Comparison of File Systems in RTEMS

Udit Kumar Agarwal
Netaji Subhas Institute of Technology
New Delhi, India
dev.madaari@gmail.com

Vara Punit Ashokbhai
Bangalore, India
punitvara@gmail.com

Gedare Bloom
Howard University
Washington, DC, USA
gedare.bloom@howard.edu

Christian Mauderer
embedded brains GmbH
Pucheim, Germany
christian.mauderer@
embedded-brains.de

Joel Sherrill
OAR Corp.
Huntsville, AL, USA
joel.sherrill@oarcorp.com

ABSTRACT

Real-Time Executive for Multiprocessor Systems (RTEMS) is an open-source real-time operating system (RTOS) that is widely used in commercial and free embedded applications with notable adoption in space flight software and scientific instrument control for space science and high energy physics. RTEMS has rich support for POSIX environments and supports multiple POSIX and BSD file systems, along with some custom file systems designed specifically to meet the needs of real-time and embedded applications storage and retrieval of data. The range of file systems available in RTEMS motivates this study that investigates the salient features of each file system to help identify strengths and weaknesses with respect to application requirements and constraints. In this paper, we provide a comparison of the available RTEMS file systems and present some performance benchmarking results.

CCS CONCEPTS

• **Computer systems organization → Real-time operating systems**; **Embedded software**; • **Software and its engineering → File systems management**;

KEYWORDS

RTEMS, Filesystems, Benchmarking

ACM Reference Format:
Udit Kumar Agarwal, Vara Punit Ashokbhai, Gedare Bloom, Christian Mauderer, and Joel Sherrill. 2018. Comparison of File Systems in RTEMS. In *Proceedings of EWiLi Embedded Operating System Workshop (EWiLi'18).* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Prior to the 1990s embedded applications were self-contained by design to include all the necessary code and storage to read and write any persistent data such as application configuration settings and file-like data objects. These applications used custom data formats and hardware-specific code for interfacing with input/output (I/O) storage peripherals. Reuse of code and formats across multiple applications, or even several generations of the same application with new hardware, was not well-supported and resulted in lost development time due to code redevelopment and debugging. These losses were mitigated by the advent of file system support in embedded operating systems.

A file system encapsulates data storage and I/O access from the abstract file view of application logic. Encapsulation leads to better software engineering practices such as code reuse, modular testing, and portable software. A modern real-time operating system (RTOS) supports file systems as an important and necessary service for applications or even RTOS storage requirements. For example, RTOS configuration settings may be stored in files that are available after the file system services are initialized by the kernel.

File system design and implementation in an embedded operating system has distinct challenges with respect to general-purpose operating systems. In particular, an embedded OS is characterized by having few to no human-user interfaces, remote or automatic management, possibility of being rebooted by failure mode or fault tolerance mechanisms, and constrained in availability of time, space, and power resources. These characteristics motivate a strong requirement that file systems in embedded and RTOS operate properly in boundary and extreme cases, gracefully handle failures, and can be made resource-efficient depending on the constraints of the specific embedded application.

In this paper, we analyze the available file system support in the Real-Time Executive for Multiprocessor Systems (RTEMS) RTOS with a focus on the discriminating features of each file system. In Section 3, we describe each file system and provide a qualitative comparison of their strengths and weaknesses with a focus on how file system features intend to meet specific application requirements. Section 4 presents file system benchmarking results to give a notion of the performance characteristics for some of the RTEMS file systems. The contribution of this paper is a comprehensive analysis and comparison of file systems in a widely-used open-source RTOS.

2 OVERVIEW OF RTEMS FILE SYSTEMS

The file system stack in RTEMS has evolved to include two intertwined stack implementations: the native stack and the `libbsd` stack. The native stack is a layered architecture for file system design and implementation that separates the application programming interface (API) of files from the storage media. The `libbsd` stack borrows code from the FreeBSD file system stack to supplement the native stack with additional support for networking, USB, and SD/MMC subsystems. In this section, we discuss the integration of networking stacks with file systems in RTEMS and describe the

EWiLi'18, 4 October 2018, Torino, Italy
Copyright held by the owner/author(s)..

