shadowread           32798635ops     9075ops/s  17.7mb/s    9.387ms/op
    [0.001ms - 1737.397ms]
dbwr-aiowait         327720ops       91ops/s   0.0mb/s    9.244ms/op
    [0.001ms - 687.413ms]
dbwr-block           327727ops       91ops/s   0.0mb/s   91.007ms/op
    [0.001ms - 610.922ms]
dbwr-hog             327727ops       91ops/s   0.0mb/s    0.065ms/op
    [0.007ms - 235.675ms]
dbwrite-a            32773980ops     9068ops/s  17.7mb/s    0.025ms/op
    [0.000ms - 304.879ms]
lg-block             10241ops        3ops/s   0.0mb/s  352.800ms/op
    [103.666ms - 1025.298ms]
lg-aiowait           10242ops        3ops/s   0.0mb/s    0.001ms/op
    [0.001ms - 2.008ms]
lg-write             10243ops        3ops/s   0.7mb/s    0.043ms/op
    [0.001ms - 76.668ms]
3653.763: IO Summary: 65920820 ops 18239.825 ops/s 9075/9071 rd/wr  36.1
    mb/s 4.729ms/op
3653.773: Shutting down processes
```

D.3.14 *oltp-direct.f* server output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 173MB of shared memory
0.013: OLTP Direct I/O Version 3.0 personality successfully loaded
1.337: Populating and pre-allocating filesets
1.353: logfile populated: 1 files, avg. dir. width = 1024, avg. dir.
    depth = 0.0, 0 leafdirs, 10.000MB total size
1.353: Removing logfile tree (if exists)
1.354: Pre-allocating directories in logfile tree
1.355: Pre-allocating files in logfile tree
1.377: datafiles populated: 1000 files, avg. dir. width = 1024, avg. dir.
     depth = 1.0, 0 leafdirs, 10000.000MB total size
1.377: Removing datafiles tree (if exists)
1.379: Pre-allocating directories in datafiles tree
1.379: Pre-allocating files in datafiles tree
28.320: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
28.321: Population and pre-allocation of filesets completed
28.339: Dropping system caches...
29.609: Running benchmark for 3600 seconds...
29.609: Attempting to create fileset more than once, ignoring
```

```
29.668: Starting 200 shadow instances
29.815: Starting 10 dbwr instances
29.818: Starting 1 lgwr instances
31.260: Running...
3631.703: Run took 3600 seconds...
3636.942: Per-Operation Breakdown
random-rate          0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
shadow-post-dbwr     39112104ops    10863ops/s   0.0mb/s    0.196ms/op
    [0.009ms - 1871.941ms]
shadow-post-lg       39112105ops    10863ops/s   0.0mb/s    0.017ms/op
    [0.001ms - 313.872ms]
shadowhog            39112125ops    10863ops/s   0.0mb/s    0.574ms/op
    [0.150ms - 567.456ms]
shadowread           39137902ops    10870ops/s  42.4mb/s   17.595ms/op
    [0.004ms - 850.445ms]
dbwr-aiowait         391120ops      109ops/s   0.0mb/s    0.009ms/op
    [0.001ms - 87.714ms]
dbwr-block           391120ops      109ops/s   0.0mb/s   90.593ms/op
    [0.001ms - 1902.611ms]
dbwr-hog             391130ops      109ops/s   0.0mb/s    0.025ms/op
    [0.007ms - 34.140ms]
dbwrite-a            39114280ops    10863ops/s  42.4mb/s    0.010ms/op
    [0.000ms - 496.369ms]
lg-block             12222ops        3ops/s   0.0mb/s  294.544ms/op
    [187.306ms - 2045.680ms]
lg-aiowait           12223ops        3ops/s   0.0mb/s    0.000ms/op
    [0.000ms - 0.162ms]
lg-write             12224ops        3ops/s   0.8mb/s    0.013ms/op
    [0.001ms - 28.804ms]
3636.943: IO Summary: 78667749 ops 21848.935 ops/s 10870/10867 rd/wr
    85.7mb/s 8.759ms/op
3636.943: Shutting down processes
```

D.4 Zynq-7000 Platform Filebench Outputs

The following show raw outputs from Filebench benchmarks on the Zynq-7000 platform. Information about the performance metrics collected by Filebench during benchmarking can be found at [157].

D.4.1 *webserver.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.051: Web-server Version 3.1 personality successfully loaded
165.144: Populating and pre-allocating filesets
165.146: logfiles populated: 1 files, avg. dir. width = 20, avg. dir.
    depth = 0.0, 0 leafdirs, 0.002MB total size
165.146: Removing logfiles tree (if exists)
165.161: Pre-allocating directories in logfiles tree
165.163: Pre-allocating files in logfiles tree
170.994: bigfileset populated: 600000 files, avg. dir. width = 20, avg.
    dir. depth = 4.4, 0 leafdirs, 9392.595MB total size
170.994: Removing bigfileset tree (if exists)
171.008: Pre-allocating directories in bigfileset tree
182.315: Pre-allocating files in bigfileset tree
964.032: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
964.032: Population and pre-allocation of filesets completed
964.102: Dropping system caches...
967.743: Running benchmark for 3600 seconds...
967.744: Attempting to create fileset more than once, ignoring
967.836: Starting 1 filereader instances
970.391: Running...
4571.147: Run took 3600 seconds...
4571.791: Per-Operation Breakdown
appendlog            719177ops      200ops/s   1.6mb/s    2.258ms/op
    [0.001ms - 291.721ms]
closefile10          719127ops      200ops/s   0.0mb/s    0.121ms/op
    [0.007ms - 102.459ms]
readfile10           719127ops      200ops/s   3.1mb/s    1.812ms/op
    [0.052ms - 320.452ms]
openfile10           719131ops      200ops/s   0.0mb/s   10.034ms/op
    [0.092ms - 236.751ms]
closefile9           719135ops      200ops/s   0.0mb/s    0.122ms/op
    [0.008ms - 260.299ms]
readfile9            719136ops      200ops/s   3.1mb/s    1.820ms/op
    [0.042ms - 344.725ms]
openfile9            719137ops      200ops/s   0.0mb/s   10.039ms/op
    [0.082ms - 359.705ms]
closefile8           719138ops      200ops/s   0.0mb/s    0.122ms/op
    [0.008ms - 109.057ms]
readfile8            719138ops      200ops/s   3.1mb/s    1.828ms/op
    [0.047ms - 372.241ms]
openfile8            719142ops      200ops/s   0.0mb/s   10.060ms/op
    [0.085ms - 341.668ms]
closefile7           719142ops      200ops/s   0.0mb/s    0.124ms/op
    [0.008ms - 102.426ms]
```

```
readfile7           719142ops      200ops/s   3.1mb/s    1.826ms/op
    [0.054ms - 350.941ms]
openfile7           719146ops      200ops/s   0.0mb/s   10.060ms/op
    [0.090ms - 393.948ms]
closefile6          719146ops      200ops/s   0.0mb/s    0.125ms/op
    [0.007ms - 103.117ms]
readfile6           719146ops      200ops/s   3.1mb/s    1.836ms/op
    [0.043ms - 385.350ms]
openfile6           719151ops      200ops/s   0.0mb/s   10.090ms/op
    [0.091ms - 345.379ms]
closefile5          719151ops      200ops/s   0.0mb/s    0.125ms/op
    [0.007ms - 113.420ms]
readfile5           719151ops      200ops/s   3.1mb/s    1.833ms/op
    [0.213ms - 400.240ms]
openfile5           719152ops      200ops/s   0.0mb/s   10.084ms/op
    [0.088ms - 362.332ms]
closefile4          719153ops      200ops/s   0.0mb/s    0.125ms/op
    [0.008ms - 114.230ms]
readfile4           719154ops      200ops/s   3.1mb/s    1.850ms/op
    [0.044ms - 378.009ms]
openfile4           719160ops      200ops/s   0.0mb/s   10.145ms/op
    [0.088ms - 237.855ms]
closefile3          719161ops      200ops/s   0.0mb/s    0.125ms/op
    [0.007ms - 116.818ms]
readfile3           719163ops      200ops/s   3.1mb/s    1.849ms/op
    [0.227ms - 311.343ms]
openfile3           719166ops      200ops/s   0.0mb/s   10.193ms/op
    [0.088ms - 316.456ms]
closefile2          719169ops      200ops/s   0.0mb/s    0.127ms/op
    [0.007ms - 112.648ms]
readfile2           719169ops      200ops/s   3.1mb/s    1.857ms/op
    [0.043ms - 381.955ms]
openfile2           719171ops      200ops/s   0.0mb/s   10.409ms/op
    [0.090ms - 332.945ms]
closefile1          719172ops      200ops/s   0.0mb/s    0.135ms/op
    [0.007ms - 108.097ms]
readfile1           719172ops      200ops/s   3.1mb/s    1.926ms/op
    [0.040ms - 333.504ms]
openfile1           719177ops      200ops/s   0.0mb/s   10.742ms/op
    [0.089ms - 320.039ms]
4571.791: IO Summary: 22293702 ops 6191.228 ops/s 1997/200 rd/wr  32.8mb/
    s 3.994ms/op
4571.791: Shutting down processes
```

D.4.2 *webserver-direct.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.037: Web-server Direct I/O Version 3.1 personality successfully loaded
149.450: Populating and pre-allocating filesets
149.517: logfiles populated: 1 files, avg. dir. width = 20, avg. dir.
    depth = 0.0, 0 leafdirs, 0.004MB total size
149.517: Removing logfiles tree (if exists)
149.532: Pre-allocating directories in logfiles tree
149.533: Pre-allocating files in logfiles tree
```

```
155.603: bigfileset populated: 600000 files, avg. dir. width = 20, avg.
    dir. depth = 4.4, 0 leafdirs, 10577.414MB total size
155.603: Removing bigfileset tree (if exists)
155.618: Pre-allocating directories in bigfileset tree
168.000: Pre-allocating files in bigfileset tree
957.712: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
957.714: Population and pre-allocation of filesets completed
957.792: Dropping system caches...
961.448: Running benchmark for 3600 seconds...
961.449: Attempting to create fileset more than once, ignoring
961.593: Starting 1 filereader instances
963.492: Running...
4564.079: Run took 3600 seconds...
4564.623: Per-Operation Breakdown
appendlog           325802ops       90ops/s   0.9mb/s  236.832ms/op
    [0.001ms - 512.378ms]
closefile10         325754ops       90ops/s   0.0mb/s    0.046ms/op
    [0.008ms - 90.847ms]
readfile10          325758ops       90ops/s   1.6mb/s   24.489ms/op
    [1.523ms - 211.093ms]
openfile10          325758ops       90ops/s   0.0mb/s    0.615ms/op
    [0.088ms - 147.742ms]
closefile9          325758ops       90ops/s   0.0mb/s    0.043ms/op
    [0.009ms - 44.079ms]
readfile9           325759ops       90ops/s   1.6mb/s   24.296ms/op
    [1.564ms - 189.662ms]
openfile9           325759ops       90ops/s   0.0mb/s    0.616ms/op
    [0.086ms - 233.239ms]
closefile8          325759ops       90ops/s   0.0mb/s    0.040ms/op
    [0.008ms - 71.803ms]
readfile8           325762ops       90ops/s   1.6mb/s   24.332ms/op
    [1.440ms - 216.914ms]
openfile8           325762ops       90ops/s   0.0mb/s    0.605ms/op
    [0.088ms - 195.162ms]
closefile7          325762ops       90ops/s   0.0mb/s    0.036ms/op
    [0.008ms - 25.369ms]
readfile7           325765ops       90ops/s   1.6mb/s   24.165ms/op
    [1.497ms - 194.801ms]
openfile7           325765ops       90ops/s   0.0mb/s    0.621ms/op
    [0.087ms - 152.622ms]
closefile6          325765ops       90ops/s   0.0mb/s    0.035ms/op
    [0.008ms - 80.719ms]
readfile6           325765ops       90ops/s   1.6mb/s   24.253ms/op
    [1.336ms - 177.146ms]
openfile6           325766ops       90ops/s   0.0mb/s    0.610ms/op
    [0.085ms - 153.066ms]
closefile5          325766ops       90ops/s   0.0mb/s    0.031ms/op
    [0.008ms - 19.090ms]
readfile5           325769ops       90ops/s   1.6mb/s   24.180ms/op
    [1.141ms - 184.123ms]
openfile5           325770ops       90ops/s   0.0mb/s    0.635ms/op
    [0.087ms - 151.682ms]
closefile4          325770ops       90ops/s   0.0mb/s    0.030ms/op
    [0.008ms - 29.301ms]
readfile4           325773ops       90ops/s   1.6mb/s   24.258ms/op
    [1.419ms - 259.298ms]
```

```
openfile4            325773ops       90ops/s   0.0mb/s    0.620ms/op
    [0.089ms - 185.917ms]
closefile3           325773ops       90ops/s   0.0mb/s    0.027ms/op
    [0.008ms - 24.811ms]
readfile3            325774ops       90ops/s   1.6mb/s   23.959ms/op
    [1.229ms - 244.397ms]
openfile3            325774ops       90ops/s   0.0mb/s    0.667ms/op
    [0.088ms - 176.462ms]
closefile2           325774ops       90ops/s   0.0mb/s    0.026ms/op
    [0.008ms - 23.382ms]
readfile2            325777ops       90ops/s   1.6mb/s   24.471ms/op
    [1.463ms - 166.902ms]
openfile2            325777ops       90ops/s   0.0mb/s    0.631ms/op
    [0.085ms - 189.338ms]
closefile1           325777ops       90ops/s   0.0mb/s    0.024ms/op
    [0.008ms - 32.390ms]
readfile1            325780ops       90ops/s   1.6mb/s   23.601ms/op
    [1.291ms - 202.117ms]
openfile1            325780ops       90ops/s   0.0mb/s    0.819ms/op
    [0.090ms - 173.200ms]
4564.623: IO Summary: 10098826 ops 2804.774 ops/s 905/90 rd/wr  16.8mb/s
    15.666ms/op
4564.623: Shutting down processes
```

