Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

A High Performance Reconfigurable Architecture for Flash File Systems

Irfan MIR [1], Alistair A. MCEWAN and Neil PERRINS

Embedded Systems Research Group (ESRG), Department of Engineering
University of Leicester
{im105,aam19,njp23}@le.ac.uk

Abstract. NAND flash memory is widely adopted as the primary storage medium in embedded systems. The design of the flash translation layer, and exploitation of parallel I/O architectures, are crucial in achieving high performance within a flash file system. In this paper we present our new FPGA based flash management framework using reconfigurable computing that supports high performance flash storage systems. Our implementation is in Verilog, and as such enables us to design a highly concurrent system at a hardware level in both the flash translation layer and the flash controller. Results demonstrate that implementing the flash translation layer and flash controller directly in hardware, by exploiting reconfigurable computing, permits us to exploit a highly concurrent architecture that leads to fast response times and throughput in terms of read/write operations.

Keywords. NAND flash, data storage, parallel architecture, reconfigurable computing, flash translation layer, flash controller, concurrency

Introduction

Flash memory is a solid state device commonly used in embedded systems such as digital cameras, multi-media cards (MMCs), cellular phones, USB memory sticks and PDAs. It has number of promising features such as no moving parts, low power consumption, fast access time and shock/temperature resistance. In the past few years the storage capacity of a flash memory has dramatically increased with decreased cost per cell due to fast growth in flash technology [1]. The two main types of flash memory are NOR and NAND. NOR flash memory is effectively used to store configuration data and Execute-In-Place (XIP) applications due to its random access feature. On the other hand, NAND flash memory is well-suited to use as a primary storage medium in embedded systems but can also be a part of XIP applications [2]. A generic solid state device (SSD) architecture consists of number of NAND flash memory chips [3]. Flash memory is manufactured under single level cell (SLC) and multi level cell (MLC) technologies [4].

NAND flash memory life depends upon erase/write operations which physically wear out the memory, unlike magnetic disks. A unique aspect of NAND flash memory is an erase-before-write operation—it is essential to erase the physical locations where new data is to be written. Such kind of operations are managed by a flash translation layer (FTL) [5] in NAND flash file systems. The core components of FTL are wear-levelling, address mapping and garbage collection. Wear-levelling in the FTL manages write, read and erase operations to increase the lifespan of the memory. The address mapping table maintains the mapping

[1]Corresponding Author: *Irfan Mir, Department of Engineering, University of Leicester, Leicester, UK. LE1 7RH*. E-mail: im105@le.ac.uk.

relationship between the logical and physical addresses and needs to be updated after every write operation. Garbage collection in the FTL manages the physical blocks having invalid data to make them free for further write operations. The FTL hides its design complexities from the upper software application layer where a file system such as FAT32/NTFS is implemented. Various algorithms and data structures have been proposed in [6,5,7,8] to improve address mapping schemes, wear-levelling and garbage collection for a flash file system.

The two types of flash file systems, raw NAND flash and embedded NAND flash, are discussed in [9]. For instance in the case of a cellular phone having raw NAND flash memory the FTL is implemented in embedded software as a part of a Flash File System (FFS). On the other hand in the case of a MMC having embedded NAND flash memory the FTL is implemented separately in firmware using an embedded controller. In this case a dedicated FFS written for embedded NAND flash memories does not give any extra benefits.

NAND flash controller (NFC) is also an important part of the storage architecture as it implements the low-level driver for a raw NAND flash chip where its I/O signal timings are managed according to the Open NAND Flash Interface (ONFI), an industry wide standard for the specification of NAND flash memory [10]. Bad Block Management and Error Correction Code (ECC) [11,12] are commonly implemented within the NFC to increase data integrity.

Multiple applications in embedded systems, such as cellular phones and MMCs having NAND flash-based storage architecture, can send multiple access requests concurrently to its storage architecture for write/read operations. These kinds of embedded systems, where the FTL is usually implemented in embedded software or in firmware, can limit the write/read performance due to their sequential behaviour. Due to this fact there is a desire to investigate the NAND flash-based storage architecture for improving the response times and throughput in terms of read/write operations.

The main motivation behind our approach is to exploit concurrency in flash-based storage architectures for quick response time and high throughput in term of read/write operations. This concurrent framework is viable to develop an efficient NAND flash-based file system where all major components such as wear-levelling, garbage collection, and flash controller have been incorporated at hardware level. We have designed and implemented these components in Verilog hardware description language (HDL). Unlike common practice, to implement the FTL using a high level language such as C/C++, we have developed the FTL in Verilog, truly implementing an embedded concurrent FTL along with NFC.

In this paper we present a new FPGA based flash management framework using reconfigurable computing, implemented in synthesizable Verilog HDL, which enhances the overall performance with optimised resource usage at hardware level. Our flash management framework works directly on raw NAND flash memories. To the best of our knowledge, no other work has been published utilising HDL to implement such kinds of NAND flash-based embedded file systems. This highly concurrent framework can be mapped on field programmable gate array (FPGA) fabric. Our evaluation results show that the approach permits us to exploit a highly concurrent architecture that leads to fast response times and throughput in terms of read/write operations in NAND flash-based storage systems.

The rest of this paper is organised as follows. The brief characteristics regarding NAND flash memory and its usage in storage architecture is described in section 1. In section 2 the design of new FPGA based flash management framework using reconfigurable computing is presented. The flash management framework is evaluated in section 3 where some results are presented. Finally our conclusions are discussed in section 4.

1. Background

1.1. Basics of NAND flash memory

The internal architecture of NAND flash memory, as shown in figure 1, consists of fixed number of physical blocks where each block has fixed number of pages. The page layout is divided into two parts, user data and metadata. The metadata area is used to store useful information regarding the corresponding page data such as logical block address, logical page address, error correction code, erase count number, various flags (valid/invalid data and free/occupied page) and bad block detection.