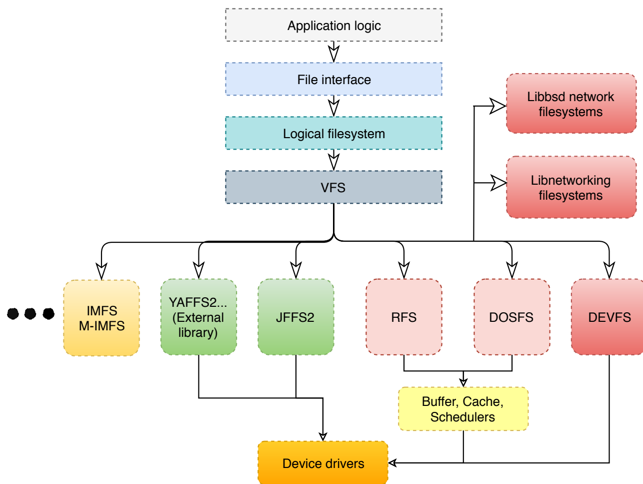

Figure 1: Layered File System Architecture in RTEMS.

layered file system architecture as adopted by RTEMS. See Bloom and Sherrill [1] for a brief introduction to the RTEMS kernel.

2.1 Networking and `libbsd` Support

RTEMS has support for three distinct networking stacks to support network file systems and network file type objects. The most mature of these stacks is the `libbsd` stack, which aims to track the FreeBSD development head's networking stack and selected device drivers. The other two stacks are lwIP and the legacy `libnetworking`, which is a *clone-and-own* copy of an old FreeBSD networking stack. The `libbsd` stack is maintained as a library in a separate repository that can be combined with RTEMS during compile-time. In order to make `libbsd` possible, some of the FreeBSD kernel interfaces have been re-implemented in RTEMS. The end result is that FreeBSD support for device auto-configuration, the network stack including IPv6 and IPsec, the USB stack, and the SD/MMC card stack are all available for use in RTEMS-based applications.

2.2 File System Organization

As shown in Figure 1, RTEMS follows a traditional layered architecture that forms a hierarchy consisting of the file API, the logical file system, a virtual filesystem switch (VFS), the physical file systems, a block cache and buffer layer, and finally the device drivers that access the storage media. In the following we briefly describe each of the components of the overall file system hierarchical architecture as they relate to RTEMS.

2.2.1 File APIs. The ubiquitous file interface provides a consistent and meaningful view of data to application logic. Two basic file interfaces are supported by RTEMS. The first is the POSIX/libc file, available through both the native and `libbsd` stacks, that provides support for POSIX and C/C++ applications, as well as other languages that are compatible with either POSIX or libc. Second are the networking file objects, which include pipes and sockets compatible with BSD network file types.

2.2.2 Logical File System. The logical file system is the view of file-based data that is provided across the interface between user applications/libraries and kernel code. Access to the logical file system is therefore part of the system call kernel interface and is highly dependent on kernel design. This layer maintains state associated with user space, such as the system- and process-open files and process file descriptors. It is at this layer that the kernel begins the translation of the application's file requests to the physical storage behind that file by way of path evaluation, directory traversal, access control permission enforcement, and translating file names into kernel data structures.

The logical file system in RTEMS is implemented as part of the `cpukit/libcsupport` code base. `rtems_libio_iops` is the table of open files, and `rtems_filesystem_location_info_t` relates paths to mountpoints. `rtems_filesystem_eval_path_generic` function parses a path and traverses the filesystem, and file permissions are enforced by `rtems_filesystem_check_access`.

2.2.3 Virtual Filesystem Switch (VFS). The VFS layer is a stable kernel interface between the logical file system that userspace sees and the various file system implementations that manage mounted storage media. VFS is an object-oriented approach that simplifies portable file system implementation.

Most of the VFS in RTEMS is accessed through the `libio.h` file. The libio defines the `rtems_filesystem_mount_table` linked list of mounted filesystems, which is analogous to a superblock in the Linux VFS. The `rtems_filesystem_operations_table` is the set of functions called on a specific filesystem, thus it is approximately a combination of the superblock, inode, and dentry ops tables in Linux. The `rtems_filesystem_file_handlers_r` function table is basically the `file_operations` table of Linux' VFS.

2.2.4 Physical File Systems. Often simply referred to as the File System Implementations, the Physical File Systems are the software components that control use of a physical storage media. For non-volatile storage, this layer makes use of the storage media itself to store a persistent subset of its data. This layer includes a wide variety of file system implementations, and is the layer most often associated with file systems by name, such as ext3 or FAT. The physical file system code translates access requests to the physical address space of the storage media, and manages the space allocated to the file system on the storage device.