D.4.3 *fileserver.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.017: File-server Version 3.0 personality successfully loaded
18.140: Populating and pre-allocating filesets
18.812: bigfileset populated: 80000 files, avg. dir. width = 20, avg. dir
    . depth = 3.8, 0 leafdirs, 10019.570MB total size
18.812: Removing bigfileset tree (if exists)
18.826: Pre-allocating directories in bigfileset tree
19.870: Pre-allocating files in bigfileset tree
175.321: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
175.321: Population and pre-allocation of filesets completed
175.341: Dropping system caches...
177.330: Running benchmark for 3600 seconds...
177.330: Attempting to create fileset more than once, ignoring
177.378: Starting 1 filereader instances
178.874: Running...
3779.937: Run took 3600 seconds...
3780.101: Per-Operation Breakdown
statfile1            649229ops      180ops/s   0.0mb/s    4.446ms/op
    [0.017ms - 278.144ms]
deletefile1          649234ops      180ops/s   0.0mb/s    7.469ms/op
    [0.223ms - 285.294ms]
closefile3           649234ops      180ops/s   0.0mb/s    0.299ms/op
    [0.009ms - 247.082ms]
readfile1            649242ops      180ops/s  23.8mb/s   10.394ms/op
    [0.243ms - 800.115ms]
openfile2            649244ops      180ops/s   0.0mb/s   10.586ms/op
    [0.043ms - 298.897ms]
```

```
closefile2           649247ops      180ops/s   0.0mb/s    0.264ms/op
    [0.008ms - 136.319ms]
appendfilerand1      649252ops      180ops/s   1.4mb/s   18.289ms/op
    [0.001ms - 974.933ms]
openfile1            649256ops      180ops/s   0.0mb/s   11.991ms/op
    [0.041ms - 372.531ms]
closefile1           649257ops      180ops/s   0.0mb/s    0.369ms/op
    [0.008ms - 256.224ms]
wrtfile1             649266ops      180ops/s  22.6mb/s   96.870ms/op
    [0.085ms - 3158.208ms]
createfile1          649275ops      180ops/s   0.0mb/s    7.764ms/op
    [0.159ms - 321.064ms]
3780.102: IO Summary: 7141736 ops 1983.231 ops/s 180/361 rd/wr  47.8mb/s
    15.340ms/op
3780.103: Shutting down processes
```

D.4.4 *fileserver-direct.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.087: File-server Direct I/O Version 3.0 personality successfully loaded
41.577: Populating and pre-allocating filesets
42.332: bigfileset populated: 80000 files, avg. dir. width = 20, avg. dir
    . depth = 3.8, 0 leafdirs, 10175.633MB total size
42.332: Removing bigfileset tree (if exists)
42.347: Pre-allocating directories in bigfileset tree
43.401: Pre-allocating files in bigfileset tree
199.844: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
199.845: Population and pre-allocation of filesets completed
199.866: Dropping system caches...
201.844: Running benchmark for 3600 seconds...
201.844: Attempting to create fileset more than once, ignoring
201.983: Starting 1 filereader instances
203.802: Running...
3804.670: Run took 3600 seconds...
3804.727: Per-Operation Breakdown
statfile1            1021327ops      284ops/s   0.0mb/s    0.105ms/op
    [0.018ms - 205.571ms]
deletefile1          1021330ops      284ops/s   0.0mb/s    1.047ms/op
    [0.199ms - 540.199ms]
closefile3           1021332ops      284ops/s   0.0mb/s    0.063ms/op
    [0.008ms - 220.346ms]
readfile1            1021336ops      284ops/s  38.8mb/s   38.947ms/op
    [1.311ms - 449.529ms]
openfile2            1021339ops      284ops/s   0.0mb/s    3.023ms/op
    [0.036ms - 259.390ms]
closefile2           1021349ops      284ops/s   0.0mb/s    0.081ms/op
    [0.008ms - 154.359ms]
appendfilerand1      1021355ops      284ops/s   2.8mb/s   17.700ms/op
    [0.001ms - 553.886ms]
openfile1            1021357ops      284ops/s   0.0mb/s    2.935ms/op
    [0.037ms - 317.778ms]
closefile1           1021363ops      284ops/s   0.0mb/s    0.088ms/op
    [0.008ms - 173.202ms]
```

```
wrtfile1             1021368ops      284ops/s  36.1mb/s   19.627ms/op
    [0.386ms - 907.520ms]
createfile1          1021377ops      284ops/s   0.0mb/s    3.775ms/op
    [0.141ms - 316.922ms]
3804.727: IO Summary: 11234833 ops 3120.036 ops/s 284/567 rd/wr  77.7mb/s
     7.945ms/op
3804.727: Shutting down processes
```

D.4.5 *varmail.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.298: Varmail Version 3.0 personality successfully loaded
169.632: Populating and pre-allocating filesets
175.896: bigfileset populated: 600000 files, avg. dir. width = 1000000,
    avg. dir. depth = 1.0, 0 leafdirs, 9386.366MB total size
175.896: Removing bigfileset tree (if exists)
175.910: Pre-allocating directories in bigfileset tree
175.911: Pre-allocating files in bigfileset tree
748.826: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
748.827: Population and pre-allocation of filesets completed
748.907: Dropping system caches...
752.589: Running benchmark for 3600 seconds...
752.589: Attempting to create fileset more than once, ignoring
752.715: Starting 1 filereader instances
754.574: Running...
4355.370: Run took 3600 seconds...
4355.471: Per-Operation Breakdown
closefile4           1058630ops      294ops/s   0.0mb/s    0.060ms/op
    [0.009ms - 26.741ms]
readfile4            1058630ops      294ops/s   4.4mb/s    1.871ms/op
    [0.037ms - 97.251ms]
openfile4            1058633ops      294ops/s   0.0mb/s    5.233ms/op
    [0.096ms - 163.245ms]
closefile3           1058633ops      294ops/s   0.0mb/s    0.069ms/op
    [0.009ms - 89.910ms]
fsyncfile3           1058636ops      294ops/s   0.0mb/s   11.648ms/op
    [0.018ms - 96.541ms]
appendfilerand3      1058638ops      294ops/s   2.3mb/s    1.102ms/op
    [0.001ms - 48.877ms]
readfile3            1058640ops      294ops/s   4.4mb/s    1.985ms/op
    [0.033ms - 86.061ms]
openfile3            1058643ops      294ops/s   0.0mb/s    5.237ms/op
    [0.099ms - 168.195ms]
closefile2           1058643ops      294ops/s   0.0mb/s    0.077ms/op
    [0.009ms - 56.629ms]
fsyncfile2           1058643ops      294ops/s   0.0mb/s   11.695ms/op
    [0.834ms - 85.658ms]
appendfilerand2      1058643ops      294ops/s   2.3mb/s    0.923ms/op
    [0.001ms - 62.824ms]
createfile2          1058643ops      294ops/s   0.0mb/s    6.217ms/op
    [0.163ms - 163.535ms]
deletefile1          1058646ops      294ops/s   0.0mb/s    6.596ms/op
    [0.248ms - 111.562ms]
```

```
4355.471: IO Summary: 13762301 ops 3822.017 ops/s 588/588 rd/wr  13.4mb/s
     4.055ms/op
4355.472: Shutting down processes
```

D.4.6 *varmail-direct.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.043: Varmail Direct I/O Version 3.0 personality successfully loaded
3.423: Populating and pre-allocating filesets
9.370: bigfileset populated: 600000 files, avg. dir. width = 1000000, avg
    . dir. depth = 1.0, 0 leafdirs, 10571.594MB total size
9.370: Removing bigfileset tree (if exists)
9.385: Pre-allocating directories in bigfileset tree
9.386: Pre-allocating files in bigfileset tree
579.708: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
579.709: Population and pre-allocation of filesets completed
579.739: Dropping system caches...
583.468: Running benchmark for 3600 seconds...
583.468: Attempting to create fileset more than once, ignoring
583.581: Starting 1 filereader instances
585.572: Running...
4186.319: Run took 3600 seconds...
4186.538: Per-Operation Breakdown
closefile4          797500ops      221ops/s   0.0mb/s    0.066ms/op
    [0.010ms - 43.654ms]
readfile4           797500ops      221ops/s   3.9mb/s   12.102ms/op
    [0.674ms - 131.383ms]
openfile4           797501ops      221ops/s   0.0mb/s    2.469ms/op
    [0.076ms - 128.719ms]
closefile3          797501ops      221ops/s   0.0mb/s    0.049ms/op
    [0.008ms - 51.076ms]
fsyncfile3          797502ops      221ops/s   0.0mb/s    7.448ms/op
    [0.011ms - 117.516ms]
appendfilerand3     797505ops      221ops/s   2.2mb/s    7.738ms/op
    [0.001ms - 139.764ms]
readfile3           797508ops      221ops/s   3.9mb/s   12.434ms/op
    [0.927ms - 103.688ms]
openfile3           797510ops      221ops/s   0.0mb/s    2.528ms/op
    [0.075ms - 122.642ms]
closefile2          797510ops      221ops/s   0.0mb/s    0.054ms/op
    [0.008ms - 50.458ms]
fsyncfile2          797511ops      221ops/s   0.0mb/s    7.733ms/op
    [0.012ms - 124.796ms]
appendfilerand2     797512ops      221ops/s   2.2mb/s    6.866ms/op
    [0.001ms - 100.229ms]
createfile2         797516ops      221ops/s   0.0mb/s    3.263ms/op
    [0.168ms - 177.171ms]
deletefile1         797516ops      221ops/s   0.0mb/s    3.326ms/op
    [0.222ms - 86.478ms]
4186.538: IO Summary: 10367592 ops 2879.212 ops/s 443/443 rd/wr  12.1mb/s
     5.083ms/op
4186.538: Shutting down processes
```

D.4.7 *videoserver.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.028: Video Server Version 3.0 personality successfully loaded
5.464: Populating and pre-allocating filesets
5.465: passivevids populated: 28 files, avg. dir. width = 20, avg. dir.
    depth = 1.1, 0 leafdirs, 286720.000MB total size
5.466: Removing passivevids tree (if exists)
5.480: Pre-allocating directories in passivevids tree
5.482: Pre-allocating files in passivevids tree
5.500: activevids populated: 8 files, avg. dir. width = 4, avg. dir.
    depth = 1.5, 0 leafdirs, 81920.000MB total size
5.501: Removing activevids tree (if exists)
5.539: Pre-allocating directories in activevids tree
5.546: Pre-allocating files in activevids tree
2529.270: Waiting for pre-allocation to finish (in case of a parallel pre
    -allocation)
2530.791: Population and pre-allocation of filesets completed
2530.855: Dropping system caches...
2532.995: Running benchmark for 3600 seconds...
2532.995: Attempting to create fileset more than once, ignoring
2533.086: Starting 1 vidreaders instances
2533.118: Starting 1 vidwriter instances
2534.975: Running...
6135.898: Run took 3600 seconds...
6136.149: Per-Operation Breakdown
serverlimit          1377604ops      383ops/s   0.0mb/s  123.624ms/op
    [0.001ms - 9973.772ms]
vidreader            1377748ops      383ops/s  95.6mb/s    5.845ms/op
    [0.521ms - 281.093ms]
replaceinterval      10ops        0ops/s   0.0mb/s 10002.136ms/op
    [10000.253ms - 10017.513ms]
wrtclose             10ops        0ops/s   0.0mb/s   16.386ms/op [0.018ms
     - 163.659ms]
newvid               10ops        0ops/s  28.4mb/s 329394.482ms/op
    [212547.965ms - 377270.444ms]
wrtopen              11ops        0ops/s   0.0mb/s    0.623ms/op [0.207ms
     - 1.395ms]
vidremover           11ops        0ops/s   0.0mb/s  666.753ms/op [431.492
    ms - 814.425ms]
6136.149: IO Summary: 1377790 ops 382.614 ops/s 383/0 rd/wr 124.1mb/s
    8.241ms/op
6136.149: Shutting down processes
```

D.4.8 *videoserver-direct.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.074: Video Server Direct I/O Version 3.0 personality successfully
    loaded
170.621: Populating and pre-allocating filesets
170.685: passivevids populated: 28 files, avg. dir. width = 20, avg. dir.
     depth = 1.1, 0 leafdirs, 286720.000MB total size
```

```
170.685: Removing passivevids tree (if exists)
170.700: Pre-allocating directories in passivevids tree
170.701: Pre-allocating files in passivevids tree
170.729: activevids populated: 8 files, avg. dir. width = 4, avg. dir.
    depth = 1.5, 0 leafdirs, 81920.000MB total size
170.730: Removing activevids tree (if exists)
170.763: Pre-allocating directories in activevids tree
170.770: Pre-allocating files in activevids tree
2797.199: Waiting for pre-allocation to finish (in case of a parallel pre
    -allocation)
2799.229: Population and pre-allocation of filesets completed
2799.293: Dropping system caches...
2801.497: Running benchmark for 3600 seconds...
2801.497: Attempting to create fileset more than once, ignoring
2801.582: Starting 1 vidreaders instances
2801.618: Starting 1 vidwriter instances
2803.215: Running...
6403.970: Run took 3600 seconds...
6403.973: Per-Operation Breakdown
serverlimit          1379477ops      383ops/s   0.0mb/s  124.265ms/op
    [0.001ms - 9848.245ms]
vidreader            1379620ops      383ops/s  95.8mb/s    2.678ms/op
    [1.191ms - 154.055ms]
replaceinterval      35ops        0ops/s   0.0mb/s 10000.130ms/op
    [10000.098ms - 10000.724ms]
wrtclose             35ops        0ops/s   0.0mb/s    0.020ms/op [0.016ms
     - 0.032ms]
newvid               35ops        0ops/s  99.5mb/s 90753.528ms/op
    [82972.013ms - 97714.076ms]
wrtopen              36ops        0ops/s   0.0mb/s    0.306ms/op [0.154ms
     - 1.561ms]
vidremover           36ops        0ops/s   0.0mb/s   53.626ms/op [3.361ms
     - 730.381ms]
6403.973: IO Summary: 1379762 ops 383.187 ops/s 383/0 rd/wr 195.3mb/s
    4.981ms/op
6403.974: Shutting down processes
```

