NanoFS: a hardware-oriented file system

P. Ruiz-de-Clavijo, E. Ostúa, J. Juan, M.J. Bellido, J. Viejo and D. Guerrero

NanoFS is a novel file system for embedded systems and storage-class memories (like flash) and is specially designed to be directly implemented in hardware. NanoFS is based on an original internal layout intended to achieve an optimal hardware implementation of the file system's file lookup and data fetch operations. File system spe-cification on a sample reader module completely implemented in a pro-grammable device is introduced.

Introduction: State-of-the-art electronic embedded systems extensively use storage-class memories as main storage elements due to the maturity of the technology, low cost and high capacities; a good example being flash memories used in secure digital (SD) cards typically found in consumer electronics. In most cases, especially in high-level applications, the storage resources of these devices are used through the file system abstraction like in most computer systems.

There exist plenty of file system specifications, such as the file allocation table (FAT) file system [1] which is widely used in embedded systems because of its ubiquity and availability in most computer systems. However, FAT and most existing file systems were not specifically designed for embedded systems or storage-class memories, but to supply the requirements of desktop computer using magnetic disks. Nevertheless, there have been some specific developments targeting the embedded systems market. A micro controller implementation of FAT for embedded systems has been proposed in [2] with limited success. UUFS is a low-cost flash memory file system implemented in C language [3]. A file system specification optimised for page-mode flash technologies is described in [4]. WIPS is a file system optimised for storage-class memories with snapshot support implemented in the Linux kernel [5]. In addition, HCC Embedded Inc. offers a commercial range of embedded file systems aimed at real-time operating systems and micro controllers [6].

On the other hand, some initiatives have made progress in implementing file system operations in hardware using field programmable gate array (FPGA) technology like in [7], where the core functionality of a UNIX-like file system is implemented in hardware, although this development is not specifically targeted to embedded systems but to accelerate file operations in high performance computing systems.

This Letter introduces NanoFS: a novel file system specifically designed for embedded systems and consumer-grade (low-cost) storage-class memory devices. Contrary to the approaches mentioned above, this specification is specially tailored to allow an efficient hardware implementation, so that arbitrary files can be read by a pure-hardware module with a little hardware footprint. In embedded systems, the ability to access the file system directly from the system's hardware even before the systems software is booted, or in the absence of a soft-ware stack, opens the door to a number of interesting applications like easy boot loader implementation or flexible configuration storage. This Letter also describes the implementation in an FPGA chip of an all-hardware NanoFS reader module for SD high capacity (SDHC) memory cards. The implementation results are compared with the existing file system hardware implementations.

Nano file system: As with other file systems, NanoFS divides the storage device into blocks of a configurable size (the file system's block size). Despite its simplicity, NanoFS provides complete function-ality typically found in UNIX-like operating systems: files, directories, fifos, permissions, symbolic links and arbitrary metadata. The key feature of the NanoFS is that it organises files and directories as a set of linked lists using the file system's blocks as nodes. Each node includes a 32 bits pointer that is a reference to the next node/block. There are three types of nodes: the superblock node; directory nodes, which hold the file system structure and file metadata; and data nodes that hold file contents.

The block layout of the NanoFS is shown in Fig. 1. The example uses a block size of 512 bytes, and contains two files in the root directory. The superblock is the first block of the file system. It contains critical information like the magic number (mn), the block size (bs), the revision number (rv) and the file system's size (fs). It also provides extra room for additional metadata and future extensions. The root pointer (rop) and the free pointer (frp) in the superblock point to the root directory node and to the list of free blocks, respectively.