The physical file systems available in RTEMS address storage requirements for a variety of media, including memory, block, flash, and network devices. Two file systems are explicitly designed to work from a heap memory region: In Memory File System (IMFS) and Mini In Memory File System (Mini-IMFS). Typically, these file systems provide a small, memory-resident root file system to facilitate mounting other file systems and to ensure a file system is available even if storage devices are not connected. Mini-IMFS is a stripped-down version of IMFS aiming toward lower memory overhead. The two primary block device file systems in RTEMS are DOSFS and the RTEMS File System (RFS). DOSFS is compatible with FAT12, FAT16, and FAT32. RFS is a customized file system aiming toward low overhead and stability for extended use in long-lived embedded systems. RTEMS includes a stable port of JFFSv2 with journaling support. YAFFS2 is also available, but this file system imposes a GPL or commercial license to remove the copyleft so it

Table 1: Feature Comparison of RTEMS File Systems. The two GPLv2 licenses have either link-time exceptions or commercial versions available.

Name	Type	License	Key Characteristic(s)
IMFS	RAM	RTEMS	POSIX features, RAM based
Mini-IMFS	RAM	RTEMS	Reduced footprint IMFS
DOSFS	Block	RTEMS	Focus on compatibility
RFS	Block	RTEMS	Predictable, non-power of 2
JFFSv2	Flash	GPLv2*	Log structured, NOR
YAFFS	Flash	GPLv2*	Log structured, NAND/NOR

is not suitable for all users and is not directly integrated with the RTEMS code base. NFSv2 and TFTP/FTP file systems are available with suitable networking stack support.

2.2.5 Block Buffers and Cache. Block file systems and block devices interact through the `libblock` cache. This cache improves block I/O throughput using multi-threaded readers/writers using typically a dedicated worker thread per block device to coordinate and synchronize the physical access, while multiple threads may allow reads and writes to pend in parallel. A special feature of this cache is the use of varying block sizes, which can improve memory utilization compared with fixed block sizes. The block cache interfaces with device drivers using read, write, and control commands.

2.2.6 Device Drivers. RTEMS has a hodge-podge collection of device drivers without any consistent or uniform device driver framework. Over time, useful frameworks from other operating systems have been ported to RTEMS so that the drivers may be reused. Such frameworks include the `libbsd` as a coarse-grained framework to support many FreeBSD drivers, and the I^2C/SPI framework that provides a Linux-compatible API.

3 COMPARISON OF RTEMS FILE SYSTEMS

RTEMS has significant support for myriad file systems which forms an infrastructure to support different functionalities such as mountable file systems, hard and softlinks to files and directories to access them quickly. The filesystems support RAM, non-volatile memory, physical disks, and network storage. RTEMS has POSIX compliant APIs which support encourage application portability. Robustly implemented file systems can use structures able to adapt to contemporary data storage devices. Internally, RTEMS device drivers follow a traditional API interface based on initialize(), open(), read(), write(), and ioctl() interfaces. The RTEMS file system and system call interfaces are layered on top of the pluggable device driver interface. Prominent features of different non-networked file systems are discussed in following sections. Specifically not discussed are the RTEMS Network File System (NFS), FTP File System (FTPFS), and TFTP File System (TFTPFS).

3.1 IMFS: In Memory File System

The In-Memory File System (IMFS) was the first file system implemented for RTEMS. It supports the complete complement of POSIX file system features including files, directories, device nodes, and pipes. This was a critical design consideration as the IMFS was first used to debug the file system stack through the C Library and System Call interfaces. A key characteristic of this file system is that disk blocks are obtained from dynamically allocated memory (e.g., via malloc()). Unless configured to have no file system support, an instance of the IMFS is created during RTEMS initialization for use as the root file system. An interesting feature is that the file contents for an IMFS instance can be populated from an uncompressed tar file located in read-only memory (ROM). Until the contents of a file is modified, it will be accessed from ROM. This provides an effective way to have a file system with static content without consuming RAM. Associated with each IMFS entity including mounted file systems is a `jnode`. A `jnode` instance has all necessary information available in `struct IMFS_jnode_tt` such as user id, group id, when was file last accessed, modified or status changed. Identification of every file system node can be done with `rtems_filesystem_location_info_t`.