D.4.9 *netsfs.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.024: NetworkFileServer Version 1.0 personality successfully loaded
73.931: Populating and pre-allocating filesets
75.859: bigfileset populated: 400000 files, avg. dir. width = 20, avg.
    dir. depth = 4.3, 0 leafdirs, 10391.271MB total size
75.859: Removing bigfileset tree (if exists)
75.873: Pre-allocating directories in bigfileset tree
82.138: Pre-allocating files in bigfileset tree
445.954: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
445.954: Population and pre-allocation of filesets completed
445.977: Dropping system caches...
448.990: Running benchmark for 3600 seconds...
448.990: Attempting to create fileset more than once, ignoring
449.076: Starting 1 netclient instances
450.908: Running...
```

```
4051.255: Run took 3600 seconds...
4051.256: Per-Operation Breakdown
ratecontrol          35990ops       10ops/s   0.0mb/s   94.580ms/op
    [0.001ms - 1035.751ms]
statfile1            35991ops       10ops/s   0.0mb/s    0.021ms/op
    [0.015ms - 0.186ms]
appnd1.closefile6    107973ops       30ops/s   0.0mb/s    0.011ms/op
    [0.007ms - 0.199ms]
appnd1.appendfilerand6 107973ops       30ops/s   0.5mb/s    0.254ms/op
    [0.001ms - 2.582ms]
appnd1.openfile6     107973ops       30ops/s   0.0mb/s    0.042ms/op
    [0.030ms - 2.983ms]
appnd1               0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
rmw1.deletefile1     215946ops       60ops/s   0.0mb/s    0.176ms/op
    [0.147ms - 3.490ms]
rmw1.closefile2      215946ops       60ops/s   0.0mb/s    0.005ms/op
    [0.003ms - 0.607ms]
rmw1.closefile1      215946ops       60ops/s   0.0mb/s    0.012ms/op
    [0.009ms - 0.293ms]
rmw1.writefile2      215946ops       60ops/s   0.1mb/s    0.088ms/op
    [0.080ms - 0.675ms]
rmw1.newfile2        215946ops       60ops/s   0.0mb/s    0.152ms/op
    [0.134ms - 8.296ms]
rmw1.readfile1       215946ops       60ops/s   0.6mb/s    0.045ms/op
    [0.017ms - 1.214ms]
rmw1.openfile1       215946ops       60ops/s   0.0mb/s    0.042ms/op
    [0.030ms - 0.385ms]
rmw1                 0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
launch1.closefile5   35991ops       10ops/s   0.0mb/s    0.005ms/op
    [0.003ms - 0.198ms]
launch1.closefile4   35991ops       10ops/s   0.0mb/s    0.010ms/op
    [0.007ms - 0.188ms]
launch1.readfile5    35991ops       10ops/s   0.5mb/s    0.142ms/op
    [0.017ms - 0.934ms]
launch1.openfile5    35991ops       10ops/s   0.0mb/s    0.042ms/op
    [0.030ms - 2.234ms]
launch1.closefile3   35991ops       10ops/s   0.0mb/s    0.011ms/op
    [0.008ms - 0.188ms]
launch1.readfile4    35991ops       10ops/s   0.5mb/s    0.144ms/op
    [0.016ms - 1.018ms]
launch1.openfile4    35991ops       10ops/s   0.0mb/s    0.044ms/op
    [0.031ms - 2.570ms]
launch1.readfile3    35991ops       10ops/s   0.5mb/s    0.170ms/op
    [0.019ms - 1.623ms]
launch1.openfile3    35991ops       10ops/s   0.0mb/s    0.037ms/op
    [0.029ms - 7.752ms]
launch1              0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
4051.256: IO Summary: 2195451 ops 609.789 ops/s 90/90 rd/wr   2.7mb/s
    0.076ms/op
4051.256: Shutting down processes
```

D.4.10 *netsfs-direct.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.116: NetworkFileServer Version 1.0 personality successfully loaded
22.714: Populating and pre-allocating filesets
24.630: bigfileset populated: 400000 files, avg. dir. width = 20, avg.
    dir. depth = 4.3, 0 leafdirs, 10359.824MB total size
24.631: Removing bigfileset tree (if exists)
24.645: Pre-allocating directories in bigfileset tree
31.169: Pre-allocating files in bigfileset tree
392.725: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
392.725: Population and pre-allocation of filesets completed
392.747: Dropping system caches...
396.074: Running benchmark for 3600 seconds...
396.074: Attempting to create fileset more than once, ignoring
396.176: Starting 1 netclient instances
397.639: Running...
3998.011: Run took 3600 seconds...
3998.013: Per-Operation Breakdown
ratecontrol          35985ops       10ops/s   0.0mb/s   85.487ms/op
    [0.002ms - 937.406ms]
statfile1            35985ops       10ops/s   0.0mb/s    0.020ms/op
    [0.016ms - 0.348ms]
appnd1.closefile6    107955ops       30ops/s   0.0mb/s    0.012ms/op
    [0.008ms - 0.190ms]
appnd1.appendfilerand6 107955ops       30ops/s   0.6mb/s    0.450ms/op
    [0.001ms - 4.497ms]
appnd1.openfile6     107955ops       30ops/s   0.0mb/s    0.042ms/op
    [0.030ms - 2.486ms]
appnd1               0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
rmw1.deletefile1     215911ops       60ops/s   0.0mb/s    0.194ms/op
    [0.157ms - 1.015ms]
rmw1.closefile2      215912ops       60ops/s   0.0mb/s    0.005ms/op
    [0.003ms - 0.568ms]
rmw1.closefile1      215912ops       60ops/s   0.0mb/s    0.013ms/op
    [0.009ms - 0.225ms]
rmw1.writefile2      215912ops       60ops/s   0.2mb/s    0.090ms/op
    [0.080ms - 0.871ms]
rmw1.newfile2        215912ops       60ops/s   0.0mb/s    0.156ms/op
    [0.138ms - 1.036ms]
rmw1.readfile1       215912ops       60ops/s   0.8mb/s    0.228ms/op
    [0.046ms - 7.658ms]
rmw1.openfile1       215912ops       60ops/s   0.0mb/s    0.042ms/op
    [0.035ms - 2.961ms]
rmw1                 0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
launch1.closefile5   35986ops       10ops/s   0.0mb/s    0.012ms/op
    [0.004ms - 0.215ms]
launch1.closefile4   35986ops       10ops/s   0.0mb/s    0.012ms/op
    [0.009ms - 0.165ms]
launch1.readfile5    35986ops       10ops/s   0.6mb/s    1.803ms/op
    [0.206ms - 9.888ms]
launch1.openfile5    35986ops       10ops/s   0.0mb/s    0.042ms/op
    [0.036ms - 2.873ms]
launch1.closefile3   35986ops       10ops/s   0.0mb/s    0.012ms/op
    [0.009ms - 0.189ms]
```

```
launch1.readfile4    35986ops       10ops/s   0.6mb/s    1.850ms/op
    [0.223ms - 11.478ms]
launch1.openfile4    35986ops       10ops/s   0.0mb/s    0.045ms/op
    [0.039ms - 2.348ms]
launch1.readfile3    35986ops       10ops/s   0.6mb/s    3.992ms/op
    [0.230ms - 13.763ms]
launch1.openfile3    35986ops       10ops/s   0.0mb/s    0.037ms/op
    [0.029ms - 9.443ms]
launch1              0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
3998.013: IO Summary: 2195107 ops 609.689 ops/s 90/90 rd/wr   3.4mb/s
    0.225ms/op
3998.013: Shutting down processes
```

D.4.11 *webproxy.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.038: Web proxy-server Version 3.0 personality successfully loaded
149.233: Populating and pre-allocating filesets
152.747: bigfileset populated: 600000 files, avg. dir. width = 1000000,
    avg. dir. depth = 1.0, 0 leafdirs, 9375.000MB total size
152.748: Removing bigfileset tree (if exists)
152.762: Pre-allocating directories in bigfileset tree
152.763: Pre-allocating files in bigfileset tree
697.886: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
697.886: Population and pre-allocation of filesets completed
697.909: Dropping system caches...
701.583: Running benchmark for 3600 seconds...
701.583: Attempting to create fileset more than once, ignoring
701.735: Starting 1 proxycache instances
704.565: Running...
4305.433: Run took 3600 seconds...
4306.972: Per-Operation Breakdown
limit                0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
closefile6           459359ops      128ops/s   0.0mb/s    0.085ms/op
    [0.009ms - 294.265ms]
readfile6            459359ops      128ops/s   1.6mb/s    1.588ms/op
    [0.046ms - 4611.980ms]
openfile6            459366ops      128ops/s   0.0mb/s   86.825ms/op
    [0.034ms - 5217.863ms]
closefile5           459367ops      128ops/s   0.0mb/s    0.088ms/op
    [0.009ms - 138.276ms]
readfile5            459367ops      128ops/s   1.6mb/s    1.584ms/op
    [0.027ms - 386.483ms]
openfile5            459382ops      128ops/s   0.0mb/s   87.734ms/op
    [0.033ms - 5459.838ms]
closefile4           459382ops      128ops/s   0.0mb/s    0.084ms/op
    [0.008ms - 99.945ms]
readfile4            459382ops      128ops/s   1.6mb/s    1.619ms/op
    [0.040ms - 331.575ms]
openfile4            459398ops      128ops/s   0.0mb/s   87.291ms/op
    [0.033ms - 5161.888ms]
```

```
closefile3          459398ops      128ops/s   0.0mb/s    0.087ms/op
    [0.008ms - 99.435ms]
readfile3           459398ops      128ops/s   1.6mb/s    1.646ms/op
    [0.039ms - 4691.849ms]
openfile3           459414ops      128ops/s   0.0mb/s   85.768ms/op
    [0.033ms - 5200.237ms]
closefile2          459414ops      128ops/s   0.0mb/s    0.089ms/op
    [0.009ms - 119.728ms]
readfile2           459414ops      128ops/s   1.6mb/s    1.611ms/op
    [0.036ms - 3391.758ms]
openfile2           459422ops      128ops/s   0.0mb/s   83.323ms/op
    [0.037ms - 5237.164ms]
closefile1          459422ops      128ops/s   0.0mb/s    0.060ms/op
    [0.008ms - 263.612ms]
appendfilerand1     459423ops      128ops/s   1.0mb/s    5.506ms/op
    [0.001ms - 4953.901ms]
createfile1         459429ops      128ops/s   0.0mb/s   83.385ms/op
    [0.224ms - 5221.517ms]
deletefile1         459439ops      128ops/s   0.0mb/s   81.732ms/op
    [0.280ms - 2110.336ms]
4306.972: IO Summary: 8728535 ops 2423.366 ops/s 638/128 rd/wr   9.2mb/s
    32.111ms/op
4306.972: Shutting down processes
```

D.4.12 *webproxy-direct.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.754: Web proxy-server Direct I/O Version 3.0 personality successfully
    loaded
7.558: Populating and pre-allocating filesets
10.955: bigfileset populated: 600000 files, avg. dir. width = 1000000,
    avg. dir. depth = 1.0, 0 leafdirs, 9375.000MB total size
10.955: Removing bigfileset tree (if exists)
10.969: Pre-allocating directories in bigfileset tree
10.970: Pre-allocating files in bigfileset tree
558.584: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
558.584: Population and pre-allocation of filesets completed
558.607: Dropping system caches...
562.349: Running benchmark for 3600 seconds...
562.349: Attempting to create fileset more than once, ignoring
562.474: Starting 1 proxycache instances
564.989: Running...
4165.835: Run took 3600 seconds...
4166.511: Per-Operation Breakdown
limit               0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
closefile6          331853ops       92ops/s   0.0mb/s    0.227ms/op
    [0.009ms - 214.566ms]
readfile6           331854ops       92ops/s   1.3mb/s   41.954ms/op
    [1.100ms - 358.073ms]
openfile6           331860ops       92ops/s   0.0mb/s   43.176ms/op
    [0.036ms - 1122.635ms]
closefile5          331863ops       92ops/s   0.0mb/s    0.223ms/op
    [0.009ms - 155.138ms]
```

```
readfile5           331864ops       92ops/s   1.3mb/s   41.982ms/op
    [1.114ms - 360.820ms]
openfile5           331870ops       92ops/s   0.0mb/s   43.155ms/op
    [0.037ms - 1076.310ms]
closefile4          331875ops       92ops/s   0.0mb/s    0.217ms/op
    [0.009ms - 152.931ms]
readfile4           331875ops       92ops/s   1.3mb/s   41.903ms/op
    [1.083ms - 350.737ms]
openfile4           331883ops       92ops/s   0.0mb/s   43.145ms/op
    [0.038ms - 981.753ms]
closefile3          331891ops       92ops/s   0.0mb/s    0.220ms/op
    [0.009ms - 158.653ms]
readfile3           331893ops       92ops/s   1.3mb/s   41.703ms/op
    [1.078ms - 353.147ms]
openfile3           331903ops       92ops/s   0.0mb/s   42.641ms/op
    [0.040ms - 968.972ms]
closefile2          331907ops       92ops/s   0.0mb/s    0.216ms/op
    [0.009ms - 175.705ms]
readfile2           331907ops       92ops/s   1.3mb/s   41.978ms/op
    [1.123ms - 484.395ms]
openfile2           331914ops       92ops/s   0.0mb/s   44.107ms/op
    [0.037ms - 1295.449ms]
closefile1          331918ops       92ops/s   0.0mb/s    0.199ms/op
    [0.009ms - 270.207ms]
appendfilerand1     331918ops       92ops/s   0.9mb/s   20.583ms/op
    [0.001ms - 311.496ms]
createfile1         331923ops       92ops/s   0.0mb/s   44.515ms/op
    [0.169ms - 1144.994ms]
deletefile1         331920ops       92ops/s   0.0mb/s   25.695ms/op
    [0.242ms - 532.063ms]
4166.511: IO Summary: 6305891 ops 1751.174 ops/s 461/92 rd/wr   7.3mb/s
    27.255ms/op
4166.511: Shutting down processes
```