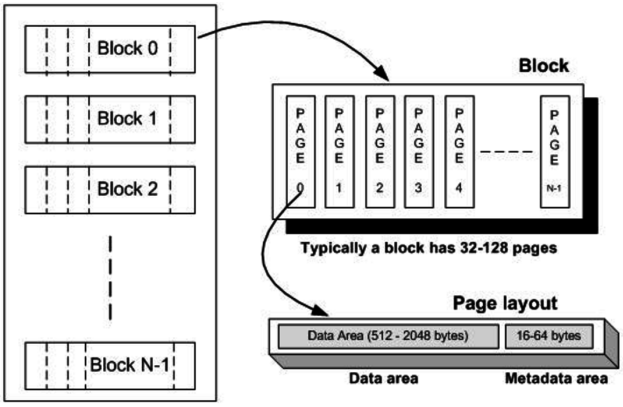

Figure 1. Internal architecture of a NAND flash memory.

The raw NAND flash memory mainly allow three operations to manage data which are program (write), read and erase. There is a constraint that the program and the read operations are performed at the page level, while the erase operation is executed at the block level only. According to the program page operation first the start command code including physical page and block address is inserted through the I/O pins of NAND flash memory. After that the page data is loaded into the internal cache register of NAND flash memory. Finally after inserting the end command code the device will go busy for a fixed program delay time (typically $200 - 700\mu sec$). At this stage the data has been programmed into the assigned physical page which can be confirmed by reading the current status of NAND flash memory. According to the read page operation the start command code including physical page and block address is inserted through the I/O pins of NAND flash memory. Then after inserting the end command code the device will go busy for a fixed read delay time (typically $20 - 50\mu sec$). At this stage the page data can be read from the cache register of NAND flash memory. According to the erase block operation the start command code including physical block address is inserted through the I/O pins of NAND flash memory. Then after inserting the end command code the device will go busy for a fixed erase delay time (typically $1.5 - 3msec$). Hereafter all the pages of the assigned physical block have been set as '1'. The NAND flash memory allows a limited number of erase operations for data reliability which is typically $10,000 - 100,000$.

1.2. Flash translation layer design

To improve the performance and lifespan of a NAND flash memory it is essential to consider an efficient design of flash translation layer (FTL). A typical FTL manages and distributes the data evenly over the NAND flash memory to increase its lifespan. Wear levelling is usually implemented by managing address mapping table. To build an address mapping table there are three kinds of mapping schemes are used page mapping, block mapping and hybrid

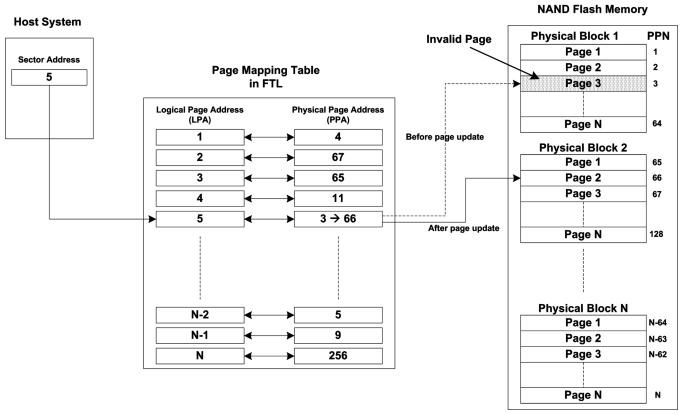

Figure 2. Page mapping scheme in FTL.

mapping [6]. In our proposed approach the page mapping scheme is used, so only page mapping is discussed here. In the page mapping scheme, data is read or written on a page basis in a NAND flash memory. There is a mapping table where the logical page address (LPA) is translated into a physical page address (PPA) for data read/write. Static random access memory (SRAM) is used to store the page mapping table. The write request including LPA comes from the upper application layer which is used as an address bus for a SRAM. In case of each write request the page mapping table assigns a new PPA to a LPA and stores it in the corresponding location in SRAM. In the case of an overwrite request the page mapping table mainly performs two actions one is to assign a new PPA to a LPA and second is to invalidate the previous one.

The complete picture of this translation is shown in figure 2 where the data needs to be written at various LPAs 5, 3, 5 and 2 in sequence. First to write data into NAND flash memory the LPA 5 of page data is first read by FTL. The page mapping table assigns a new PPA (say 3) for LPA 5. Hereafter the page data has been written at PPA 3 of NAND flash memory. The same process will be repeated to write page data for LPA 3. Now to overwrite data for LPA 5 the page mapping table assigns a new PPA (say 66) and invalidates its old PPA 3. Finally to write data for LPA 2 a new PPA (say 67) is generated.