3.2 Mini-IMFS: Mini In Memory File System

The Mini-IMFS is a reduced functionality variation of the IMFS. It does not support symbolic links, mounting/unmounting, or changing the group/owner of a file. This is useful to save memory for applications which do not require the full functionality of the IMFS.

3.3 DOSFS: DOS or FAT File System

The FAT file system was the first non-volatile storage block-based file system to be supported by RTEMS. It currently supports FAT12, FAT16, and FAT32 formats including long file name (LFN) support. RTEMS includes both a command line and programmatic interface to format FAT file system instances. This file system implementation has been deployed in data logging systems and effort has been invested to optimize it to meet the requirements of these systems.

3.4 RFS: RTEMS File System

The RTEMS File System (RFS) is a full featured file system with POSIX semantics designed to meet requirements for predictable memory usage and performance. It is a block-based file system and is layered on top of the RTEMS Block Driver Interface and thus supports a variety of storage media. It also has the unique feature of supporting block sizes which are not powers of two. This feature meets the requirement of space based applications which use non-volatite storage with error-correcting codes (ECC). Being initially developed for use in space based applications, it is designed to accomodate CPUs with low clock speeds and relatively small amounts of RAM (e.g. 20 Mhz and 2MB).

3.5 JFFS2: Journalling Flash File System v2

The Journaling Flash File System (JFFS) is a log-structured file system that supports NAND and NOR flash devices. It provides a file system directly on the flash memory rather than interfacing through a block device interface. It includes a garbage collector which can perform wear-leveling. JFFS2 also supports compression of data stored thus allowing a trade-off between storage used and execution time. However, some data does not lend itself to further compression (e.g. MP3 or JPEG). Having this as a configurable feature allows the designer to tailor the solution for the system.

3.6 YAFFS: Yet Another Flash File System

The Yet Another Flash File System (YAFFS) is a log-structured file system which was designed with data integrity and performance as goals. It supports NAND and NOR flash devices and operates directly on the flash memory. YAFFS is designed to spread writes across the disk serially from the list of erased blocks. This tends to naturally perform wear-leveling. YAFFS is dual-licensed and a license for commercial use must be obtained to use it without copyleft obligations.

4 RTEMS FILE SYSTEM BENCHMARKING

The platform used for benchmarking is BeagleBone Black rev. B6 along with RTEMS version 5 pre-release.[1] In the following we introduce the benchmarking procedure used before presenting the impact of different read/write methods by tuning ioengine and block size on specific file systems such as RFS, IMFS, and DOS. Further, RTEMS supported flash file systems are contrasted against each other, and we present the effect of various tuning parameters on the bandwidth that was recorded. We end this section with a statistical comparison of DOSFS and RFS on RAM-disk and different storage media.

4.1 Benchmarking process

The benchmark used was Flexible I/O (FIO) tester rev. 3.6 which has been ported to RTEMS.[2] Build instructions for RTEMS are available[3]. FIO provides what it refers to as ioengines that implement multiple file access methodologies. An ioengine is selected as part of configuring FIO for a particular run. Available ioengines include: *sync*, which uses native read(), write() and lseek() calls for reading, writing or positioning a file; *psync* which uses pread() and pwrite(); and *vsync* which uses readv() and writev() system calls. Unless stated, by default the sync ioengine was used for benchmarking. The benchmarking process was straightforward: setup the benchmarking environment first and FIO's job configuration settings. Benchmarking environment requires setting up the desired file system along with the target device, which in our case was mostly done with RAM-disk. All tests, including those for the flash file systems, were performed on a RAM-disk with a fixed device size along with constant CPU stack and cache size. To be precise, libblock cache size used was 10 KiB with 1 KiB as individual libblock buffer size and CPU stack minimum size was 256 KiB. As we expect users will tend to use default values, we present the results obtained with defaults (e.g., block size) as the primary results while we show variation due to changing defaults as secondary results. For consistency and reproducibility of the results, we have made the benchmark test configuration available online.[4]

Evaluating file system performance on a RAM disk has two advantages: First, typical storage media like SD-cards and USB storage sticks are short lived consumer articles that may reduce reproducibility. Second, flash based media performance tends to degrade over time so one would need to implement a mechanism to check whether the medium is still performant while running benchmarks. A RAM disk does not suffer from these two problems, and it can yield useful results to inform file system designers a long as the same timing settings are used consistently with the RAM as would the target media.