D.4.13 *oltp.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.031: OLTP Version 3.0 personality successfully loaded
2.932: Populating and pre-allocating filesets
2.933: logfile populated: 1 files, avg. dir. width = 1024, avg. dir.
    depth = 0.0, 0 leafdirs, 10.000MB total size
2.933: Removing logfile tree (if exists)
2.955: Pre-allocating directories in logfile tree
2.956: Pre-allocating files in logfile tree
3.060: datafiles populated: 1000 files, avg. dir. width = 1024, avg. dir.
     depth = 1.0, 0 leafdirs, 10000.000MB total size
3.060: Removing datafiles tree (if exists)
3.074: Pre-allocating directories in datafiles tree
3.075: Pre-allocating files in datafiles tree
149.848: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
149.848: Population and pre-allocation of filesets completed
149.870: Dropping system caches...
151.746: Running benchmark for 3600 seconds...
151.746: Attempting to create fileset more than once, ignoring
```

```
151.761: Starting 200 shadow instances
154.713: Starting 10 dbwr instances
154.967: Starting 1 lgwr instances
156.021: Running...
3765.932: Run took 3600 seconds...
3766.585: Per-Operation Breakdown
random-rate          0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
shadow-post-dbwr     3016383ops      836ops/s   0.0mb/s  121.158ms/op
    [0.205ms - 5994.334ms]
shadow-post-lg       3016461ops      836ops/s   0.0mb/s  104.515ms/op
    [0.025ms - 1939.787ms]
shadowhog            3016464ops      836ops/s   0.0mb/s    9.912ms/op
    [1.250ms - 1186.068ms]
shadowread           3042076ops      843ops/s   1.6mb/s    2.593ms/op
    [0.008ms - 1095.134ms]
dbwr-aiowait         30160ops        8ops/s   0.0mb/s    1.877ms/op
    [0.014ms - 1232.898ms]
dbwr-block           30160ops        8ops/s   0.0mb/s 1182.394ms/op
    [0.019ms - 1477.072ms]
dbwr-hog             30170ops        8ops/s   0.0mb/s    0.472ms/op
    [0.063ms - 157.480ms]
dbwrite-a            3018280ops      836ops/s   1.6mb/s    0.058ms/op
    [0.001ms - 415.538ms]
lg-block             942ops        0ops/s   0.0mb/s 3825.274ms/op
    [3242.285ms - 4472.041ms]
lg-aiowait           943ops        0ops/s   0.0mb/s    0.009ms/op [0.001
    ms - 4.000ms]
lg-write             944ops        0ops/s   0.1mb/s    0.420ms/op [0.071
    ms - 48.875ms]
3766.586: IO Summary: 6092403 ops 1687.628 ops/s 843/836 rd/wr   3.3mb/s
    1.333ms/op
3766.586: Shutting down processes
```

D.4.14 *oltp-direct.f* Zynq output

```
Filebench Version 1.5-alpha3+4kdirect
0.000: Allocated 133MB of shared memory
0.019: OLTP Direct I/O Version 3.0 personality successfully loaded
75.056: Populating and pre-allocating filesets
75.057: logfile populated: 1 files, avg. dir. width = 1024, avg. dir.
    depth = 0.0, 0 leafdirs, 10.000MB total size
75.057: Removing logfile tree (if exists)
75.071: Pre-allocating directories in logfile tree
75.072: Pre-allocating files in logfile tree
75.177: datafiles populated: 1000 files, avg. dir. width = 1024, avg. dir
    . depth = 1.0, 0 leafdirs, 10000.000MB total size
75.177: Removing datafiles tree (if exists)
75.191: Pre-allocating directories in datafiles tree
75.193: Pre-allocating files in datafiles tree
222.988: Waiting for pre-allocation to finish (in case of a parallel pre-
    allocation)
222.988: Population and pre-allocation of filesets completed
222.998: Dropping system caches...
224.883: Running benchmark for 3600 seconds...
224.883: Attempting to create fileset more than once, ignoring
```