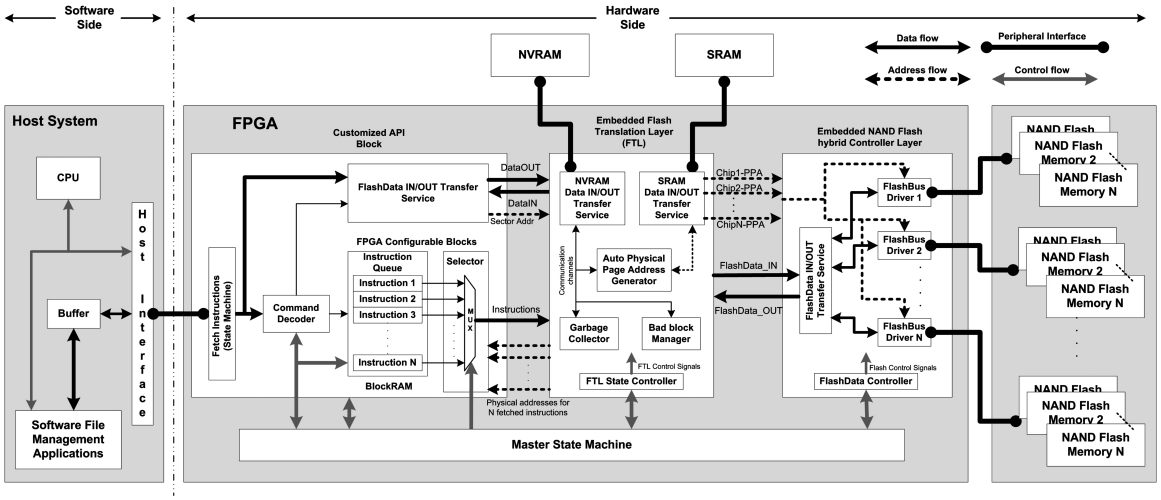

Figure 3. Flash management architecture.

2. A new flash management framework

In this section we present the new flash management framework using reconfigurable computing, including the basic design, user data/instruction transfer protocol, embedded hybrid flash translation layer and raw NAND flash controller layer. This framework is demonstrated by the practical implementation of the major algorithms of flash file system on a high-performance reconfigurable computing platform (such as FPGA), which is capable of supporting high-speed data management to run enhanced software management applications.

2.1. Design overview

The overall system, as shown in figure 3, consists of software and hardware parts. At the software side the host system transfers the I/O requests from the software file management applications to the flash management architecture. The host system's CPU has a 32-bit bi-directional data bus, 19-bit address bus and a 4-bit control bus. The maximum throughput of the CPU is considered as 50 MB/sec. At the hardware side the FPGA based flash management architecture mainly consists of a master state machine, embedded flash translation layer, embedded hybrid NAND flash controller layer, and the customised application programming interface block where the user data/instruction transfer protocol is implemented. To increase the storage capacity multiple NAND flash memories can be used on the same bus with their own chip select lines.

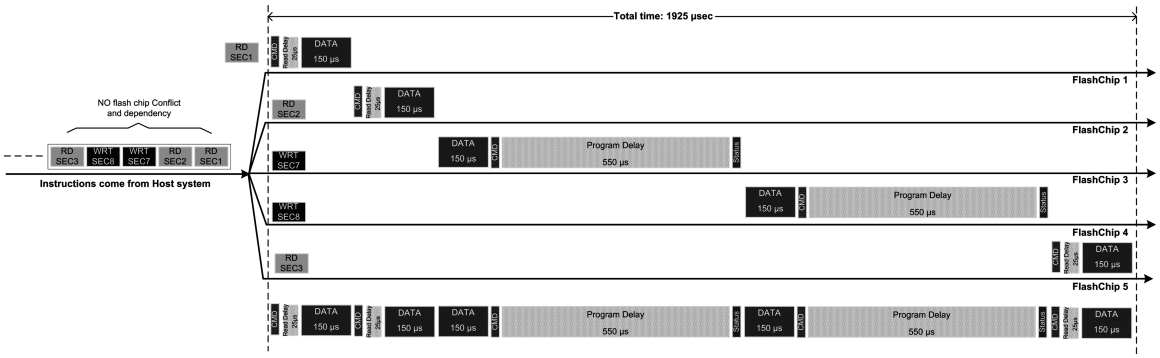

Figure 4. Timing diagram for 5 fetched instructions with no flash chip conflict or dependency under the conventional approach.

In the case of a conventional approach to Handling a number of write and read operations on a single flash bus having multi NAND flash chips, the instructions are executed in sequential manner which is shown in figure 4. On the other hand in our flash management framework the major components operate concurrently. In figure 5 and 6 the timing diagrams of sector write and read operation are illustrated which shows the data latency from host system to raw NAND flash memory. It can be observed that the concurrency is exploited in these operations where embedded FTL and NAND controller work in a parallel fashion. We manage the minimum data latency under the limitations of NAND flash memory such as fixed program delay time etc.

The master state machine manages the control bus of the flash management architecture. In idle state it checks the Chip Enable (CE) signal with Write Enable (WE) signal or Read Enable (RE) signal from the host system for checking sector data write/read request to the NAND flash memory. Hereafter the state machine generates the particular control signals for all blocks of the flash management architecture on the basis of commands which have been sent by the host system. These commands are decoded by a customised programming interface block to build a communication channel between the host system and the flash management architecture.

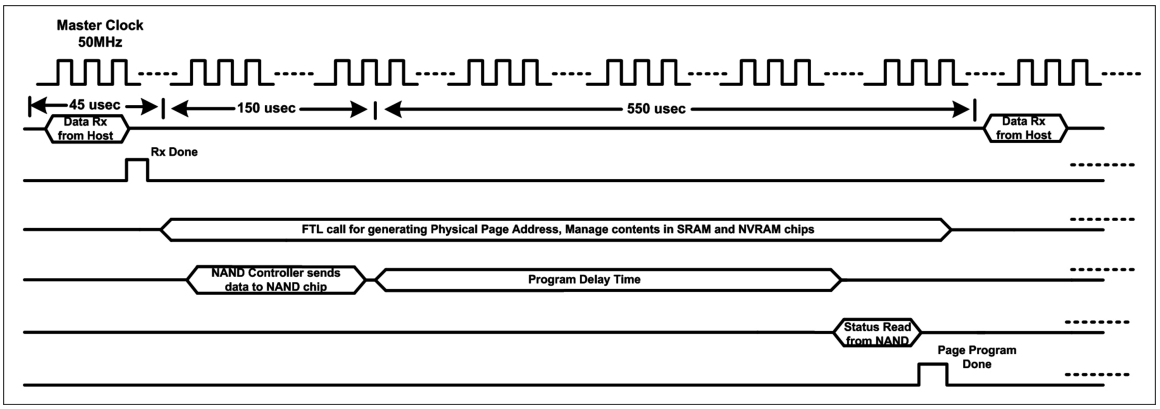

Figure 5. Timing diagram of a sector write operation.