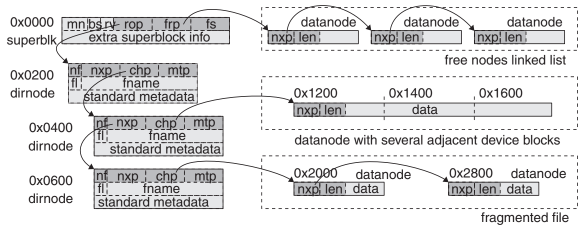

Fig. 1 *Nano filesystem internal layout*

In a directory node, the next pointer (nxp) points to the next directory node in the same directory tree level. A directory node may refer to a subdirectory entry or a file entry. In the latter, the child pointer (chp) points to the first data node of the file; in the former it points to the first directory node of the subdirectory. The entry type is in the node flags field (nf). File name (fname) and its length in bytes (*fl*) is also stored together with standard metadata (owner, modification time and so on). Additional metadata can be included in dedicated blocks through the metadata pointer (mtp). File contents are stored as a forward linked list of data nodes. This is implemented through the next pointer (nxp) and data length (len) fields in the data node. To opti-mise space, a data node can fill several adjacent blocks when there is no fragmentation, as can be seen in Fig. 1.

The main advantage of this layout over conventional file systems is that the file system can be traversed by reading a single data structure, a linked list of directory and/or data nodes, whereas most file systems need to read the file allocation structure (FAT, i-node table etc.) and the data blocks separately. This normally requires the allocation struc-tures to be copied in memory for performance reasons. With the NanoFS, when a block is fetched from the device, both data and allo-cation information are read in a single operation. As a consequence, both the lookup file and file data fetch operations are greatly simplified: the lookup file algorithm only needs to follow a linked list of directory nodes to locate a given file. Once the file is located, it can be completely read by following the linked list of data nodes that contain both the file data and the block pointers. From the hardware perspective, the implementation of the lookup and data fetch algorithms is greatly simplified since it does not need a complex allocation data structure, but only a few registers to store temporary pointers.

File system hardware implementations: In each particular implementation of the NanoFS, not every file system operation may be required. Thus, a modular design is proposed with optional modules for each file operation (lookup, read and write) sharing a common control system and data path. From the embedded systems perspective, the most useful operation to be implemented in hardware is the file read operation which may be necessary for system boot up and initialisation, while write operations can be handled by higher levels once the system is running. Therefore, the example shown here only implements the NanoFS read algorithm. Write capabilities can be added by adding the corresponding module to the architecture.

Fig. 2*a* shows the top-level hardware architecture of a NanoFS reader core. It consists of two main parts: the medium access unit (MAU) and the NanoFS control unit (NCU). The MAU controls the storage device that, in this implementation, is an SDHC card host operating in native or in SPI mode. The NCU implements the file system algorithms required by the application. For a reader module, three read modes are considered: first-file mode will read the first file stored in the file system; fixed-file mode will read a single file whose name is specified at design time; while any-file mode will lookup and read an arbitrary file whose name is provided through the *din* line. Fig. 2*b* depicts the flow control of the NanoFS read operation with four stages: (i) host initialisa-tion, (ii) NanoFS detection, (iii) lookup file and (iv) file reading. Each stage is controlled by a finite state machine that coordinates with the other stages by a dedicated handshake protocol. If only the first-file mode is needed, the optional lookup file stage (dashed lines in Fig. 2) can be removed to save hardware resources.

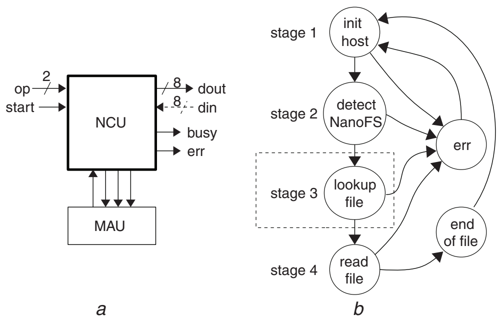

Fig. 2 *NanoFS reader cores*

a Top-level NanoFS architecture
b NanoFS flow control

Fig. 3 shows the internal structure of a NanoFS core reader for first-file mode with optional fixed-file mode (dashed lines) controlling an SDHC host unit. The structure consists of two main parts: a shared data path and a flow CU. Owing to the hardware-optimised NanoFS layout, the data path is very simple: *ptr32* is a simple SDHC host interface that includes a pointer to the current block, *next_ptr* point to the next block of the file or directory, *cnt32* is a block counter inside the file or directory and *byte_reg* is the data and status output register. In the flow CU, the master CU grants control to the slaves in turn to complete the requested operation.