```
224.974: Starting 200 shadow instances
227.920: Starting 10 dbwr instances
227.949: Starting 1 lgwr instances
229.275: Running...
3832.419: Run took 3600 seconds...
3833.584: Per-Operation Breakdown
random-rate          0ops        0ops/s   0.0mb/s    0.000ms/op [0.000ms
    - 0.000ms]
shadow-post-dbwr     2587485ops      718ops/s   0.0mb/s  103.565ms/op
    [0.207ms - 2521.263ms]
shadow-post-lg       2587637ops      718ops/s   0.0mb/s  123.268ms/op
    [0.009ms - 2408.915ms]
shadowhog            2587655ops      718ops/s   0.0mb/s   27.212ms/op
    [1.250ms - 1171.647ms]
shadowread           2613271ops      725ops/s   2.8mb/s   23.453ms/op
    [0.035ms - 1742.368ms]
dbwr-aiowait         25868ops        7ops/s   0.0mb/s  553.218ms/op
    [0.019ms - 3399.225ms]
dbwr-block           25869ops        7ops/s   0.0mb/s  726.232ms/op
    [0.014ms - 2400.455ms]
dbwr-hog             25878ops        7ops/s   0.0mb/s    1.173ms/op
    [0.063ms - 912.013ms]
dbwrite-a            2589080ops      719ops/s   2.8mb/s    0.244ms/op
    [0.001ms - 1126.217ms]
lg-block             808ops        0ops/s   0.0mb/s 4452.388ms/op
    [3213.371ms - 7219.245ms]
lg-aiowait           809ops        0ops/s   0.0mb/s    0.003ms/op [0.001
    ms - 0.905ms]
lg-write             810ops        0ops/s   0.1mb/s    0.208ms/op [0.072
    ms - 17.916ms]
3833.584: IO Summary: 5229838 ops 1451.466 ops/s 725/719 rd/wr   5.7mb/s
    14.576ms/op
3833.584: Shutting down processes
```

E

Additional Plots

E.1 Chapter 3 – Selected Zynq-7000 Platform CPU Frequency Scaling Results

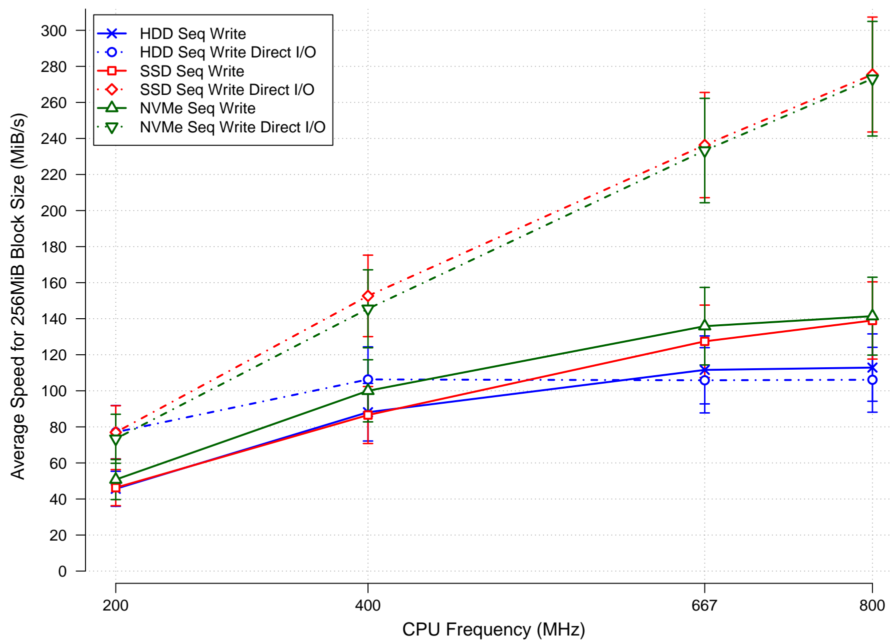


Figure E.1: Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 256 MiB block size at different frequencies (error bars show standard deviation)

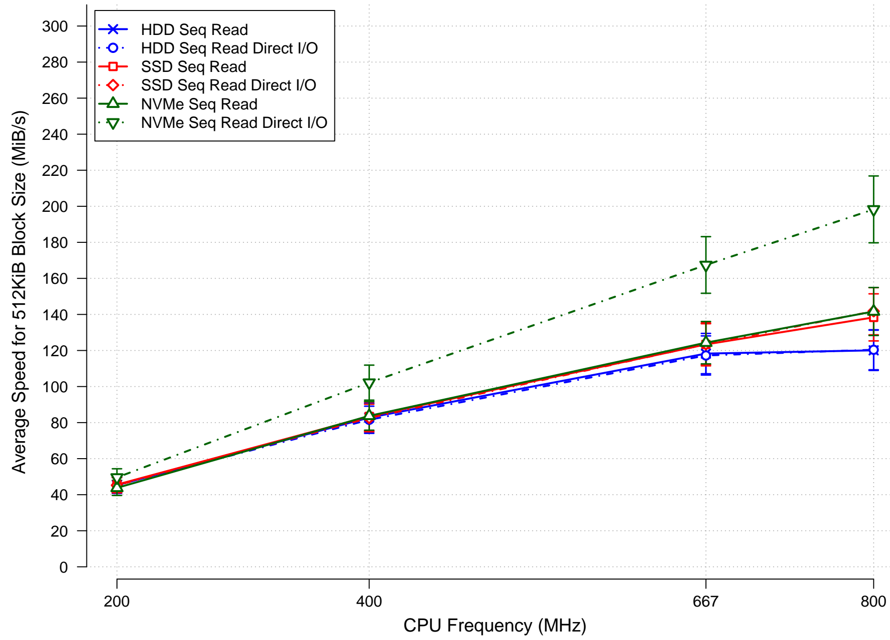


Figure E.2: Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 512 KiB block size at different frequencies (error bars show standard deviation)

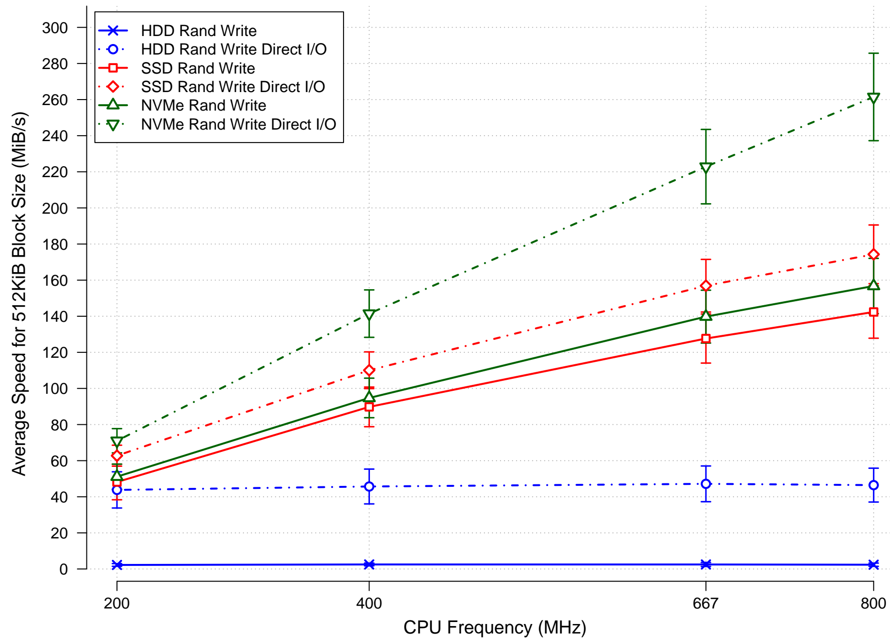


Figure E.3: Write speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 512 KiB block size at different frequencies (error bars show standard deviation)

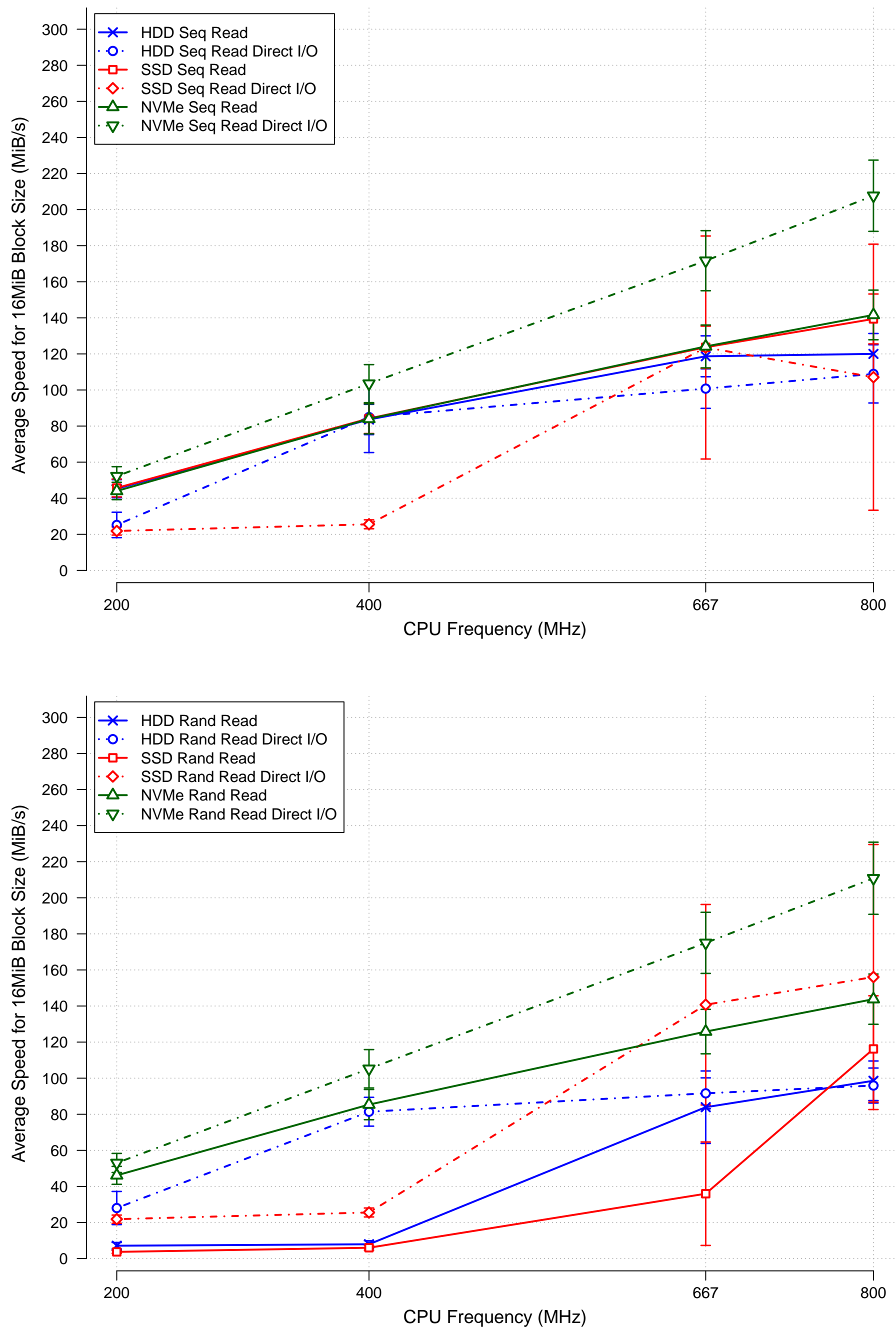


Figure E.4: Read speeds for HDD, SSD and NVMe SSD on the Zynq-7000 platform for 16 MiB block size at different frequencies (error bars show standard deviation)

E.2 Chapter 3 – 10 ms Periodic Task Results

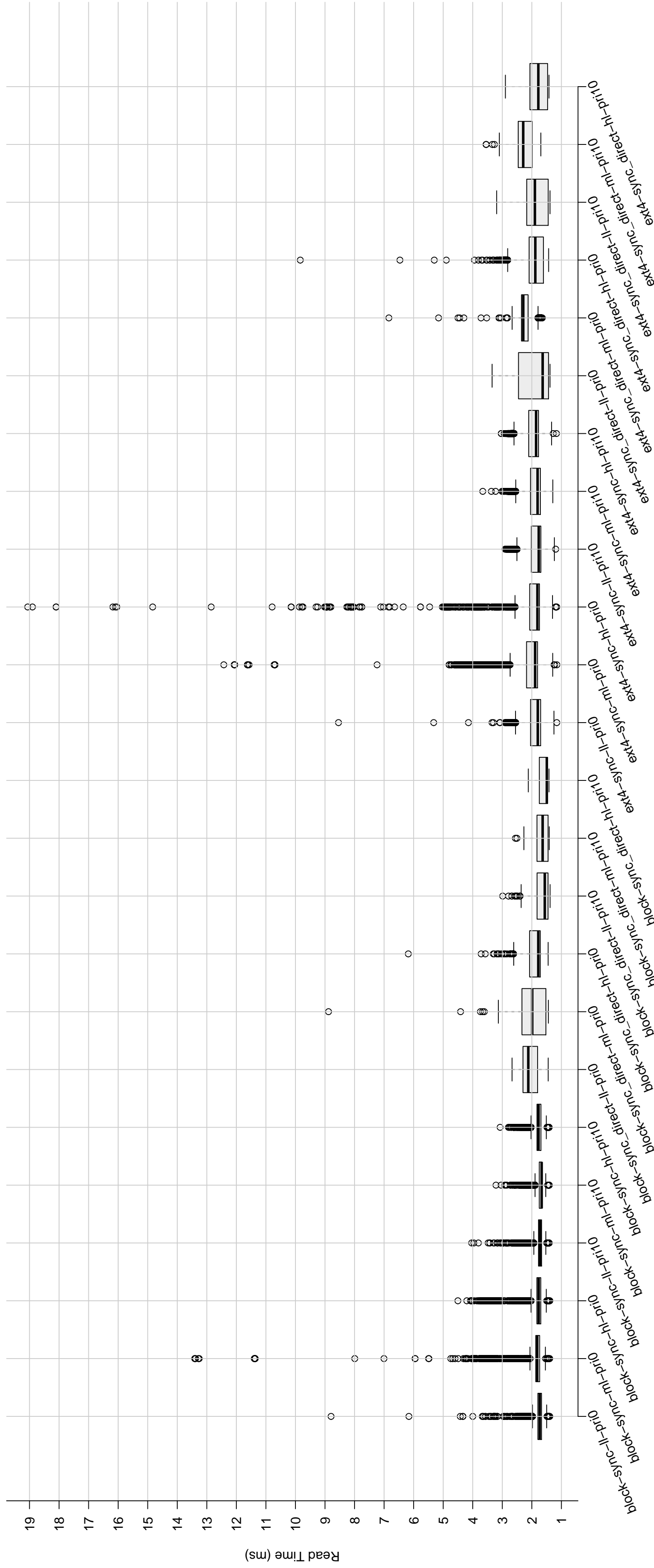


Figure E.5: 10 ms periodic synchronous reads with standard and direct I/O, at low, medium and high load, and low and high priority

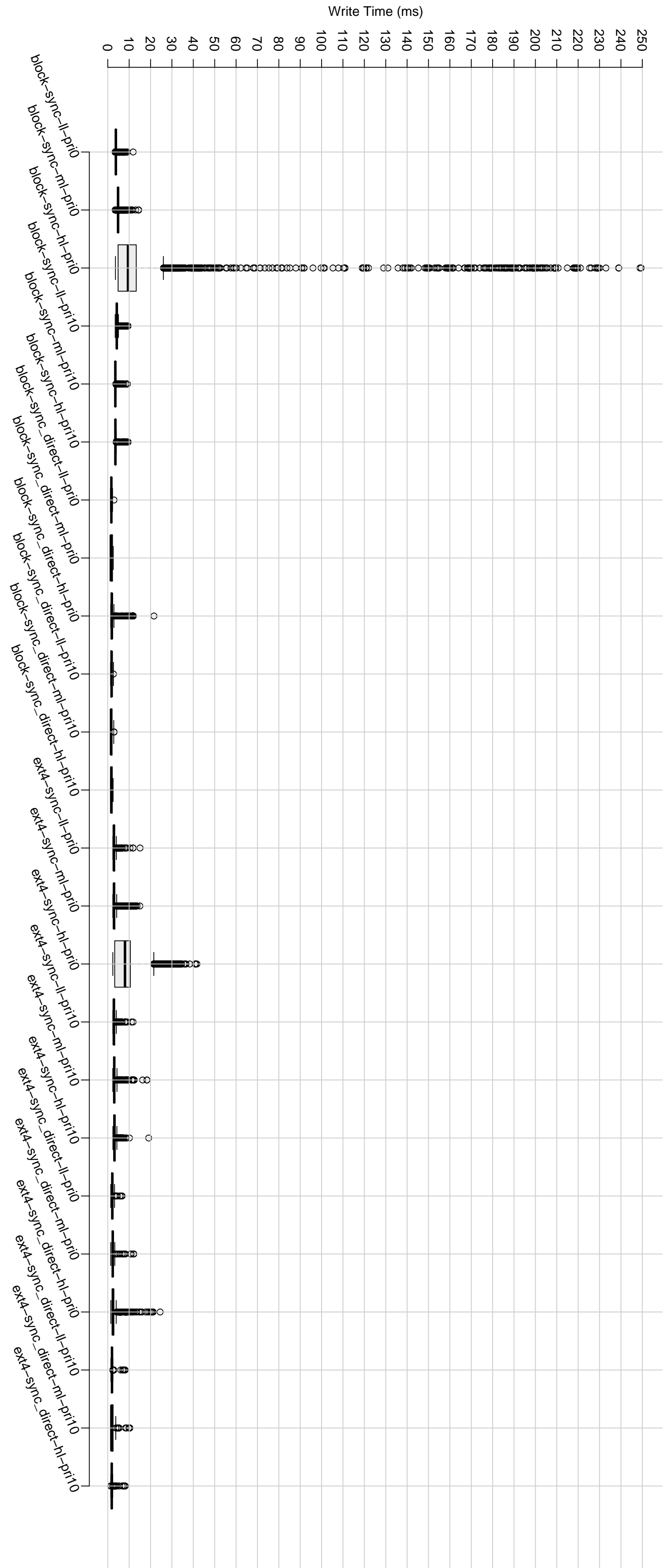


Figure E.6: 10 ms periodic synchronous writes with standard and direct I/O, at low, medium and high load, and low and high priority

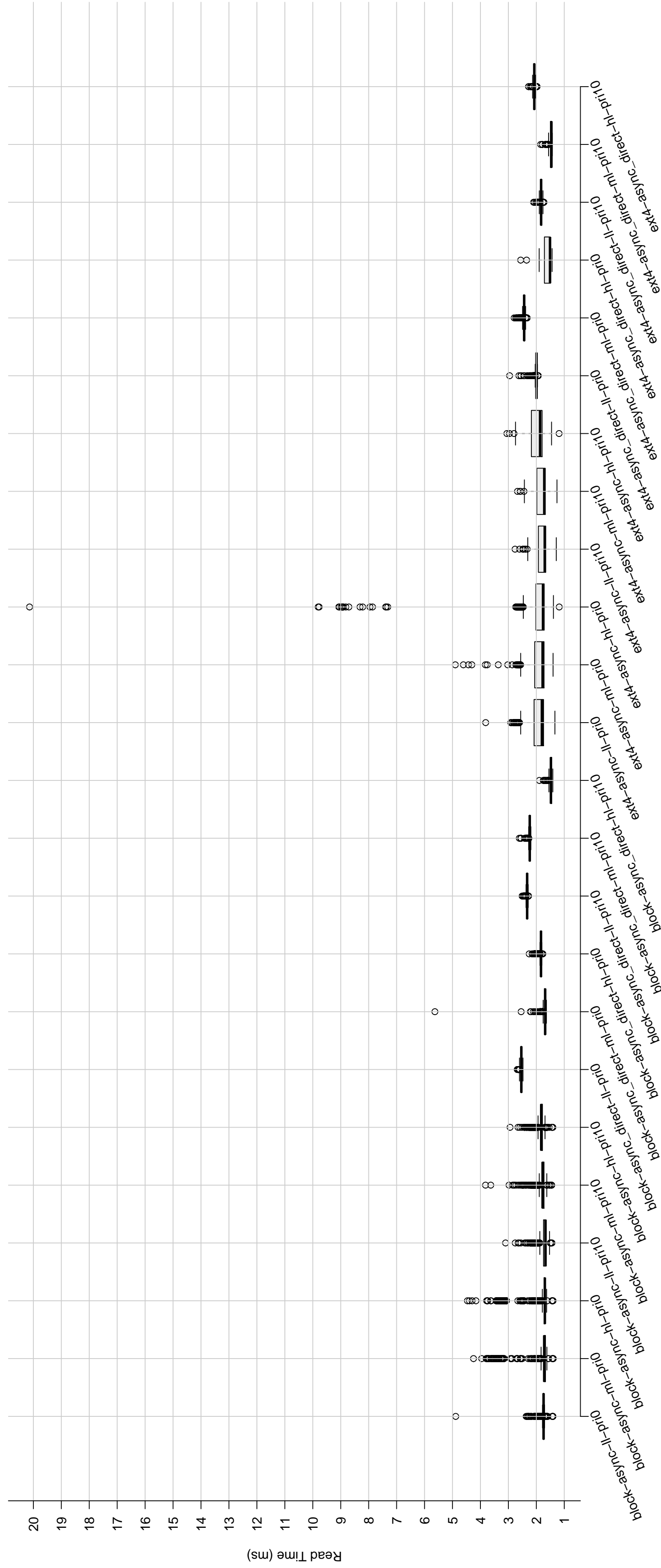


Figure E.7: 10 ms periodic asynchronous reads with standard and direct I/O, at low, medium and high load, and low and high priority

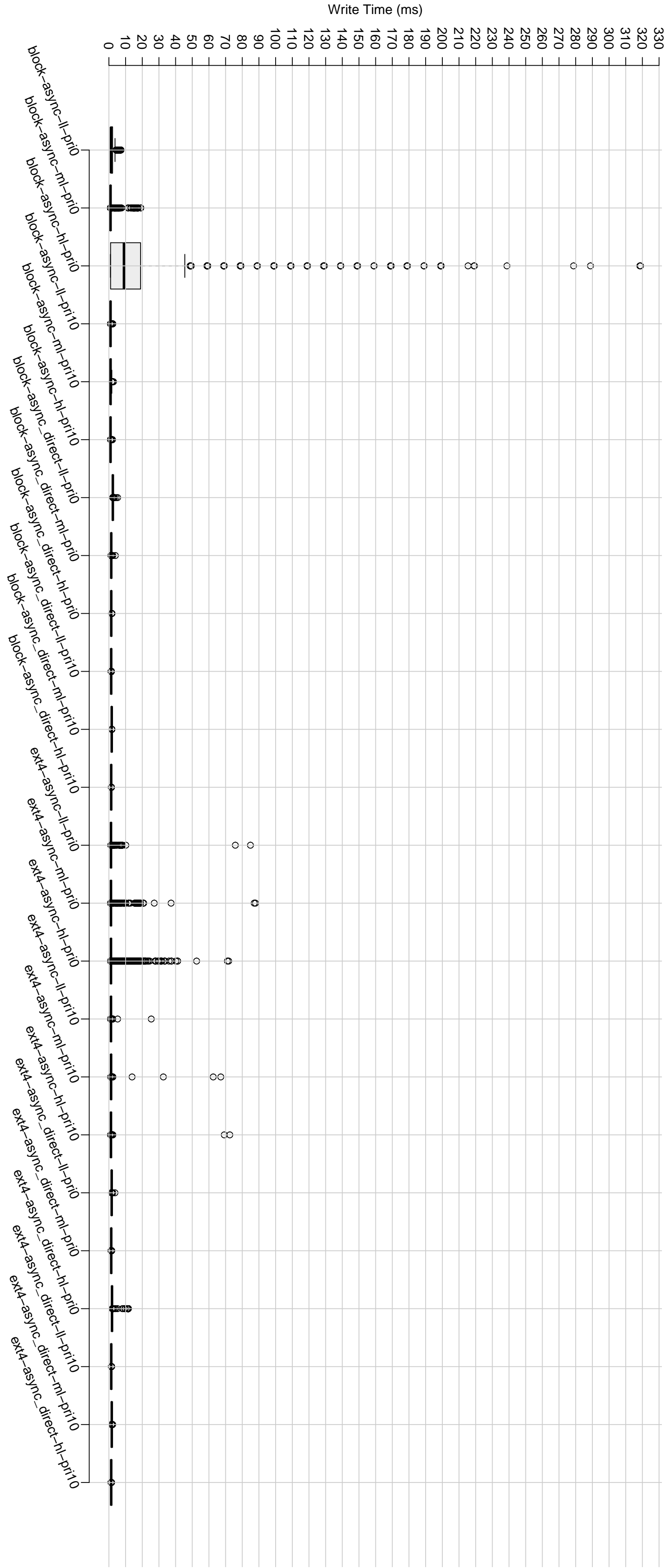


Figure E.8: 10 ms periodic asynchronous writes with standard and direct I/O, at low, medium and high load, and low and high priority

E.3 Chapter 3 – Raw Device Periodic Task Results

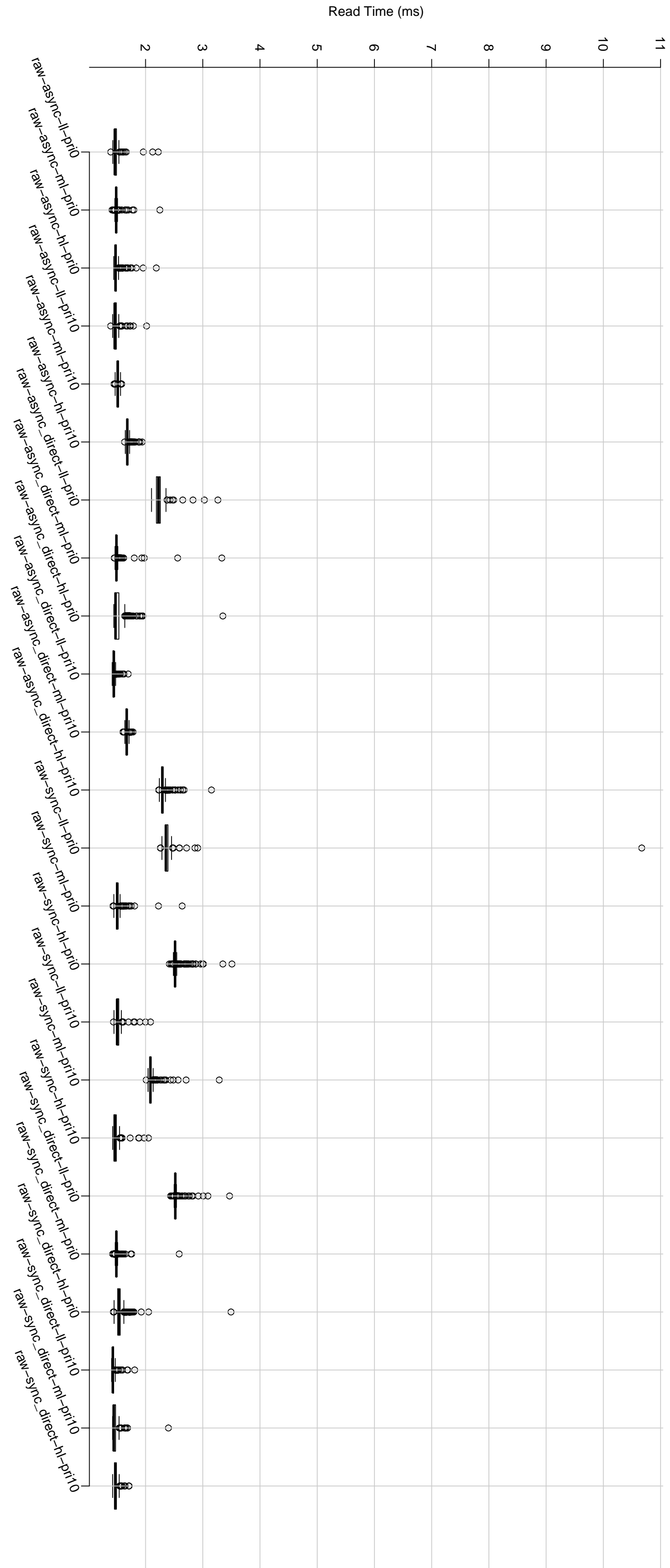


Figure E.9: 25 ms periodic reads with standard and direct I/O, at low, medium and high load, and low and high priority

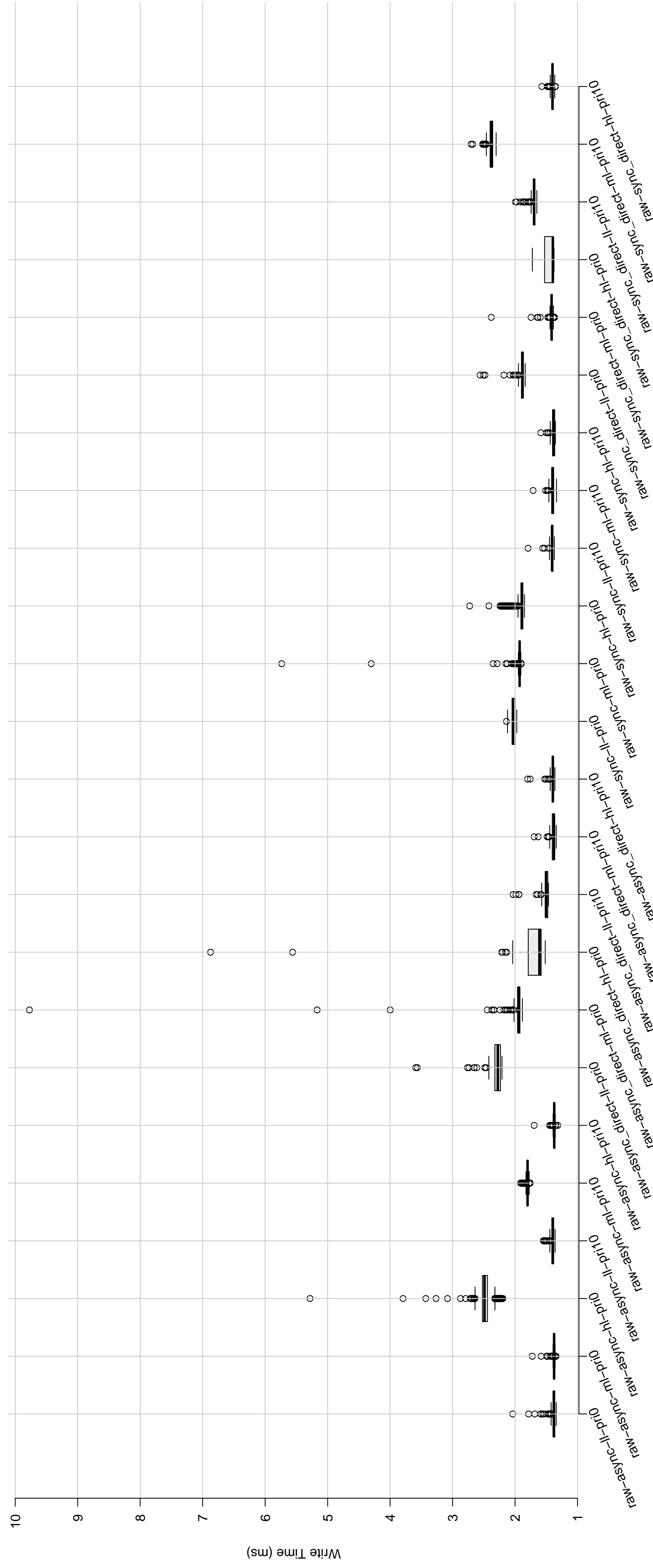


Figure E.10: 25 ms periodic writes with standard and direct I/O, at low, medium and high load, and low and high priority

References

[1] R. Joyce and N. Audsley, 'Exploring storage bottlenecks in Linux-based embedded systems', in *Proc. 5th Embedded Operating Systems Workshop*, Amsterdam, The Netherlands, Oct. 2015 (cit. on p. 15).

[2] ——, 'Exploring storage bottlenecks in Linux-based embedded systems', *ACM SIGBED Review*, vol. 13, no. 1, pp. 54–59, Jan. 2016 (cit. on p. 15).

[3] ——, 'Improving efficiency of persistent storage access in embedded Linux', in *Proc. 10th York Doctoral Symposium on Computer Science and Electronic Engineering*, York, UK, Nov. 2017 (cit. on p. 15).

[4] R. Joyce, *RTSYork/profiling-timer: First release of profiling timer component*, Sep. 2018. DOI: 10.5281/zenodo.1436577. [Online]. Available: https://doi.org/10.5281/zenodo.1436577 (cit. on pp. 15, 98).

[5] ——, *RTSYork/linux-4.1.15-rt17-profiling: First release of Linux profiling modifications*, Sep. 2018. DOI: 10.5281/zenodo.1436031. [Online]. Available: https://doi.org/10.5281/zenodo.1436031 (cit. on pp. 15, 99).

[6] ——, *RTSYork/chario-module: First release of CharIO kernel module*, Sep. 2018. DOI: 10.5281/zenodo.1436029. [Online]. Available: https://doi.org/10.5281/zenodo.1436029 (cit. on pp. 15, 133).

[7] ——, *RTSYork/chario-tests: First release of CharIO test code*, Sep. 2018. DOI: 10.5281/zenodo.1436027. [Online]. Available: https://doi.org/10.5281/zenodo.1436027 (cit. on pp. 15, 98, 133).

[8] R. Joyce and D. Hong, *RTSYork/unvme-zynq: First release of UNVMe port to Zynq MPSoC*, Sep. 2018. DOI: 10.5281/zenodo.1435241. [Online]. Available: https://doi.org/10.5281/zenodo.1435241 (cit. on pp. 16, 149, 150).

[9] R. Joyce, *RTSYork/nvme-microblaze: First release of NVMe driver for MicroBlaze*, Sep. 2018. DOI: 10.5281/zenodo.1435246. [Online]. Available: https://doi.org/10.5281/zenodo.1435246 (cit. on pp. 16, 163).

[10] R. Joyce *et al.*, *RTSYork/filebench: filebench-1.5-alpha3+4kdirect*, Jan. 2020. DOI: 10.5281/zenodo.3598521. [Online]. Available: https://doi.org/10.5281/zenodo.3598521 (cit. on pp. 16, 119, 198).

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger and D. Coetzee, 'Better I/O through byte-addressable, persistent memory', in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009 (cit. on pp. 17, 39, 42).

[12] S. Swanson and A. M. Caulfield, 'Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage', *Computer*, vol. 46, no. 8, pp. 52–59, Aug. 2013 (cit. on pp. 18, 19, 38, 42, 43, 131).

[13] A. Huffman, *NVM Express: Optimized Interface for PCI Express SSDs*, Jul. 2013. [Online]. Available: https://nvmexpress.org/wp-content/uploads/NVM-Express-Optimized-Interface-for-PCI-Express-SSDs-SF13_SSDS004_100.pdf (cit. on pp. 18, 41, 42).

[14] D. H. Walker, *A comparison of NVMe and AHCI*, White Paper, Jul. 2012. [Online]. Available: https://sata-io.org/system/files/member-downloads/NVMe%20and%20AHCI_%20_long_.pdf (cit. on pp. 18, 41).

[15] I. Atkin, *Getting the hang of IOPS*, White Paper, Jun. 2012. [Online]. Available: https://www.symantec.com/connect/sites/default/files/Getting%20The%20Hang%20Of%20IOPS%20v1.3-Whitepaper.pdf (cit. on pp. 19, 38).

[16] Intel Corporation, *Intel SSD 750 Series Product Specifications*. [Online]. Available: https://ark.intel.com/products/86742/Intel-SSD-750-Series-400GB-2_5in-PCIe-3_0-20nm-MLC (cit. on pp. 19, 38, 75, 99, 137, 140, 152, 194).

[17] V. Kasavajhala, *Solid state drive vs. hard disk drive price and performance study*, White Paper, May 2011. [Online]. Available: http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/ssd_vs_hdd_price_and_performance_study.pdf (cit. on p. 19).

[18] National Instruments, *Data acquisition: I/O for embedded systems*, White Paper, Oct. 2012. [Online]. Available: http://www.ni.com/white-paper/7021/en/ (cit. on p. 20).

[19] CERN, *About | Worldwide LHC Computing Grid*. [Online]. Available: http://wlcg-public.web.cern.ch/about (cit. on p. 21).

[20] A. Jacobs, 'The pathologies of big data', *Queue*, vol. 7, no. 6, Jul. 2009. [Online]. Available: http://queue.acm.org/detail.cfm?id=1563874 (cit. on p. 21).

[21] A. Putnam *et al.*, 'A reconfigurable fabric for accelerating large-scale datacenter services', in *Proc. 41st Int. Symp. Computer Architectures*, Jun. 2014 (cit. on pp. 21, 53).

[22] A. Mendon, 'The case for a hardware filesystem', Ph.D. dissertation, University of North Carolina at Charlotte, NC, 2012 (cit. on pp. 21, 30, 31, 49, 54).

[23] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Waltham, MA, USA: Morgan Kaufmann, 2012 (cit. on pp. 22–25, 187).