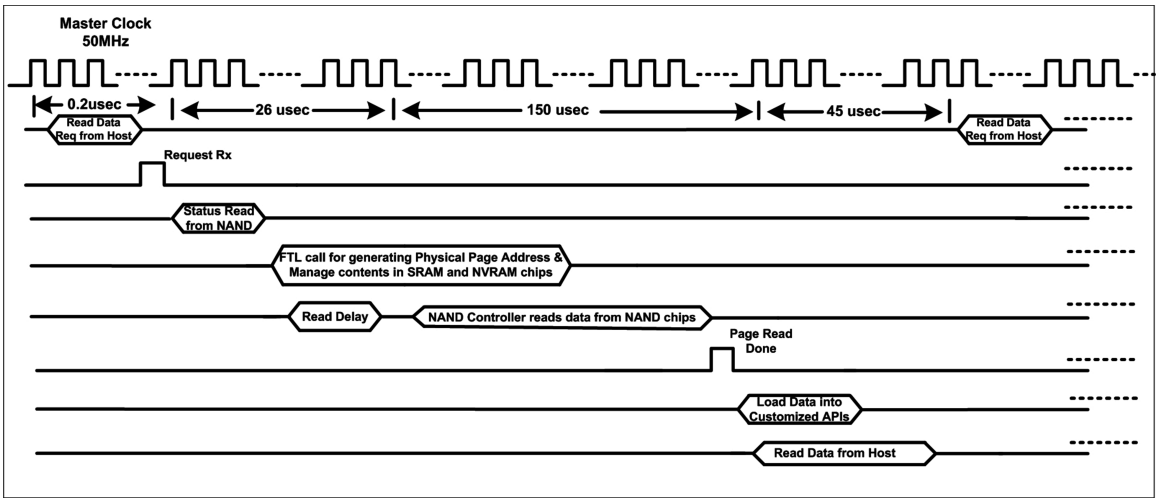

Figure 6. Timing diagram of a sector read operation.

2.2. Customised application programming interface block

The customised application programming interface (API) block is developed to build a channel communication between host system and flash management architecture which is illustrated in figure 7. The host system can perform four kinds of operations (write, read, erase and status), which are defined by the commands that are shown in table 1.

It mainly consists of five parts; fetch instructions logic, command decoder, flash data IN/OUT transfer service, instruction queue and instruction selector. The fetch instructions logic is based on a state machine which is designed to fetch the instructions from the host system whenever it is required. Here a handshake protocol is implemented which synchronises the data communication between the host system and the flash management framework. The command decoder decodes the instructions (read, write, erase and status) and passes them to the instruction queue. Meanwhile the data corresponding to the sector write instruction passes to the flash data IN/OUT transfer service block where it is stored in a parameterised write buffer. The data transfer service contains two parameterised buffers, one for read and one for write. In our design we utilise the block RAMs for these buffers to save the gate resources of an FPGA. In the read buffer the sector read data from the NAND flash memory is stored. In write buffer the sector data is stored which needs to be written in flash memory. The depth of write and read buffers are programmable and can be changed through a software command or on-board switches. In our implementation we limit the size of write and read buffers on

the basis of number of flash buses where each flash bus can have number of flash chips. The size of write and read buffers will be $PAGE\_SIZE * M * N$. Here M is number of flash buses and N is number of flash chips per flash bus.

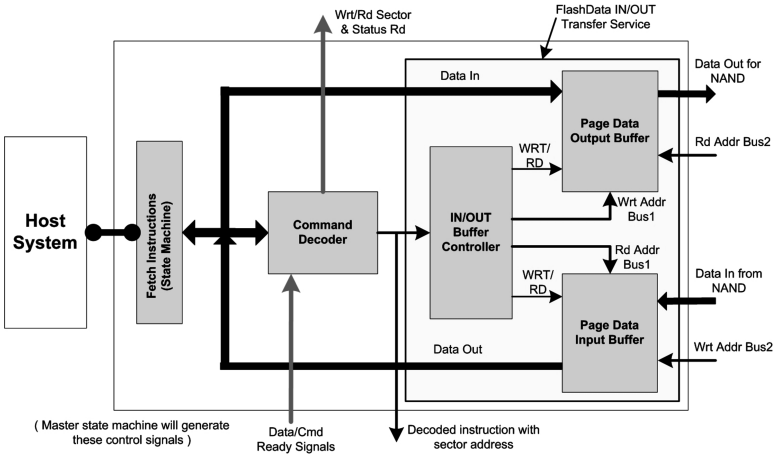

Figure 7. Channel communication between host system and flash management framework.

For a sector write operation the host system sends the sector data with a write command to the flash management architecture through the 32-bit bi-directional data bus in 45 microseconds. In case of a sector read operation the data is read from flash management architecture in 40 microseconds. In a status read operation the host system can check the current state of all NAND flash memories in MxN flash array. For better and reliable communication the status should be read first before performing sector write or read operations.

Table 1. Host system commands.

Command name	Command code	Description
Sector Write	24'h1_SectorAddress	The host system sends this command to flash management framework for writing a sector data having 2, 4 or 8KB. In the 24-bits command code the most significant nibble is used as a write sector code and rest of them are used for the sector address.
Sector Read	24'h2_SectorAddress	The host system sends this command to flash management framework for reading a sector data having 2, 4 or 8KB. In the 24-bits command code the most significant nibble is used as a read sector code and rest of them are used for the sector address.
Block Erase	24'h3_BlockAddress	The host system sends this command to flash management framework for erasing the physical block of a particular NAND flash chip in a flash bus. In the 24-bits command code the most nibble is used as erase block code and rest of them are used for the erase block address.
Status read	24'h4_xxxxx	The host system sends this command to flash management framework for reading the current status of NAND flash chips per flash bus. In the 24-bits command code the most significant nibble is used as a status read code and rest of them are used for the flash bus address.

