Improving Efficiency of Persistent Storage Access in Embedded Linux

Russell Joyce and Neil Audsley
Department of Computer Science
University of York, York, UK
{russell.joyce, neil.audsley}@york.ac.uk

***Abstract*—**
Real-time embedded systems increasingly need to process and store large volumes of persistent data, requiring fast, timely and predictable storage. Traditional methods of accessing storage using general-purpose operating system-based file systems do not provide the performance and timing predictability needed. This paper firstly examines the speed and consistency of SSD operations in an embedded Linux system, identifying areas where inefficiencies in the storage stack cause issues for performance and predictability. Secondly, the CharIO storage device driver is proposed to bypass Linux file systems and the kernel block layer, in order to increase performance, and provide improved timing predictability.

I. 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. Whilst file system performance in real-time operating systems such as FreeRTOS [1] and RTEMS [2] is improving, for embedded Linux, provision of predictable, efficient access to storage remains an open issue, despite improvements to Linux for real-time use [3]. This paper 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.

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. Whilst 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 [4], [5].

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 [6]. Initially in this paper we show that using the standard Linux file system provision does not lead to sufficiently predictable storage access. The paper then proposes a new approach that removes the complexity of the Linux file system to provide applications with fast, predictable access to persistent storage. We propose to bypass the file system and block layer entirely, thus removing a number of obstacles to loading and storing data, at the expense of conveniences such as a file hierarchy and system-wide caching. We further extend this method to support physical memory addressing for storage commands from user applications, bypassing extra levels of Linux such as virtual memory and cache management, and allowing the direct transfer of stored data from addressable areas outside of main memory. We also propose a potential management interface through 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 remainder of the paper is structured as follows. Section II introduces appropriate background and related work. Section III describes the CharIO storage interface, which is evaluated along with other storage access methods in Section IV. Conclusions and ideas for further work are offered in Section V.

II. 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 [7]. 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

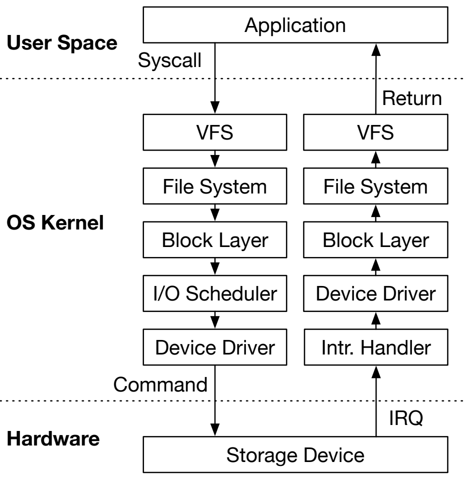

Fig. 1. Storage access layers in Linux

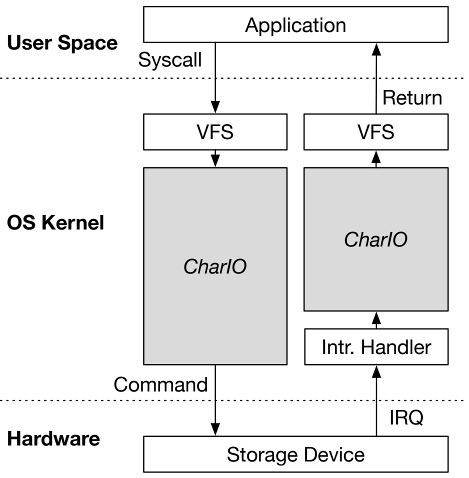

Fig. 2. Storage access layers using CharIO driver

may not be appropriate for some systems [5]. 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 [8], and when pairings between file systems and storage device types can have an extreme effect on how well each performs [9]. Alongside suggestions to reduce the 'obesity' of file systems by tailoring them to be more appropriate to the systems they serve [4], more-extreme suggestions for improving storage performance include replacing operating system support with a more efficient in-memory user-space file system [10], or offloading file system functionality entirely into a custom hardware accelerator core [11].

Accessing storage in Linux, and in most other operating systems, involves several interacting layers of software, as shown in Figure 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, however systems are emerging such as open-channel SSDs [12] that give more control to software drivers.

The interactions between these layers can be hard to predict, as they are designed for fast best-case performance, rather than predictability 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. 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 [13].

III. 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 an NVMe SSD to user-space applications as a basic character device. Using a character device has the advantage of conforming to the standard model of device nodes being accessible through the VFS, while removing complex block layer features such as the I/O scheduler, request queueing mechanisms, page cache, and asynchronous requests.

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 instead of a `/dev/nvmeXnX` block device node 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 against the standard Linux storage stack in Figure 2.

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' functions. This requires transfers to be aligned to the block structure of the underlying storage device, for example, a transfer size must be a multiple of 4096 bytes if that is the block size used. Each transfer is completed atomically and sequentially, with system calls blocking as data is transferred, after which control is returned to the calling application.