[1] commit id: cf811a4eb2 and RTEMS Source Builder commit id: 25f4db09c8
[2] source at https://github.com/madaari/fio/tree/paper
[3] uditagarwal.in/index.php/2018/08/02/benchmarking-rtems-filesystems-using-fio
[4] https://github.com/madaari/fio/blob/paper/os/rtems/rtems-init.c

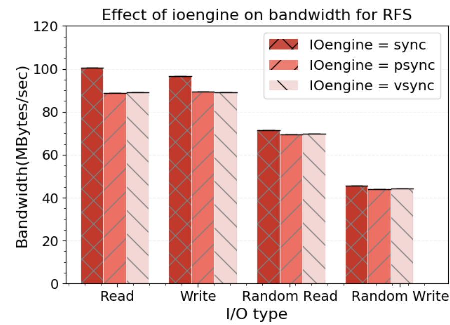

Figure 2: Effect of IOengine on bandwidth for RFS.

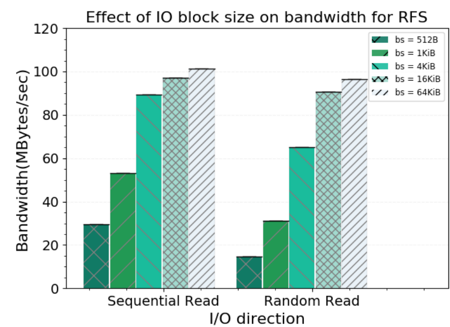

Figure 3: Effect of IO block size on bandwidth for RFS.

4.2 RFS

Figure 2 shows better performance of RFS with the sync ioengine, which indicates that RTEMS native read()/write() calls are faster then the POSIX implementation of pread/readv or pwrite/writev. This is not surprising as the default implementation of readv/writev is a loop which makes multiple calls to read/write. Read and write bandwidth are similar because of using a RAM-disk. The difference in sequential read and random read is due to the finite libblock cache size. Figure 3 shows correlation between the IO block size and the bandwidth. At smaller block sizes, a gradual increase in block size significantly reduces the number of read/write operations, thus a large increase in bandwidth. At larger block sizes, the latter factor gets dominated and time to transfer a block of data compensates for the change in the number of IO operations.

4.3 DOSFS

Similar to RFS, Figure 4 shows the variance of bandwidth with IO block sizes for DOSFS. Speed here is well below the mark where the second factor (i.e., the time required to complete an IO request) dominates, and thus bandwidth increases with IO block size. Tests for DOSFS were all performed with block size 512B (default) and

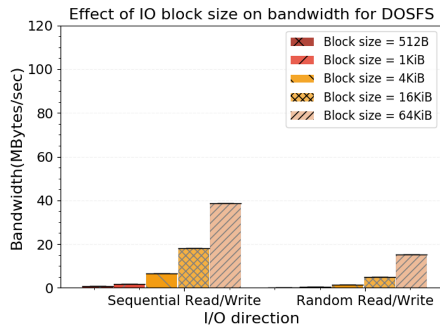

Figure 4: Effect of IO block size on bandwidth for DOSFS.

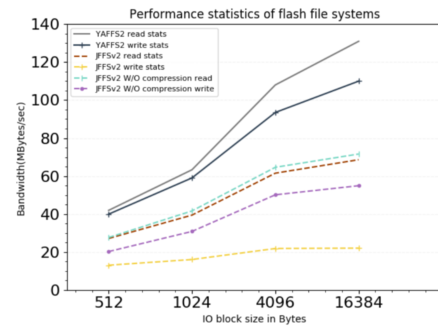

Figure 6: Performance statistics of flash file systems.

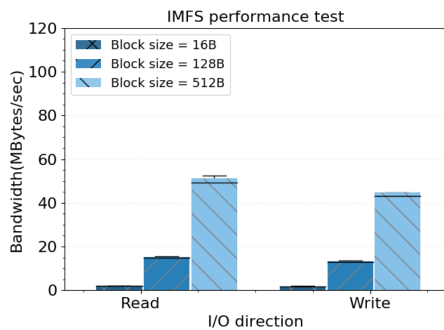

Figure 5: Effect of IO block size on bandwidth for IMFS.