In the case of an MxN flash array our flash management framework has the ability to execute maximum N write/read non-dependent instructions per flash bus. We adopt a policy to process the instructions per flash bus where the write instructions always start first. After starting the write instructions we process the read instructions. In case of chip conflict,

which can only happen between write and read instructions, the read instruction always executes after its corresponding write instruction. In our flash management framework a chip conflict cannot occur among write instructions in N write/read non-dependent instructions due to the embedded flash translation layer where the write instructions always execute on different chips. In the case of every flash chip conflict between any two fetched write and read instructions, a new instruction will be fetched.

In our flash management framework there is an instruction queue where a fixed number of instructions is fetched at a time for fast data management. Here an instruction queue is a reconfigurable Block RAM where its depth can be controlled through a software command or on-board switches. In our design we limit the maximum size of an instruction queue on the basis of the number, N, of flash chips per flash bus. After fetching all instructions in the instruction queue they are passed to the selector where these instructions are rearranged and passed to the flash translation layer. The selector's control bus is controlled by the master state machine where the fetched instructions are scheduled so that the write instructions are performed first followed by the read instructions ending with any read instructions where a chip conflict occurred with a write instruction.

In the case of write/read non-dependent instructions we achieve the maximum throughput and minimum response time per a complete instruction queue compared to a conventional implementation. Even in the case of an erase instruction per a complete instruction queue our management architecture shows tremendous performance.

2.3. Embedded flash translation layer

The embedded flash translation layer (FTL) mainly consists of an auto physical page address generator, static random access memory (SRAM) controller, non-volatile random access memory (NVRAM) controller, garbage collector and bad block manager as illustrated in figure 8. These blocks are managed by a local state machine. This state machine is controlled by one master state machine of the flash management architecture.

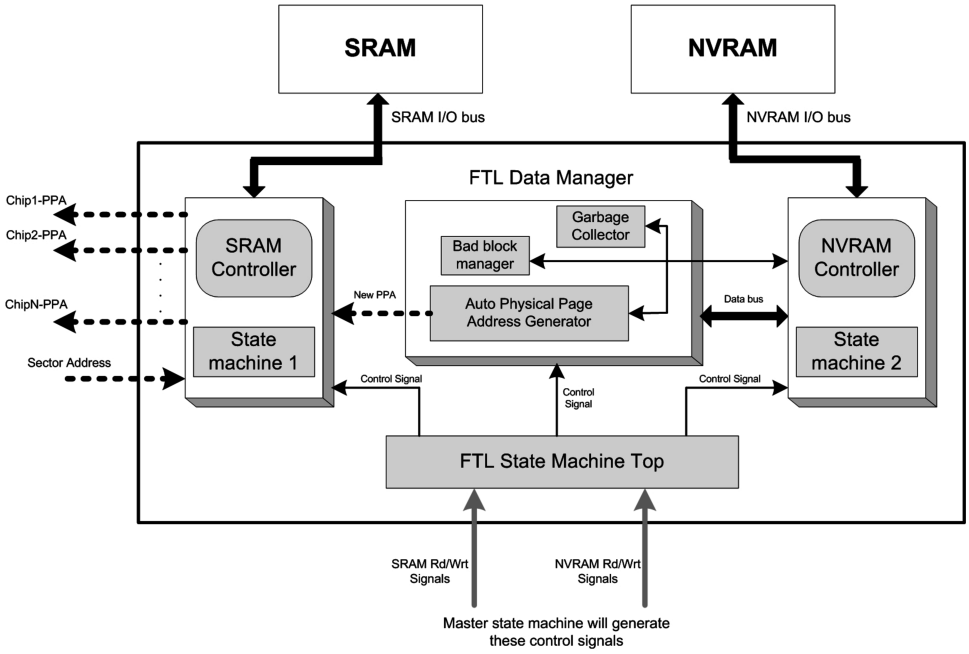

Figure 8. Block diagram of concurrent embedded FTL.

It is important to keep a record of the status of all the pages of NAND flash memory where each page can have a status such as free, occupied, valid or invalid. There can be

various options to maintain the page status. In the first option the page status can be written in the metadata area of the page in NAND flash memory which can be expensive in terms of write operations especially in the case of invalid page status. In the second option the page status can be written in SRAM along with page mapping table which can be lost in case of sudden power loss shutdown. To avoid these concerns we select NVRAM in embedded FTL design for updating the status of all pages of NAND flash memory. We assign two bits in NVRAM for the status of each page of NAND flash memory where one bit is for free/occupied and second is for valid/invalid.

All blocks in the embedded FTL work in a concurrent fashion. First of all the master state machine sends a control signal to the local state machine of the embedded FTL when the sector write/read requests in the instruction queue of the customised API block are received from the host system. The local state machine generates a control for the auto physical page address generator to generate a new physical page address (PPA) for each sector write request in the instruction queue. For a sector read request the auto physical page address generator is not involved in the operation.

In our management architecture we use one NVRAM per flash bus. The size of each NVRAM is based on N logical partitions where each partition is reserved for a NAND flash chip in a flash bus. The size of each partition is almost twice the total number of pages of the NAND flash chip. A bit is reserved for each page in NVRAM which indicates whether this page is free to program or not. We set '0' for a free page and '1' for an occupied page. The contents of the NVRAM are managed by the page program operation, bad block manager and garbage collector. At the start up the bad block manager detects the bad blocks in the NAND flash memory. The corresponding bits of all the pages belonging to these bad blocks are set as '1' permanently in NVRAM. In another scenario right after the confirmation of the page program operation the reserve bit of that page is also set as '1'. The garbage collector constantly monitors the invalid pages in concurrent fashion and sends request to the master state machine to erase a physical block of NAND flash memory, if required. Once a block erase operation has completed a command is sent to NVRAM controller for setting the corresponding bits as '0' of all the pages belong to the erased physical block. The NVRAM controller manages the I/O signal timings of the NVRAM by using its local state machine which is controlled by the FTL's state machine.