A. Low-level Operation

When a read or write system call is triggered from user-space on `/dev/chardiskX`, control is passed through the

VFS to the CharIO driver, which creates and sends requests on to the NVMe driver layer for processing. Any large transfers are split into 4MiB chunks, in order to remain compatible with existing NVMe and kernel Direct Memory Access (DMA) functions. Seek system calls work the same as for any block device, changing the current file pointer appropriately, but with the caveat that the seek must be aligned to the start of a storage block.

Once control is passed to the NVMe driver, the pages of memory specified by the user are set up for DMA transfer. This involves pinning the pages in memory 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.

When these DMA preparations are complete, the request is split into 32-block chunks to comply with the format of NVMe commands. A command structure is then created for each portion of the transfer, and each is sent sequentially to the storage device. The driver takes advantage of the hardware command queues present on the storage device, sending each new command as soon as the previous one has been accepted, rather than waiting for the command to be complete. This reduces the time spent waiting for the device, while also eliminating the need for software queues in the kernel. The completion of each command is indicated by an interrupt, which is forwarded through the PCIe driver to the CharIO NVMe layer. Once the overall transfer has completed, control is returned to the application that originally made the system call.

1) Physical Memory Addressing: An additional method of reading and writing data with CharIO is through using a physically-addressed memory location, which may be within or outside the memory area managed by the Linux kernel. To accept physical addresses, the CharIO device node uses custom *ioctl* read and write commands that relay the start block number and the transfer size to the kernel driver. In contrast, the standard Linux virtual file system read and write functions operate using the virtual address space of the calling process, and do not allow I/O operations that bypass the page cache to access addresses mapped outside of paged system memory.

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

2) Potential Driver Enhancements: 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. As the module has full control over memory allocations, it would be possible to set up DMA regions ahead of time and re-use them across requests, further reducing the amount of unpredictable activity involved in I/O operations. This would, however, reduce the flexibility of individual transfer buffer locations, cause a small amount more memory to be held active while the driver is loaded, and mean further departures from the standard Linux kernel methods of handling memory.

B. User-space Library

In addition to accessing CharIO devices directly through `/dev/chardiskX` device nodes, a simple user-space pseudo-file system has been developed, allowing more-structured access to the storage. To maintain the ideas 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.

IV. Evaluation

To evaluate the CharIO driver, we set up an experimental platform around an Avnet Zynq Mini-ITX development board [14]. This contains a Xilinx Zynq-7000 system-on-chip with 1GiB of DDR3 SDRAM. The Zynq contains a dual-core 800MHz ARM Cortex-A9 CPU [15], 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 [17] applied.

A custom profiling timer component was developed for high-accuracy, unobtrusive timing of events from the FPGA logic. 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 are 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.

A. Storage System Performance

We performed a series of experiments 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,