[24] Wikipedia contributors, *Instructions per second — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Instructions_per_second&oldid=847448846 (cit. on pp. 23, 25, 183).

[25] Techopedia, Inc., *Extended Data Out Random Access Memory (EDO RAM)*. [Online]. Available: https://www.techopedia.com/definition/6981/extended-data-out-random-access-memory-edo-ram (cit. on pp. 24, 25).

[26] Wikipedia contributors, *Synchronous dynamic random-access memory — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Synchronous_dynamic_random-access_memory&oldid=852826003 (cit. on pp. 24, 25).

[27] Frank Denneman, *Memory Deep Dive: DDR4 Memory*, Feb. 2015. [Online]. Available: http://frankdenneman.nl/2015/02/25/memory-deep-dive-ddr4/ (cit. on pp. 24, 25).

[28] Micron Technology, Inc., *Memory speeds and compatibility*. [Online]. Available: http://uk.crucial.com/gbr/en/support-memory-speeds-compatibility (cit. on pp. 24, 25).

[29] C. Sandler, *Fix Your Own PC*, 8th ed. Hoboken, NJ, USA: Wiley, 2007 (cit. on pp. 24, 25, 185).

[30] C. W. Carlson, 'Gen 6 Fibre Channel what you need to know', in *Proc. 2014 Data Storage Innovation Conference*, Santa Clara, CA, USA, Apr. 2014. [Online]. Available: http://www.snia.org/sites/default/orig/DSI2014/presentations/StorPlumb/CraigCarlson_Gen_6_Fibre_Channel%20_v02.pdf (cit. on pp. 24, 25, 185).

[31] Wikipedia contributors, *Parallel SCSI — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Parallel_SCSI&oldid=859515769 (cit. on pp. 24, 25, 185).

[32] ——, *Serial Attached SCSI — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Serial_Attached_SCSI&oldid=853550545 (cit. on pp. 24, 25, 185).

[33] ——, *PCI Express — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=PCI_Express&oldid=860582955 (cit. on pp. 24, 25, 185).

[34] ——, *Serial ATA — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Serial_ATA&oldid=859651783 (cit. on pp. 24, 25, 185).

[35] A. Mendon, 'Design and implementation of a hardware filesystem', M.S. thesis, University of North Carolina at Charlotte, NC, 2008 (cit. on pp. 30, 54).

[36] S. Ahn, J. Choi, D. Lee, S. H. Noh, S. L. Min and Y. Cho, 'Design, implementation, and performance evaluation of flash memory-based file system on chip', *Journal of Information Science and Engineering*, vol. 23, no. 6, pp. 1865–1887, 2007 (cit. on pp. 30, 55).

[37] 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 (cit. on pp. 31, 54).

[38] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen and T. Moscibroda, 'The Feniks FPGA operating system for cloud computing', in *Proc. 8th Asia-Pacific Workshop on Systems*, Mumbai, India, Sep. 2017 (cit. on pp. 31, 58).

[39] 'JUNIPER D2.3 – static acceleration design', University of York et al., Tech. Rep., version 1.0, Dec. 2013 (cit. on p. 31).

[40] G. Singer, *The History of the Modern Graphics Processor*, Mar. 2013. [Online]. Available: http://www.techspot.com/article/650-history-of-the-gpu/ (cit. on p. 31).

[41] Xilinx, Inc., *Zynq-7000 SoC*. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (cit. on pp. 31, 140, 174, 190).

[42] ——, *Zynq UltraScale+ MPSoC*. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html (cit. on pp. 31, 34, 148, 166, 174, 192).

[43] Avnet, Inc., *Mini-ITX board*. [Online]. Available: http://zedboard.org/product/mini-itx-board (cit. on pp. 32, 84, 140, 190).

[44] Xilinx, Inc., *Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html (cit. on pp. 32, 149, 166, 192).

[45] ——, *Virtex-7 FPGA Family*. [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html (cit. on p. 32).

[46] ASICS World Services, Ltd., *Serial ATA I/II/III Host Controller IP Core*, Product Brief, 2015. [Online]. Available: http://www.asics.ws/doc/sata_host_brief.pdf (cit. on p. 33).

[47] Design Gateway Co., Ltd., *Serial ATA-IP core Introduction*, Product Brief, Jan. 2010. [Online]. Available: http://www.dgway.com/products/IP/SATA-IP/SATA2-Brief-E.pdf (cit. on p. 33).

[48] IntelliProp, Inc., *IPC-SA101A-HI SATA Host App Core*. [Online]. Available: http://www.intelliprop.com/hardware-storage-design/ip-cores/sata-host-core.htm (cit. on p. 33).

[49] A. A. Mendon, B. Huang and R. Sass, 'A high performance, open source SATA2 core', in *Proc. Field Programmable Logic and Applications*, Aug. 2012 (cit. on pp. 33, 54).

[50] L. Woods and K. Eguro, 'Groundhog – a Serial ATA host bus adapter (HBA) for FPGAs', in *Proc. 20th Int. Symp. Field-Programmable Cust. Comput. Mach.*, 2012, pp. 220–223 (cit. on p. 33).

[51] C. Gorman, P. Siqueira and R. Tessier, 'An open-source SATA core for Virtex-4 FPGAs', in *Proc. Int. Conf. Field-Programmable Technol.*, 2013, pp. 454–457 (cit. on p. 34).

[52] S. Tam and L. Jones, *Embedded serial ATA storage system (XAPP716)*, Application Note, Oct. 2006. [Online]. Available: http://www.xilinx.com/support/answers/31408.htm (cit. on p. 34).

[53] Xilinx, Inc., *Vivado Design Suite*. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html (cit. on p. 34).

[54] ——, *PetaLinux Tools*. [Online]. Available: http://www.xilinx.com/tools/petalinux-sdk.htm (cit. on p. 34).

[55] ——, *Vivado High-Level Synthesis*. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/ (cit. on pp. 35, 168).

[56] ——, *SDSoC Development Environment*. [Online]. Available: http://www.xilinx.com/products/design-tools/sdx/sdsoc.html (cit. on p. 35).

[57] ——, *SDAccel Development Environment*. [Online]. Available: http://www.xilinx.com/products/design-tools/sdx/sdaccel.html (cit. on p. 35).

[58] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2005 (cit. on pp. 35–38, 49, 57, 59).

[59] R. Love, *Linux Kernel Development*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2010 (cit. on p. 36).

[60] M. T. Jones, *Anatomy of the Linux file system*, Oct. 2007. [Online]. Available: http://www.ibm.com/developerworks/library/l-linux-filesystem/ (cit. on p. 37).

[61] ——, *Anatomy of the Linux virtual file system switch*, Aug. 2009. [Online]. Available: http://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/ (cit. on p. 37).

[62] G. Kroah-Hartman, *Linux Kernel in a Nutshell*. Sebastopol, CA, USA: O'Reilly Media, 2006 (cit. on p. 38).

[63] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee and J. Jeong, 'Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs', in *Proc. 2019 USENIX Annual Technical Conference*, Renton, WA, USA, Jul. 2019 (cit. on pp. 38, 46).

[64] 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, USA, Nov. 2010 (cit. on pp. 39, 43, 44, 53).

[65] F. T. Hady, A. Foong, B. Veal and D. Williams, 'Platform storage performance with 3D XPoint technology', *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, Sep. 2017. DOI: 10.1109/JPROC.2017.2731776 (cit. on p. 39).

[66] The Serial ATA International Organization, *The SATA Ecosystem*. [Online]. Available: https://sata-io.org/developers/sata-ecosystem (cit. on p. 40).

[67] ——, *Native Command Queueing*. [Online]. Available: https://www.sata-io.org/native-command-queuing (cit. on pp. 40, 53).

[68] Microsemi Corporation, *24G SAS (SAS-4)*. [Online]. Available: https://www.microsemi.com/product-directory/upcoming-technology/5440-serial-attached-scsi-technology-sas-4 (cit. on p. 40).

[69] D. Riley, *Intel SSD DC P3700 800GB and 1.6TB Review: The Future of Storage*, Aug. 2014. [Online]. Available: http://www.tomshardware.co.uk/intel-ssd-dc-p3700-nvme,review-33018.html (cit. on p. 41).

[70] J. Coburn, T. Bunker, M. Schwarz, R. Gupta and S. Swanson, 'From ARIES to MARS: Transaction support for next-generation, solid-state drives', in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, Nov. 2013 (cit. on p. 44).

[71] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu and S. Swanson, 'Willow: A user-programmable SSD', in *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, USA, Oct. 2014 (cit. on p. 44).

[72] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn and S. Swanson, 'Providing safe, user space access to fast, solid state disks', in *Proc. 17th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Mar. 2012 (cit. on pp. 44, 45).