The SRAM controller maintains the page mapping table in SRAM. The sector write/read address is used as an address bus of a SRAM. On every sector write operation a new PPA from the auto physical page address generator is written into SRAM at its corresponding sector address which is treated as a logical page address (LPA). Further this LPA is written in the metadata area of the page with the page data in NAND flash memory during the page program operation. Such kinds of records of LPAs in the metadata area are used in reconstructing the page mapping table at the start up of the FFS. When the embedded FTL receives the sector read request, the corresponding PPA of this sector is read from the SRAM at its given a sector address. Like the NVRAM controller the SRAM controller also manages the I/O signal timings of SRAM by its local state machine which is controlled by the FTL's state machine.

2.4. Embedded NAND flash hybrid controller

The embedded NAND flash hybrid controller mainly consists of a flash data controller, the flash data IN/OUT transfer service and low-level flash drivers. The flash data controller manages the sequence of instructions by monitoring the ready/busy flags of all NAND flash chips. Here the flash data IN/OUT transfer service of the customised API block is used again. Its functionality and features have already been explained in section 2.2. The embedded NAND flash hybrid controller mainly implements the low-level driver for raw NAND flash chips.

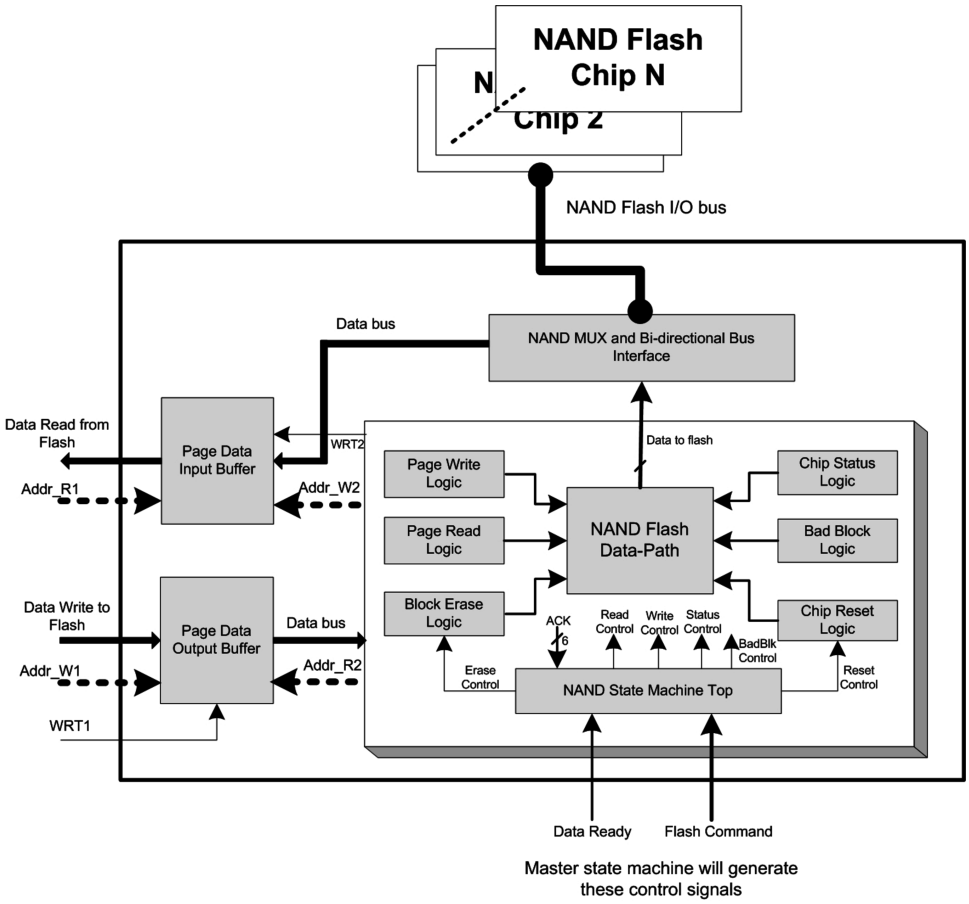

Figure 9. Block diagram of embedded NAND flash controller.

It consists of M flash driver blocks where M is the number of flash buses. The internal architecture of each flash driver consists of two buffers, Data-Path, Write, Read, Erase, Status, Bad Block and Reset logic blocks which are controlled by a local state machine. It is illustrated in figure 9. The Data-Path logic block assigns the required data patterns to the 8-bit bi-directional data bus of NAND flash memory according to the state of the storage system. Rest of the logic blocks generate the I/O control signal timings for a raw NAND flash memory according to the ONFI standard [10].

The two parameterised buffers having same size as the buffers used in customised API block are designed to be used concurrently for page write and read operations. One of them is used to store read page data from the NAND flash memory which is accessed by the host system through customised application programming interface block. Second is used to store the page data from the host system which needs to be written in NAND flash memory.

2.5. Verilog implementation

Field Programmable Gate Array (FPGA) is a reconfigurable device which is well suited to data-paths and control designs. An FPGA consists of an array of configurable logic blocks (CLBs) where concurrent architecture can be easily mapped. To implement our flash management architecture, as described in section 3, on FPGA fabric we need to develop it in register transfer level (RTL) using Verilog hardware description language (HDL). For this purpose we developed and tested the synthesizable Verilog cores for embedded FTL, embedded NAND flash controller and customised APIs separately. After that we integrated them to develop a single Verilog soft core of the flash management framework for a NAND flash-based

file system. This RTL design is fully synthesizable and can be easily mapped on any FPGA fabric.