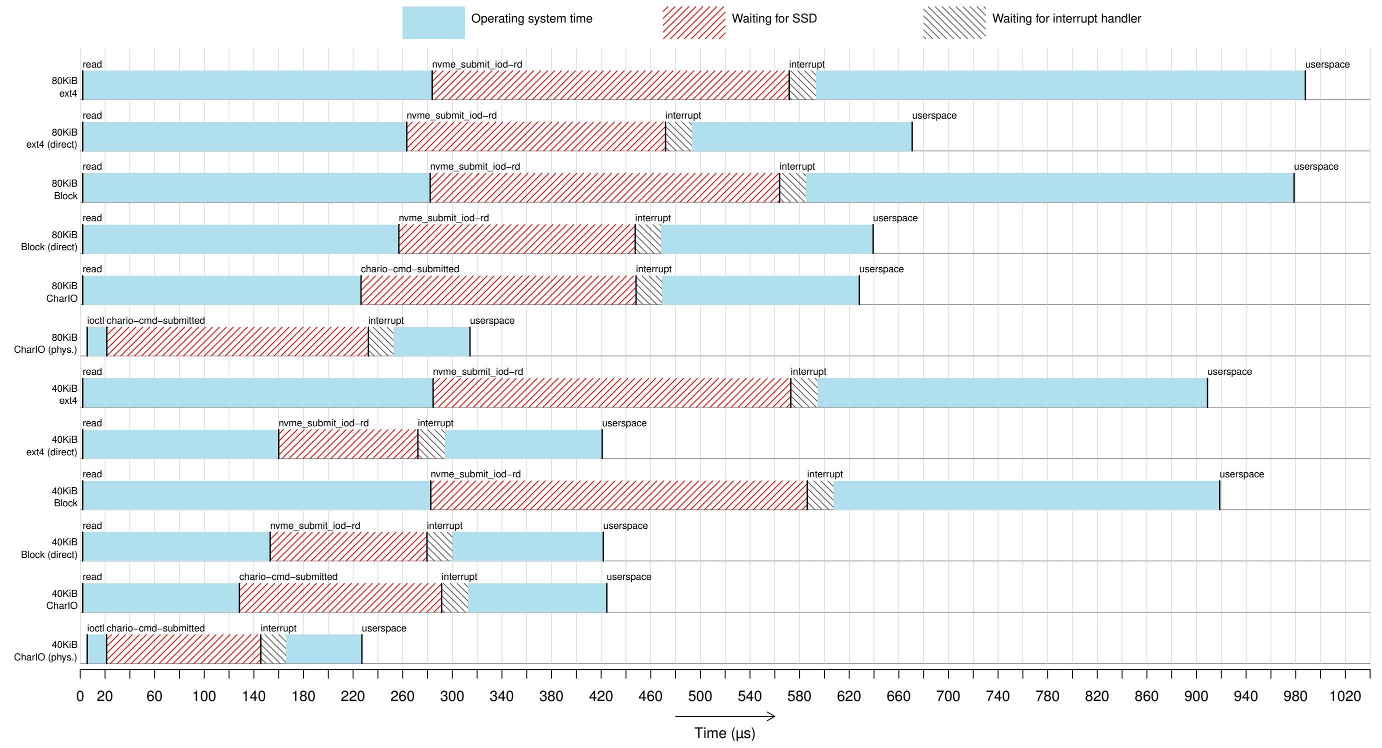

Fig. 3. Operating system and hardware latency measured for 40KiB and 80KiB reads from ext4, block device and CharIO storage

TABLE I
COMPARISON OF STORAGE ACCESS TRANSFER SPEEDS AND CPU USAGES

	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

we performed simple periodic file read and write operations 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 to a buffer in DDR outside of Linux's memory space.

The results displayed in Figure 3 show the mean times across 10,000 experimental runs, taken when performing sequential read operations with transfer sizes of 40KiB and 80KiB. Timing measurements are plotted for when: the system call is entered, the I/O command is submitted to the SSD, the hardware interrupt is triggered, the kernel begins handling the interrupt, and the user-space application is resumed.

The results show that CharIO spends less time performing computation in the kernel than ext4 or the block device, with a further significant reduction measured when using the physical addressing mode.

The results also highlight the efficiency of 'direct I/O' compared to standard cached file transfers, with no time wasted copying data via the cache. There is also little difference between the 40KiB and 80KiB transfers when using the page cache, due to the block layer caching more data than is actually requested for the smaller transfers, demonstrating the unpredictable nature of standard file access methods.

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 is extremely unpredictable, even when using direct I/O. This was found to be more exaggerated when writing, due to the file system sporadically writing additional metadata, as well as creating extra activity with features such as journaling. This behaviour meant calculating meaningful average results was impossible when testing larger transfers, as the number of commands sent to the SSD varied so greatly.

B. 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 [13], a range of block sizes were tested.

Results from these experiments are shown in Table I, 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 transfers, CharIO outperforms direct I/O speeds for both other methods, 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 I/O layer can have a considerable impact on storage performance in an embedded system due to high CPU usage.

V. Conclusion and Further 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 access is preferable in these systems.

CharIO, our proposed interface 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 these improvements come at the cost of some conveniences and features offered by traditional file systems, they may not be required for many real-time and embedded tasks, where a simpler method of accessing storage can be more appropriate.

As well as further analysis work and expanding the functionality and compatibility, one major area for further work is to examine hardware acceleration of storage access functionality, which could be an effective way to increase performance. Hardware support may include constructing commands to be sent to the SSD controller, managing the storage buffer memory, and monitoring of dirty blocks for flush operations. This could lead to similar behaviour as a standard cache controller, but extended to a large storage device, effectively with block-sized cache lines, DDR memory as the cache storage, and backed by an SSD.

References

[1] "FreeRTOS," online: http://www.freertos.org.

[2] "RTEMS Real Time Operating System," online: https://www.rtems.org.

[3] 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.

[4] E. Zadok, V. Tarasov, and P. Sehgal, "The case for specialized file systems, or, fighting file system obesity," *;login: The USENIX Magazine*, Feb. 2010.

[5] 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.

[6] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 4th ed. Addison Wesley, 2009.

[7] 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.

[8] 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: USENIX Association, Feb. 2017, pp. 329–343.

[9] 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.

[10] 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.

[11] A. A. Mendon, A. G. Schmidt, and R. Sass, "A hardware filesystem implementation with multidisk support," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[12] 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, Feb. 2017, pp. 359–374.

[13] 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.

[14] "Zynq Mini-ITX Board," online: http://zedboard.org/product/mini-itx-board.

[15] "Zynq-7000 All Programmable SoC," online: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[16] "Intel SSD 750 Series," online: http://www.intel.co.uk/content/www/uk/en/solid-state-drives/solid-state-drives-750-series.html.

[17] "Real-Time Linux," online: https://wiki.linuxfoundation.org/realtime/start.