[73] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta and S. Swanson, 'Moneta: A high-performance storage array architecture for next-generation, non-volatile memories', in *Proc. 43rd Annu. Int. Symp. Microarchitecture*, Dec. 2010 (cit. on p. 44).

[74] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena and M. M. Swift, 'Aerie: Flexible file-system interfaces to storage-class memory', in *Proc. 9th Eur. Conf. Comput. Syst.*, Apr. 2014 (cit. on p. 45).

[75] E. Zadok, V. Tarasov and P. Sehgal, 'The case for specialized file systems, or, fighting file system obesity', *;login: The USENIX Magazine*, vol. 35, no. 1, pp. 38–40, Feb. 2010 (cit. on pp. 45, 47, 130, 131).

[76] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu and G. Palani, 'Designing a true direct-access file system with DevFS', in *Proc. 16th USENIX Conf. File and Storage Technologies*, Oakland, CA, USA, Feb. 2018 (cit. on p. 45).

[77] E. Zadok and I. Badulescu, 'A stackable file system interface for Linux', in *Proc. LinuxExpo*, 1999, pp. 141–151 (cit. on pp. 47, 48).

[78] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu and C. P. Wright, 'On incremental file system development', *ACM Transactions on Storage*, vol. 2, no. 2, pp. 161–196, 2006 (cit. on p. 47).

[79] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger and E. Zadok, 'RAIF: Redundant array of independent filesystems', in *Proc. 24th IEEE Conference on Mass Storage Systems and Technologies*, San Diego, CA, USA, Sep. 2007 (cit. on p. 47).

[80] 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 (cit. on pp. 48, 53, 130, 131).

[81] M. Patočka, 'Design and implementation of the Spad filesystem', Ph.D. dissertation, Charles University in Prague, 2011 (cit. on p. 48).

[82] ——, 'An architecture for high performance file system I/O', *International Journal of Computer, Information, Systems and Control Engineering*, vol. 1, no. 5, 2007 (cit. on p. 48).

[83] Oracle Corporation, *Unleashing application performance with solid-state drives and Sun servers*, White Paper, May 2010. [Online]. Available: https://www.oracle.com/technetwork/articles/systems-hardware-architecture/app-performance-ssd-sun-servers-163876.pdf (cit. on p. 50).

[84] A. Molano, K. Juvva and R. Rajkumar, 'Real-time filesystems: Guaranteeing timing constraints for disk accesses in RT-Mach', in *Proc. 18th IEEE Real-Time Systems Symposium*, Dec. 1997 (cit. on p. 50).

[85] S. Reif, L. Gerhorst, K. Bender and T. Hönig, 'Towards low-jitter and energy-efficient data processing in cyber-physical information systems', in *Proc. 52nd Hawaii International Conference on System Sciences*, Grand Wailea, Maui, HI, USA, Jan. 2019 (cit. on p. 50).

[86] G. D. Nijs, B. V. D. Brink and W. Almesberger, 'Active block I/O scheduling system (ABISS)', in *Proc. Linux Symposium*, Jul. 2005, pp. 109–126 (cit. on p. 50).

[87] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong and C. Maltzahn, 'Efficient guaranteed disk request scheduling with Fahrrad', in *Proc. 3rd ACM European Conference on Computer Systems*, Apr. 2008, pp. 13–25 (cit. on p. 50).

[88] V. Tarasov, G. Sim, A. Povzner and E. Zadok, 'Efficient I/O scheduling with accurately estimated disk drive latencies', in *Proc. 8th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2012 (cit. on p. 51).

[89] K. Shvachko, H. Kuang, S. Radia and R. Chansler, 'The Hadoop distributed file system', in *Proc. 26th IEEE Symposium on Mass Storage Systems and Technologies*, Incline Village, NV, USA, May 2010 (cit. on p. 51).

[90] P. Schwan, 'Lustre: Building a file system for 1,000-node clusters', in *Proc. Linux Symposium*, Ottawa, Canada, Jul. 2003 (cit. on p. 51).

[91] J. Xu and S. Swanson, 'NOVA: A log-structured file system for hybrid volatile/non-volatile main memories', in *Proc. 14th USENIX Conf. File and Storage Technologies*, Santa Clara, CA, USA, Feb. 2016 (cit. on p. 52).

[92] C. Kalita, G. Barua and P. Sehgal, 'DurableFS: A file system for NVRAM', *CSI Transactions on ICT*, Apr. 2019. DOI: 10.1007/s40012-019-00215-0 (cit. on p. 52).

[93] Altera Corporation, *Adding hardware accelerators to reduce power in embedded systems*, White Paper, Sep. 2009. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01112-hw-reduce-power.pdf (cit. on p. 52).

[94] M. Ouellette and D. Connors, 'Analysis of hardware acceleration in reconfigurable embedded systems', in *Proc. 19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005 (cit. on p. 52).

[95] J. van Lunteren and A. Guanella, 'Hardware-accelerated regular expression matching at multiple tens of gb/s', in *Proc. IEEE International Conference on Computer Communications*, Jan. 2012 (cit. on p. 52).

[96] S. Wong, F. Duarte and S. Vassiliadis, 'A hardware cache memcpy accelerator', in *Proc. IEEE International Conference on Field Programmable Technology*, Dec. 2006 (cit. on pp. 52, 57).

[97] 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 (cit. on p. 53).

[98] E. Riedel, G. Gibson and C. Faloutsos, 'Active storage for large-scale data mining and multimedia', in *Proc. 24th Int. Conf. Very Large Databases*, New York, NY, USA, Aug. 1998 (cit. on p. 53).

[99] A. Acharya, M. Uysal and J. Saltz, 'Active disks: Programming model, algorithms and evaluation', in *Proc. eighth Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, San Jose, CA, USA, Oct. 1998 (cit. on p. 53).

[100] G. R. Ganger, 'Blurring the line between OSes and storage devices', Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep., Dec. 2001 (cit. on p. 53).

[101] A. A. Mendon, A. G. Schmidt and R. Sass, 'A hardware filesystem implementation with multidisk support', *International Journal of Reconfigurable Computing*, vol. 2009, 2009 (cit. on pp. 54, 131).

[102] R. Card, T. Ts'o and S. Tweedie, 'Design and implementation of the second extended filesystem', in *Proc. First Dutch International Symposium on Linux*, Dec. 2004 (cit. on p. 54).

[103] C. A. H. Jemma, 'Simulating a hardware accelerated filesystem', M.S. thesis, University of York, UK, 2012 (cit. on p. 56).

[104] Exar Corporation, *AltraHD: High performance application transparent compression for Hadoop*, Product Brief, 2014. [Online]. Available: http://www.exar.com/common/content/document.ashx?id=21341 (cit. on p. 56).

[105] Oracle Corporation, *Oracle advanced security transparent data encryption using Sun Crypto Accelerator 6000 PCIe card*, White Paper, Jun. 2010. [Online]. Available: http://www.oracle.com/technetwork/architect/resources/articles/whitepapers-security-159820.html (cit. on p. 56).

[106] F. Duarte and S. Wong, 'Profiling Bluetooth and Linux on the Xilinx Virtex II Pro', in *Proc. 9th EUROMICRO Conference on Digital System Design*, Aug. 2006 (cit. on p. 57).

[107] H. K.-H. So, 'BORPH: An operating system for FPGA-based reconfigurable computers', Ph.D. dissertation, University of California at Berkeley, CA, Jul. 2007 (cit. on p. 58).

[108] H. K.-H. So and R. Brodersen, 'File system access from reconfigurable FPGA hardware processes in BORPH', in *Proc. Int. Conf. F. Program. Log. Appl.*, Sep. 2008, pp. 567–570 (cit. on p. 58).

[109] *perf: Linux profiling with performance counters*, 2015. [Online]. Available: https://perf.wiki.kernel.org/index.php?title=Main_Page&oldid=3535 (cit. on p. 59).

[110] *About OProfile*, 2015. [Online]. Available: http://oprofile.sourceforge.net/about/ (cit. on p. 59).

[111] A. Z. Netto and R. S. Arnold, *Evaluate performance for Linux on POWER*, Jun. 2012. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-evaluatelinuxonpower/ (cit. on p. 59).

[112] S. L. Graham, P. B. Kessler and M. K. Mckusick, 'gprof: A call graph execution profiler', in *Proc. 1982 SIGPLAN Symposium on Compiler Construction*, Boston, MA, USA, Jun. 1982 (cit. on p. 59).

[113] B. Gregg, *DTrace Tools*, 2018. [Online]. Available: http://www.brendangregg.com/dtrace.html (cit. on p. 59).