It is a fully synchronous design which is based on a single master clock and state machines to implement the concurrent flash management architecture. To maintain synchronisation in this architecture was a challenging task which was tackled by implementing state machines.

3. Performance evaluation

In this section we present the performance evaluation of our flash management architecture for a NAND flash-based file system. We create three scenarios for performance evaluation of our flash management framework. In the first scenario the write and read instructions without chip conflict are performed per flash bus where each flash bus has 5 NAND flash chips. In the second scenario the write and read instructions having a couple of chip conflicts are processed per flash bus. In the third scenario the write and read instructions including an erase instruction with a couple of chip conflicts are processed per flash bus. For these scenarios we consider a NAND flash chip based on 8-bit data bus and have 128 pages per erase block where the page size is 4KB. The page read, page program and block erase delay timings are considered 25 microseconds, 550 microseconds and 1.5 milliseconds respectively. Further the data transfer time of a flash page is considered as 150 microseconds.

3.1. Response time under various scenarios

Here we discuss three scenarios to analyse the timing of write/read/erase instructions which are processed under our flash management framework. The ideal and worst cases are presented to determine the average response time. It is seen that in all scenarios the average response time is much better than the conventional design process. We manage the minimum data latency under the limitations of NAND flash memory such as fixed program delay time etc.

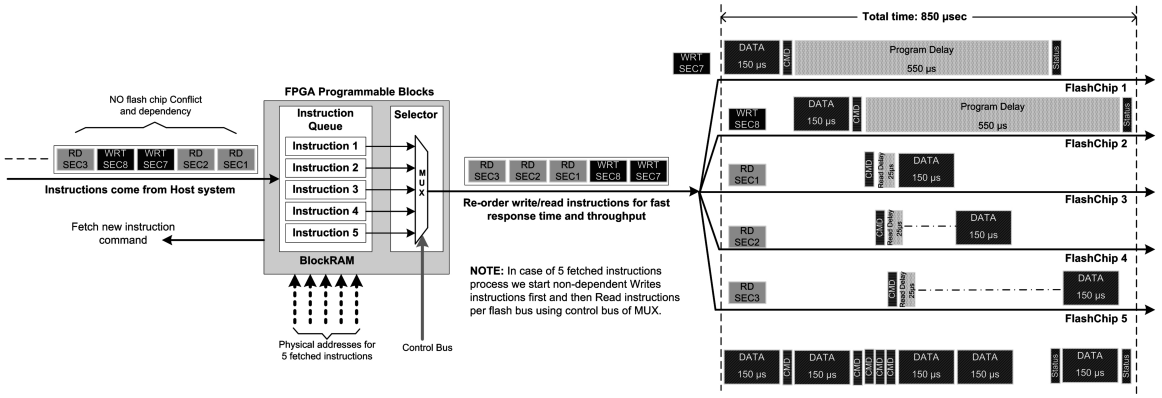

Figure 10. Timing diagram for 5 fetched instructions with no flash chip conflict or dependency under our flash management framework.

In figure 10 the timing diagram of non-dependent multi sector write and read operations per flash bus having no chip conflicts is illustrated. It is an ideal scenario where the concurrency is exploited in sector write and read operations. In figure 11 the timing diagram of multi sector write and read operations per flash bus having chip conflicts is illustrated. Under this scenario limited concurrency is exploited in sector write and read operations. In figure 12 the timing diagram of non-dependent multi sector write or read operations including an

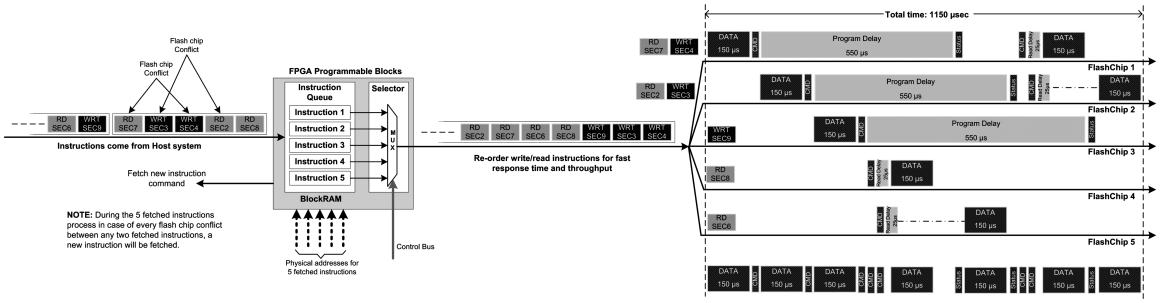

Figure 11. Timing diagram for 5 fetched instructions with flash chip conflict and dependency under our flash management framework.

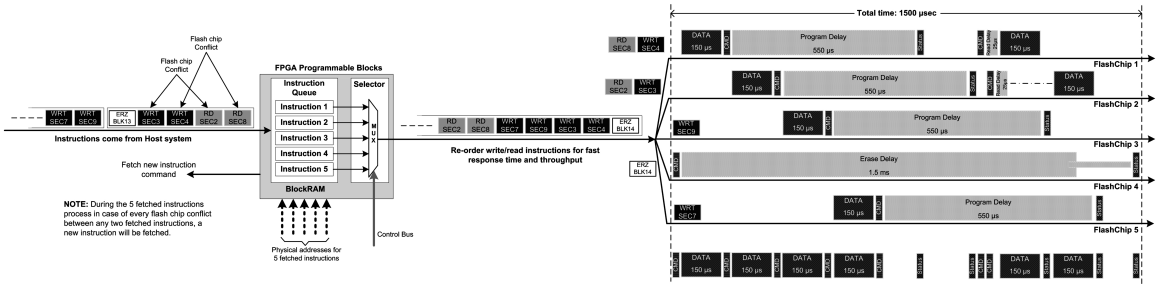

Figure 12. Timing diagram for 5 fetched instructions including an erase with flash chip conflict and dependency under our flash management framework.

erase instruction per flash bus with some chip conflicts is illustrated. These timing diagrams show the data latency from host system to raw NAND flash memories per flash bus in our flash management framework.