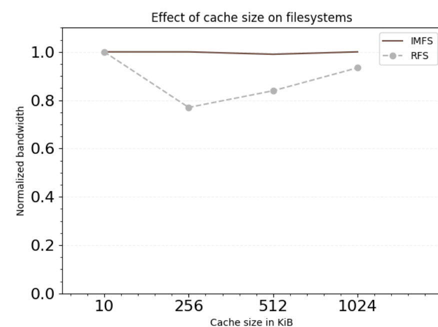

Figure 7: Effect of cache size on bandwidth of file systems.

thus that explains lower bandwidth than from RFS (which uses 1 KiB block sizes as default).

4.4 IMFS and Mini-IMFS

Performance for IMFS and Mini-IMFS are similar, which is expected since Mini-IMFS is just IMFS minus some functionality to have a low memory overhead. Figure 5 shows the IMFS performance statistics with a file system block size up to 512B. IMFS uses an i-node structure based on that of the original UNIX file systems. So as the block size goes up, one can have more and more blocks in the maximum file size. Further, increasing the block size (default: 128B) leads to more memory wastage due to internal fragmentation. Thus, depending on the application, one has to either compromise the maximum file size or the amount of memory wasted.

4.5 Contrasting flash file systems

Benchmarking of flash file systems was performed with file system block size of 4 KiB for both file systems, which is shown in Figure 6. The flash device layer was simulated with a RAM-disk, primarily to ease comparison with the RFS and FATFS that were benchmarked using a RAM-disk. Using a RAM-disk also eliminated any limits or variances introduced by the flash device driver, flash translation layer, or hardware itself. Here, YAFFS2 performs better then JFFS2 due to more efficient garbage collection along with no data compression. JFFS2 on the other hand supports data compression by default, and is thus a better fit to small partitions of NAND especially if there are text-only files. However, data does not always compress very well, and enabling compression for some data formats hurts performance without gaining storage space. For example, an embedded web server containing a lot of already compressed images will not benefit from compression, but a system that writes sensor log files will use the available space a lot more effectively at the expense of spending CPU time for compression. Disabling compression has a noticeable benefit to JFFS2 write performance, and modest improvement to read throughput. It is possible these results may differ with a FLASH disk, as the computational overhead of compression could be hidden by moderately longer access times with respect to the RAM disk.

4.6 Effect of tuning parameters

The test in Figure 7 was performed with file system block size constant and IO block size equal to 4 KiB using IMFS and RFS. Here, normalized bandwidth is used to put focus more on how bandwidth changes rather than individual differences in bandwidth for both file systems. Normalized bandwidth was calculated by dividing the individual bandwidths with bandwidth at cache size 10 KiB for both file systems. IMFS uses standard heap allocator for allocating files (by malloc) and thus it does not have any dependency on libblock and cache. RFS on the other hand, is sensitive to block cache parameters. The initial drop in bandwidth is suspected to be due to overwhelming of L1 and L2 cache and thus hitting a 20 cycle memory access miss penalty. The size of L1 cache in this case is 32 KiB and that of L2 is 256 KiB on BeagleBone Black. Further

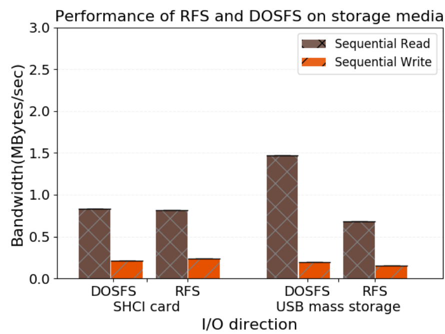

Figure 8: Performance of RFS and DOSFS on storage media.

increasing the libblock cache size results in an increased bandwidth as the number of cache misses decreases significantly.

The size of each file within a file system can be expressed as an integral multiple of file system block size. IO bandwidth is almost linearly proportional to file system block size (at least at smaller block sizes). Having large block sizes for a file system can lead to storage waste due to internal fragmentation when allocating smaller file sizes. A system with a small number of large files can efficiently use large block sizes, while a system with a large number of small files would more efficiently use the storage with a smaller block size.

Figure 8 shows performance of RFS and DOSFS with file system block size in both the cases to be same and equal to 512B. SDHC card used was a 16 GiB variant made by Lexar and USB stick was of 4 GiB by an unknown OEM. In determining the performance of a file system on a storage media, some factors are independent of the file system such as device parameters (e.g., USB/SDHC card manufacturer, flash implementation, partition size, etc.) and device driver implementation (e.g., version/source of device driver, efficiency of driver, etc.). Thus, results in Figure 7 should not be used to contrast RFS/DOSFS performance across different storage media but only for the given platform.