[114] A. Traeger, E. Zadok, N. Joukov and C. P. Wright, 'A nine year study of file system and storage benchmarking', *ACM Transactions on Storage*, vol. 2, no. 4, May 2008 (cit. on p. 60).

[115] V. Tarasov, S. Bhanage, E. Zadok and M. Seltzer, 'Benchmarking file system benchmarking: It *IS* rocket science', in *Proc. 13th USENIX Workshop in Hot Topics in Operating Systems*, Napa Valley, CA, USA, May 2011 (cit. on p. 60).

[116] N. Agrawal, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, 'Generating realistic impressions for file-system benchmarking', *ACM Transactions on Storage*, vol. 5, no. 4, Dec. 2009 (cit. on p. 60).

[117] The FreeBSD Documentation Project, *9.4. Block Devices (Are Gone)*, Revision 52176, 2018. [Online]. Available: https://www.freebsd.org/doc/en/books/arch-handbook/driverbasics-block.html (cit. on p. 69).

[118] L. Torvalds, *Re: O_DIRECT question*, Linux Kernel Mailing List, Jan. 2007. [Online]. Available: https://lkml.org/lkml/2007/1/11/121 (cit. on pp. 70, 106).

[119] R. P. J. Day, *[PATCH] Remove obsolete raw device support.* Linux Kernel Mailing List, Feb. 2007. [Online]. Available: https://lkml.org/lkml/2007/2/16/431 (cit. on p. 71).

[120] D. Jones, *undeprecate raw driver.* Linux Kernel Mailing List, May 2007. [Online]. Available: https://lkml.org/lkml/2007/5/13/92 (cit. on pp. 71, 106).

[121] *open(2) – Linux Programmer's Manual*, Release 4.02, Aug. 2015 (cit. on p. 72).

[122] *OProfile – A System Profiler for Linux*, Aug. 2015. [Online]. Available: http://oprofile.sourceforge.net/ (cit. on p. 73).

[123] Western Digital Corporation, *WD Blue PC Desktop Hard Drive*. [Online]. Available: https://www.wdc.com/products/internal-storage/wd-blue-pc-desktop-hard-drive.html (cit. on pp. 75, 194).

[124] Intel Corporation, *Intel SSD 535 Series Product Specifications*. [Online]. Available: https://ark.intel.com/products/86734/Intel-SSD-535-Series-240GB-2_5in-SATA-6Gbs-16nm-MLC (cit. on pp. 75, 194).

[125] StarTech.com, *PCI Express SATA III Controller Card*. [Online]. Available: https://www.startech.com/uk/Cards-Adapters/HDD-Controllers/SATA-Cards/4-Port-PCI-Express-SATA-6Gbps-RAID-Controller-Card~PEXSAT34RH (cit. on pp. 75, 194).

[126] J. Axboe, *fio – Flexible I/O tester*. [Online]. Available: https://fio.readthedocs.io/ (cit. on pp. 76, 153).

[127] V. Tarasov, E. Zadok and S. Shepler, 'Filebench: A flexible framework for file system benchmarking', *;login: The USENIX Magazine*, vol. 41, no. 1, pp. 6–12, Mar. 2016 (cit. on pp. 118, 120).

[128] V. Tarasov, *Filebench – Predefined personalities*, Jul. 2016. [Online]. Available: https://github.com/filebench/filebench/wiki/Predefined-personalities/0dfec37c8e64e8f46b6bde9ef4dcd0eeb8c46458 (cit. on p. 119).

[129] Real Time Engineers Ltd., *The FreeRTOS Kernel*. [Online]. Available: http://www.freertos.org (cit. on pp. 124, 129).

[130] The RTEMS Project, *RTEMS Real Time Operating System (RTOS)*, https://www.rtems.org (cit. on pp. 124, 129).

[131] Micron Technology, Inc., *UNVMe - A User Space NVMe Driver*. [Online]. Available: https://github.com/MicronSSD/unvme (cit. on pp. 125, 148).

[132] J. S., *Introduction to the Storage Performance Development Kit (SPDK)*, Sep. 2016. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk (cit. on p. 125).

[133] J. Brown and B. Martin, 'How fast is fast enough? Choosing between Xenomai and Linux for real-time applications', in *Proc. 12th Real-Time Linux Workshop*, Nairobi, Kenya, Oct. 2010 (cit. on p. 129).

[134] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 4th ed. Addison Wesley, 2009 (cit. on p. 130).

[135] Z. Cao, V. Tarasov, H. Raman, D. Hildebrand and E. Zadok, 'On the performance variation in modern storage stacks', in *Proc. 15th USENIX Conference on File and Storage Technologies (FAST 17)*, Santa Clara, CA, USA: USENIX Association, Feb. 2017, pp. 329–343 (cit. on p. 131).

[136] R. Santana, R. Rangaswami, V. Tarasov and D. Hildebrand, 'A fast and slippery slope for file systems', *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 27–34, Jan. 2016, ISSN: 0163-5980 (cit. on p. 131).

[137] E. H. M. Sha, Y. Jia, X. Chen, Q. Zhuge, W. Jiang and J. Qin, 'The design and implementation of an efficient user-space in-memory file system', in *Proc. 5th Non-Volatile Memory Systems and Applications Symp. (NVMSA)*, Daegu, South Korea, Aug. 2016 (cit. on p. 131).

[138] M. Bjørling, J. Gonzalez and P. Bonnet, 'LightNVM: The Linux open-channel SSD subsystem', in *Proc. 15th USENIX Conference on File and Storage Technologies (FAST 17)*, Santa Clara, CA, USA, Feb. 2017, pp. 359–374 (cit. on p. 131).

[139] *Real-Time Linux*. [Online]. Available: https://wiki.linuxfoundation.org/realtime/start (cit. on p. 140).

[140] Micron Technology, Inc., *UNFS - User Space Nameless Filesystem*. [Online]. Available: https://github.com/MicronSSD/unfs (cit. on p. 148).

[141] 'PHANTOM D1.4 – final design for cross-layer programming, security and runtime monitoring', University of York et al., Tech. Rep., version 1.0, Apr. 2019 (cit. on p. 152).

[142] Xilinx, Inc., *MicroBlaze Soft Processor Core*. [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html (cit. on p. 163).

[143] K. Bailey, L. Ceze, S. D. Gribble and H. M. Levy, 'Operating system implications of fast, cheap, non-volatile memory', in *Proc. 13th USENIX conference on hot topics in operating systems*, Napa, CA, USA, May 2011 (cit. on p. 173).

[144] JEDEC Solid State Technology Association, *JEDEC announces support for NVDIMM hybrid memory modules*, Press Release, May 2015. [Online]. Available: https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules (cit. on pp. 173, 178).

[145] B. Gervasi and J. Hinkle, 'Overcoming system memory challenges with persistent memory and NVDIMM-P', in *JEDEC Server Forum 2017*, Jun. 2017. [Online]. Available: https://www.jedec.org/sites/default/files/Bill_Gervasi.pdf (cit. on pp. 173, 179).

[146] Xilinx, Inc., *Xilinx Introduces UltraScale Multi-Processing Architecture for the Industry's First All Programmable MPSoCs*, Feb. 2014. [Online]. Available: https://www.xilinx.com/news/press/2014/xilinx-introduces-ultrascale-multi-processing-architecture-for-the-industry-s-first-all-programmable-mpsocs.html (cit. on p. 178).

[147] Wikipedia contributors, *Transistor count — Wikipedia, the free encyclopedia*, 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Transistor_count&oldid=860599625 (cit. on p. 186).

[148] Xilinx, Inc., *Virtex-E 1.8 V Field Programmable Gate Arrays*, Mar. 2014. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds112.pdf (cit. on p. 186).

[149] ——, *Xilinx Virtex-II Series FPGAs*. [Online]. Available: https://www.xilinx.com/publications/matrix/virtexmatrix.pdf (cit. on p. 186).

[150] ——, *Virtex-4 Family Overview*, Aug. 2010. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds112.pdf (cit. on p. 186).

[151] ——, *Virtex-5 Family Overview*, Aug. 2015. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds100.pdf (cit. on p. 186).

[152] ——, *Virtex-6 Family Overview*, Aug. 2015. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf (cit. on p. 186).

[153] ——, *7 Series FPGAs Data Sheet: Overview*, Feb. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (cit. on p. 186).

[154] ——, *UltraScale Architecture and Product Data Sheet: Overview*, Aug. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf (cit. on p. 186).

[155] Intel Corporation, *Intel Core 2 Quad Processor Q9450*. [Online]. Available: https://ark.intel.com/products/33923/Intel-Core2-Quad-Processor-Q9450-12M-Cache-2_66-GHz-1333-MHz-FSB (cit. on p. 189).

[156] V. Tarasov, *Filebench – Workload model language*, May 2017. [Online]. Available: https://github.com/filebench/filebench/wiki/Workload-model-language/d88e134c876552f3d726fd1268306ffba0ee596f (cit. on p. 198).

[157] V. Tarasov and ezk, *Filebench – Collected metrics*, May 2017. [Online]. Available: https://github.com/filebench/filebench/wiki/Collected-metrics/725835ea71b9b7d7488988657c337a4ae9336947 (cit. on pp. 219, 235).