3.2. High-speed write/read operations performance

The figure 13 shows the comparison between the conventional approach and our proposed approach in terms of throughput using one and two flash buses. It shows the throughput of non-dependent multi write and read operations under conventional approach and the scenarios which are mentioned above. For these experiments the 4 KB sector size is used with two flash buses where each bus has 5 NAND flash chips which have a shared 8-bit data bus and 4 KB page size. The results are extracted by comparing the conventional approach with the three scenarios under our approach in terms of throughput. To achieve such kind of improvements our approach effectively utilises the program delay time of page data which is much greater than the read delay time of page data in NAND flash memory.

We can get higher throughput for multi write and read operations if the sector size is equal to the flash page size in a single flash bus scenario. On the other hand the throughput goes at maximum level if the sector size is equal to the sum of flash page sizes of one of the NAND flash chips per flash bus in multi flash bus scenario. The performance gradually decreases as the sector size starts increasing or decreasing for both write and read operations.

4. Conclusions

In this paper we have presented the design of a FPGA-based flash management framework for NAND flash-based file systems. It is observed that the response time and throughput of write/read operations can be degraded in the NAND flash-based storage architecture in embedded systems due to the software implementation of FTL design. By exploiting the con-

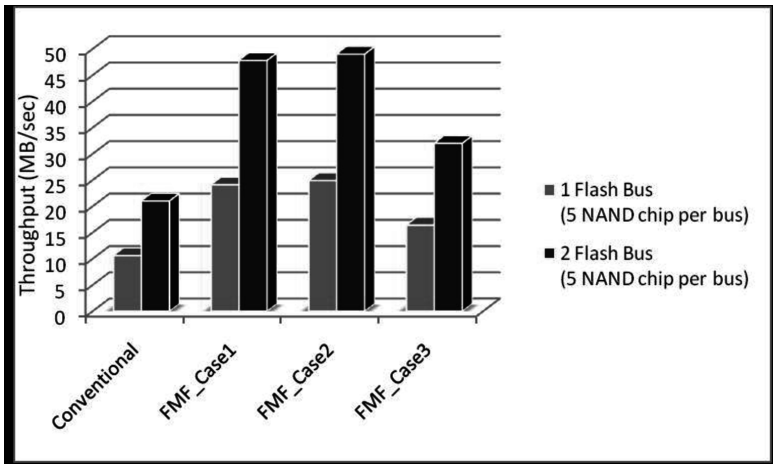

Figure 13. Comparison of our flash management architecture using three scenarios with conventional architectures in term of throughput.

currency in the FTL design and flash controller we can improve the throughput and access time of multiple applications in embedded systems having NAND flash-based storage architecture.

The new flash management framework is a viable way to develop a NAND flash-based file system where all major components such as address mapping, wear-levelling, garbage collector, bad block manager, and flash driver have been incorporated at a hardware level. Our synthesizable Verilog implementation of flash management architecture is flexible and can be mapped onto any FPGA fabric. Compared to conventional approach, our flash management framework provides a high throughput and reduced response time for NAND flash-based storage systems for the same overhead. We are building a parameterised test bench to evaluate our flash management framework in all possible scenarios under the synthetic and real workloads.

References

[1] Hwang Chang-gyu. Nanotechnology enables a new memory growth model. In *Proceedings of the IEEE*, volume 91, pages 1765–1771. IEEE, nov 2003.

[2] C. Park, J. Seo, D. Seo, S. Kim, and B. Kim. Cost-efficient memory architecture design of NAND flash memory embedded systems. In *21st International Conference on Computer Design*, pages 474–480, oct 2003.

[3] J. D. Davis and L. L. Zhang. FRP: A Nonvolatile Memory Research Platform Targeting NAND flash. In *Proceedings of the First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, WISH '09. ACM, 2009.

[4] Micron. Intel, Micron First to Sample 3-Bit-Per-Cell NAND Flash Memory on Industry-Leading 25-Nanometer Silicon Process Technology. Press Release, August 2010. http://news.micron.com/releaseDetail.cfm?ReleaseID=499901.

[5] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. Application Note 684, Intel Corporation, December 1998.

[6] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, June 2005.

[7] Amir Ban. Flash File System, US patent 5404485, April 1995.

[8] Po-Liang Wu, Yuan-Hao Chang, and Tei-Wei Kuo. A file-system-aware FTL design for flash-memory storage systems. In *Design, Automation and Test in EuropeConference and Exhibition*, DATE '09, pages 393–398. DATE '09, IEEE, 2009.

[9] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based ap-

plications. *ACM Transactions in Embedded Computing Systems*, 7:38:1–38:23, August 2008. http://doi.acm.org/10.1145/1376804.1376806.

[10] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0. ONFi Workgroup, Hynix Semiconductor, 2006. http://www.onfi.org.

[11] Samsung Elecronics Co. NAND Flash ECC Algorithm. Samsung Elecronics, June 2004.

[12] Te-Hsuan Chen, Yu-Ying Hsiao, Yu-Tsao Hsing, and Cheng-Wen Wu. An Adaptive-Rate Error Correction Scheme for NAND Flash Memory. In *VLSI Test Symposium*, pages 53–58, Los Alamitos, CA, USA, 2009. IEEE Computer Society. http://doi.ieeecomputersociety.org/10.1109/VTS.2009.24.