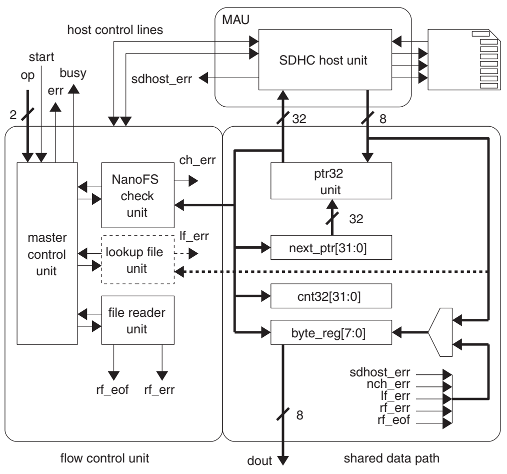

Fig. 3 *NanoFS reader for first-file or fixed-file mode with SDHC host unit*

Table 1: Resources utilisation on FGPA for NanoFS readers cores and MAU core

Core	Device	Slices FF	LUTs	Slices	Slices (%)
First-file	XC3S100E	126/1920	170/1920	120/960	12.5
Fixed-file	XC3S100E	147/1920	278/1920	192/960	20
MAU	XC3S100E	89/1920	246/1920	139/960	14
First-file	XC4VFX60	126/25280	202/50560	116/25280	0.5
Fixed-file	XC4VFX60	147/25280	338/50560	207/25280	0.8
MAU	XC4VFX60	89/25280	250/50560	142/25280	0.6

Results: Two versions of the core (first-file mode and fixed-file mode) have been implemented in two different FPGA chips from Xilinx: XC3S100E, a low-range device with 960 slices and XC4VFX60, a high-range device with 25 280 slices. The systems are integrated together with an MAU. The resource utilisation results are summarised in Table 1. The core results do not include the MAU resources which are listed separately. Both NanoFS cores easily fit in the smaller FPGA (26 and 34% slice utilisation, respectively, including the MAU). The impact on the bigger chip is negligible (1.1–1.4%). In comparison, the hardware-implemented functions of the general purpose UNIX-like file system in [7] use 1335 slices (5.3%) and 3 BRAMS (512 B block size) in an XC4VFX60 device.

Conclusion: Direct access to the file system from the hardware is of great interest to embedded systems (boot-up, efficiency and so on). File system operations can be completely and efficiently performed in hardware by using specific hardware-friendly file system designs like the NanoFS introduced in this Letter.

Acknowledgment: This work has been partially supported by the Ministerio de Ciencia e Innovación of the Spanish Government under project TEC2011-27936 (HIPERSYS) and by the European Regional Development Fund (ERDF).

11 June 2013
doi: 10.1049/el.2013.1961

P. Ruiz-de-Clavijo, E. Ostúa, J. Juan, M.J. Bellido, J. Viejo and D. Guerrero (*Departamento de Tecnología Electrónica, E.T.S. Ingeniería Informática, Universidad de Sevilla, Av. Reina Mercedes s/n, 41012 Sevilla, Spain*)

E-mail: paulino@dte.us.es

References

1 Microsoft Knowledge Base: 'Description of the FAT32 File System', Article 154997, Microsoft Corporation, 2004
2 Bannack, A., and Wolaniuk, G.B.: 'FAT 16/32 file system driver for Atmel AVR', 2004. Available at http://www.sourceforge.net/projects/fatdriveravr
3 'UFFS: an ultra low cost flash file system for embedded system' Available at https://sites.google.com/site/gouffs, accessed 1 June 2013
4 Ban, A., and Hasharon, R.: 'Flash file system optimized for page-mode flash technologies' United States Patent Number 5937425, 1999
5 Lee, E., Yoo, S., Jang, J., and Bahn, H.: 'WIPS: a write-in-place snapshot file system for storage-class memory', *IET Electron. Lett.*, 2012, **48**, (17), pp. 1053–1054
6 HCC Embedded. Available at http://www.hcc-embedded.com, accessed June 2013
7 Mendon, A.A., Schmidt, A.G., and Sass, R.: 'A hardware filesystem implementation with multidisk support', *Int. J. Reconfigurable Comput.*, 2009, **2009**, pp. 1–13