5 RELATED WORK

The most closely related work in characterizing and comparing performance of file systems in embedded operating systems focuses on flash file systems [2–8]. Although flash disks are the dominant form of secondary storage in embedded systems, non-flash file systems are also widely used and deserve attention in RTOS design, implementation, and optimization. We are not aware of any attempts to characterize the performance of file systems across the breadth of an RTOS. This is most likely to be performed as part of file system selection and tuning for a specific embedded system deployment.

6 CONCLUSION

In this paper, we reviewed the design and implementation of the file system stacks available in the RTEMS open-source RTOS, compared the features of the file system implementations qualitatively, and presented quantitative benchmarking results of commonly used RTEMS file systems for RAM disk file management. Future work can further characterize the performance of RTEMS file systems using alternate metrics, benchmarks, storage devices, and microcontroller platforms. The outcomes of performance characterization should be used to validate that file system design goals are met, and to guide implementation tuning to optimize for design constraints. This work was done on a single target board and it appeared that CPU cache and bus bandwidth were limiting factors. It is desirable to define a benchmarking methodology which would allow embedded systems designers to evaluate the impact of target hardware, file system, and block size selection for their deployment. We are also interested in determining fair methods to compare different implementations of the same physical file system, for example FAT32 or JFFSv2, across multiple operating systems.

ACKNOWLEDGMENTS

The authors thank the Google Summer of Code Program for supporting the students who have worked on this effort. Gedare Bloom is supported in part by the National Science Foundation under Grant No. CNS-1646317 and the U.S. Department of Homeland Security under Grant Award Number 2017-ST-062-000003.

REFERENCES

[1] Gedare Bloom and Joel Sherrill. 2014. Scheduling and Thread Management with RTEMS. *SIGBED Rev.* 11, 1 (Feb. 2014), 20–25. DOI: http://dx.doi.org/10.1145/2597457.2597459

[2] Seung-Ho Lim, Sung-Hoon Baek, Joo-Young Hwang, and Kyu-Ho Park. 2006. Write Back Routine for JFFS2 Efficient I/O. In *Proceedings of the 2006 International Conference on Embedded and Ubiquitous Computing (EUC'06)*. Springer-Verlag, Berlin, Heidelberg, 795–804. DOI: http://dx.doi.org/10.1007/11802167_80

[3] Pierre Olivier, Jalil Boukhobza, and Eric Senn. 2012. Micro-benchmarking Flash Memory File-System Wear Leveling and Garbage Collection: A Focus on Initial State Impact. In *Proceedings of the 2012 IEEE 15th International Conference on Computational Science and Engineering (CSE '12)*. IEEE Computer Society, Washington, DC, USA, 437–444. DOI: http://dx.doi.org/10.1109/ICCSE.2012.67

[4] Pierre Olivier, Jalil Boukhobza, and Eric Senn. 2012. On Benchmarking Embedded Linux Flash File Systems. *SIGBED Rev.* 9, 2 (June 2012), 43–47. DOI: http://dx.doi.org/10.1145/2318836.2318844

[5] Pierre Olivier, Jalil Boukhobza, and Eric Senn. 2014. Flashmon V2: Monitoring Raw NAND Flash Memory I/O Requests on Embedded Linux. *SIGBED Rev.* 11, 1 (Feb. 2014), 38–43. DOI: http://dx.doi.org/10.1145/2597457.2597462

[6] Pierre Olivier, Jalil Boukhobza, and Eric Senn. 2015. Revisiting Read-ahead Efficiency for Raw NAND Flash Storage in Embedded Linux. *SIGBED Rev.* 11, 4 (Jan. 2015), 43–48. DOI: http://dx.doi.org/10.1145/2724942.2724949

[7] Pierre Olivier, Jalil Boukhobza, Eric Senn, and Hamza Ouarnoughi. 2016. A Methodology for Estimating Performance and Power Consumption of Embedded Flash File Systems. *ACM Trans. Embed. Comput. Syst.* 15, 4 (Aug. 2016), 79:1–79:25. DOI: http://dx.doi.org/10.1145/2903139

[8] Pierre Olivier, Jalil Boukhobza, Mathieu Soula, Michelle Le Grand, Ismat Chaib Draa, and Eric Senn. 2014. A Tracing Toolset for Embedded Linux Flash File System. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '14)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 153–158. DOI: http://dx.doi.org/10.4108/icst.valuetools.2014.258179